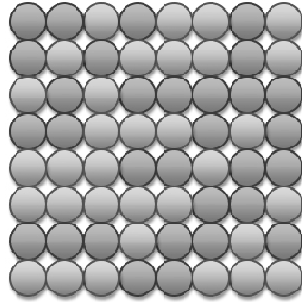


UNIT 5B
Binary Search

Binary Search

- Required: Array A of n unique elements.
  - The elements must be <u>sorted</u> in increasing order.
- Result: The index of a specific element (called the <u>key</u>) 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

Algorithm

1. Set lower = -1.
2. Set upper = the length of the array
3. Repeat the following:
   a. Return nil if the range is empty. You're done.
   b. Set mid = the midpoint between lower and upper
   c. Return mid if a[mid] is the key you're looking for. You're done.
   d. If the key is less than a[mid], set upper = mid. Otherwise, set lower = mid.

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
                        66 73 74 79 83 91 95

 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
                        66 73 74

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								

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				41	48	58								

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				41										

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

`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
```


Binary Search in Ruby: Iteratively

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

```
lower     upper     mid
-1        15        7     73 > a[7]
                          Set lower = mid
7         15        11    73 < a[11]
                          Set upper = mid
7         11        9     73 == a[9]
                          Return 9
```

Binary Search in Ruby: Recursively

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

Analyzing Efficiency

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

In general...

- In general, we can split search region in half $\lfloor \log_2 n \rfloor + 1$ times before it becomes empty.
- Recall the log function:

  $\mathbf{\log_a b = c}$ is equivalent to $\mathbf{a^c = b}$

  Examples:

  $\log_2 128 = 7$

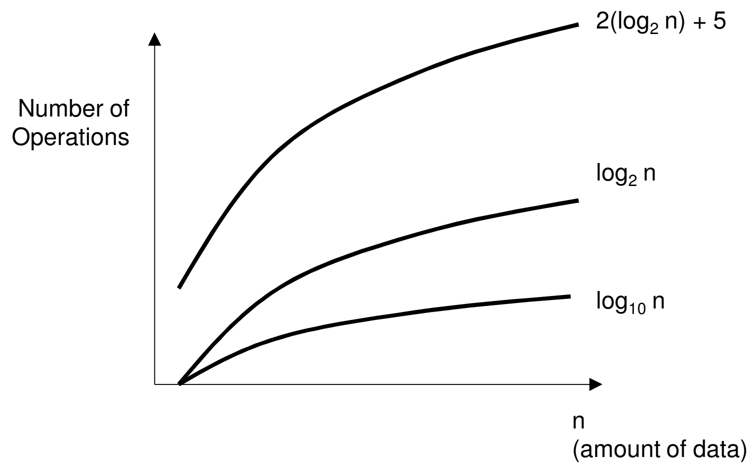
  $\log_2 n = 5$ implies $n = 32$
- In our example: when there were 15 elements, we needed 4 comparisons: $\lfloor \log_2 15 \rfloor + 1 = 3 + 1 = 4$

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.)
- <u>Number of operations</u> <u>Order of Complexity</u>

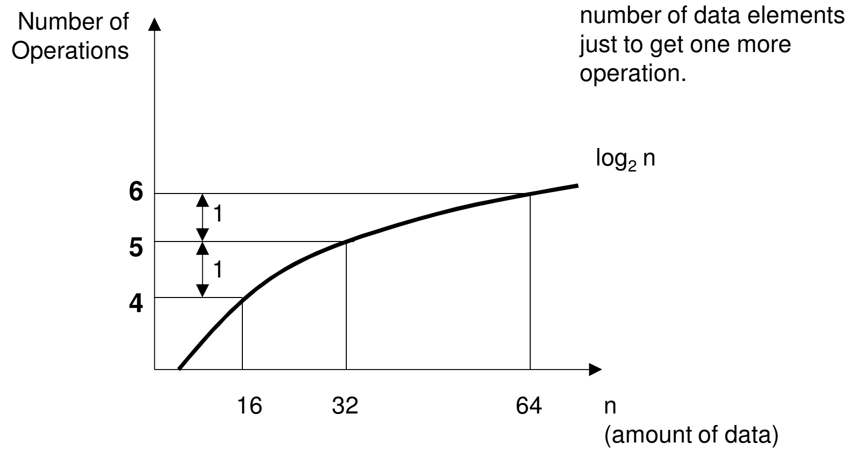
Number of operations	Order of Complexity
$\log_2 n$	O(log n)
$\log_{10} n$	O(log n)
$2(\log_2 n) + 5$	O(log n)

O(log n) (“logarithmic”)



O(log n)

For a $\log_2$ algorithm, you have to double the number of data elements just to get one more operation.



Binary Search (Worst Case)

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

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...