UNIT 5B
Binary Search

Course Announcements 1

- Sunday's review sessions GHC 4303
  - Session 1: 6-8 pm
  - Session 2: 8-10 pm.
  - Sample exam done by CAs and questions from students (sample exam available at http://www.cs.cmu.edu/~15110-s13/schedule.html)

Course Announcements 2

- Monday office hours 5-10 at GHC 4215, NOT in clusters
- Exam information
  - 2:30 exam: Sections A, B, C, D, E go to Rashid (GHC 4401) and sections F, G go to PH 125C.
  - 3:30 exam: Sections H, I, J, K, L, M, N all go to Rashid (GHC 4401) .
- Bring your CMU id!

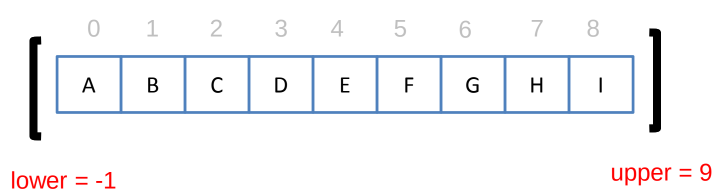
This Lecture

- A new search technique for arrays called binary search
- Application of recursion to binary search
- Logarithmic worst-case complexity

Binary Search

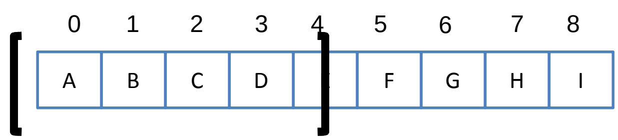
- Input: Array A of n unique elements.
  - The elements are sorted in increasing order.
- Result: The index of a specific element called the key or nil if the key is not found.
- Algorithm uses two variables lower and upper to indicate the range in the array where the search is being performed.
  - lower is always one less than the **start** of the range
  - upper is always one more than the **end** of the range

Example



List already sorted in ascending order.
Suppose we are searching for D.

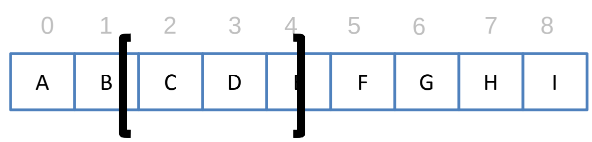
Divide and Conquer



lower = -1

upper = 4

Divide and Conquer



lower = 1

upper = 4

and so on ...

Each time we look at a smaller portion of the array within the window and ignore all the elements outside of the window

Algorithm

1. Set lower = -1.
2. Set upper = the length of the array a
3. Return BinarySearch(list, key, lower, upper).

BinarySearch(list, key, lower, upper):

1. Return nil if the range is empty.
2. Set mid equal the midpoint between lower and upper
3. Return mid if a[mid] is the key you're looking for.
4. If the key is less than a[mid] then
   return BinarySearch(list,key,lower,mid)
   Otherwise, return BinarySearch(list,key,mid,upper).

Example 1: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Found: return 9

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Not found: return nil

Finding `mid`

- How do you find the midpoint of the range?

  mid = (lower + upper) / 2

  Example: lower = -1, upper = 9

  (range has 9 elements)

  mid = 4
- What happens if the range has an even number of elements?

Range is empty

- How do we determine if the range is empty?

lower + 1 == upper

Reccursive Binary Search in Ruby

```
def bsearch(list, key)
  return bs_helper(list, key, -1, list.length)
end
def bs_helper(list, key, lower, upper)
  return nil if lower + 1 == upper
  mid = (lower + upper)/2
  return mid if key == list[mid]
  if key < list[mid] then
    return bs_helper(list, key, lower, mid)
  else
    return bs_helper(list, key, mid, upper)
  end
end
```

Example 1: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

```
                    key  lower  upper
bs_helper(list, 73,   -1,    15)
                    mid = 7 and 73 > a[7]
bs_helper(list, 73,    7,    15)
                    mid = 11 and 73 < a[11]
bs_helper(list, 73,    7,    11)
                    mid = 9 and 73 == a[9]
                    ---> return 9
```

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

```
                    key  lower  upper
bs_helper(list, 42,   -1,    15)
                              mid = 7 and 42 < a[7]
bs_helper(list, 42,   -1,     7)
                              mid = 3 and 42 > a[3]
bs_helper(list, 42,    3,     7)
                              mid = 5 and 42 < a[5]
bs_helper(list, 42,    3,     5)
                              mid = 4 and 42 > a[4]
bs_helper(list, 73,    4,     5)
                              lower+1 == upper
                              --->    Return nil.
```

Instrumenting Binary Search

```
def bsearch(list, key)
  return bs_helper(list, key, -1, list.length, 1)
end

def bs_helper(list, key, lower, upper, count)
  print “iteration\t”, “lower\t” + "upper\t\n”
  print iteration, “\t”, lower, upper, “\t\n”
  return nil if lower + 1 == upper
  mid = (lower + upper)/2
  return mid if key == list[mid]
  if key < list[mid] then
    return bs_helper(list, key, lower, mid, count + 1)
  else
    return bs_helper(list, key, mid, upper, count + 1)
  end
end

a = TestArray.new(100).sort
```


Iterative Binary Search in Ruby

```
def bsearch(list,key)
  lower = -1
  upper = list.length
  while true do
    mid = (lower+upper) / 2
    return nil if upper == lower + 1
    return mid if key == list[mid]
    if key < list[mid] then
      upper = mid
    else
      lower = mid
    end
  end
end
```

Analyzing Efficiency

- For binary search, consider the worst-case scenario (target is not in array)
- How many times can we split the search area in half before the array becomes empty?
- For the previous examples:
  15 --> 7 --> 3 --> 1 --> 0 ... 4 times

In general...

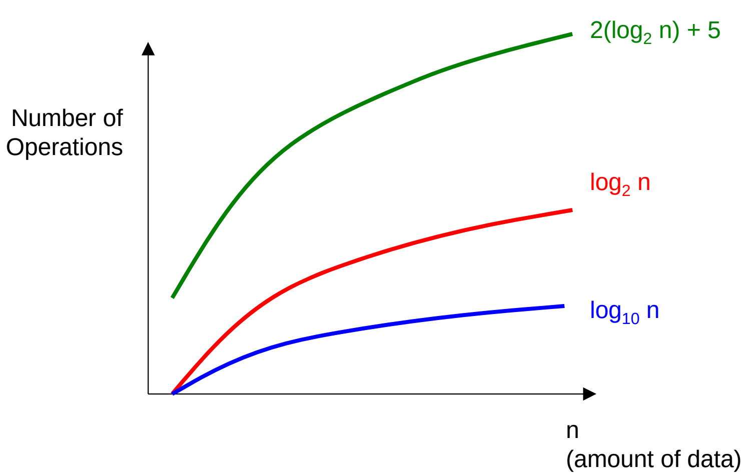
- Recall the log function:
  $\log_a b = c$ is equivalent to **$a^c = b$**
  Examples:
  $\log_2 128 = 7$
  $\log_2 n = 5$ implies $n = 32$
- In general, we can split search region in half $\lfloor \log_2 n \rfloor + 1$ times before it becomes empty.
- In our example: when there were 15 elements, we needed 4 comparisons: $\lfloor \log_2 15 \rfloor + 1 = 3 + 1 = 4$

Big O

- In the worst case, binary search requires O(log n) time on a sorted array with n elements.
  - Note that in Big O notation, we do not usually specify the base of the logarithm. (It's usually 2.)
- Number of operations / Order of Complexity

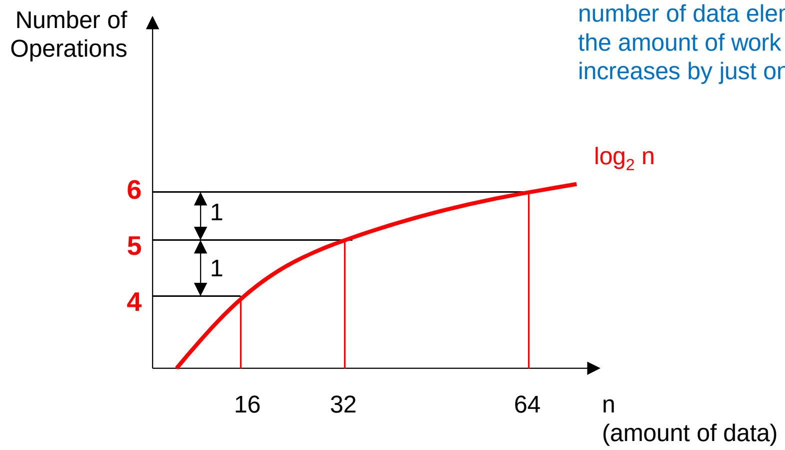
Number of operations	Order of Complexity
$\log_2 n$	O(log n)
$\log_{10} n$	O(log n)
$2(\log_2 n) + 5$	O(log n)

O(log n) ("logarithmic")



O(log n)

For a $\log_2$ algorithm,
If you double the
number of data elements
the amount of work you do
increases by just one unit



Binary Search (Worst Case)

Number of elements	Number of Comparisons
15	4
31	5
63	6
127	7
255	8
511	9
1023	10
1 million	20

Binary Search Pays Off

- Finding an element in an array with a million elements requires only 20 comparisons!
- BUT....
  - The array must be sorted.
  - What if we sort the array first using insertion sort?
    - Insertion sort $O(n^2)$ (worst case)
    - Binary search $O(\log n)$ (worst case)
    - Total time: $O(n^2) + O(\log n) = O(n^2)$

    Luckily there are faster ways to sort in the worst case...