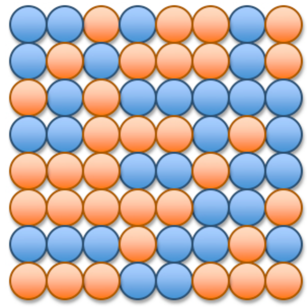


UNIT 5B
Binary Search

Binary Search

- Required: List L of n unique elements.
  - The elements must be sorted in increasing order.
- Result: The index of a specific element (called the key) or None if the key is not found.
- Algorithm uses two variables *lower* and *upper* to indicate the index range in the list 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

BinarySearch(L,key,lower,upper):

1. Return None if the range is empty.
2. Set mid = the midpoint between lower and upper
3. Return mid if L[mid] is the key you're looking for.
4. If the key is less than L[mid],
   return BinarySearch(L,key,lower,mid)
   Otherwise, return BinarySearch(L,key,mid,upper).

How do we start the binary search?

return BinarySearch(L, key, -1, length of list L).

Example 1: Search for 73

lower = -1, upper = 15, mid = 7

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 = 7, upper = 15, mid = 11

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 = 7, upper = 11, mid = 9 (base case: key is found)

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

lower = -1, upper = 15, mid = 7

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 = -1, upper = 7, mid = 3

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 = 3, upper = 7, mid = 5

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 = 3, upper = 5, mid = 4

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 = 4, upper = 5 (base case: range is empty)

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 None`

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?
  - See how integer division works in this case.

Range is empty

- How do we determine if the range is empty?

```
lower + 1 == upper
```


Binary Search in Python: Recursively

```
def bs_helper(datalist, key, lower, upper):
    if lower + 1 == upper:    # range empty
        return None
    mid = (lower + upper)//2
    if key == datalist[mid]:  # found key
        return mid
    if key < datalist[mid]:
        return bs_helper(datalist, key, lower, mid)
    else:
        return bs_helper(datalist, key, mid, upper)

def bsearch(datalist, key):
    return bs_helper(datalist, key, -1, len(datalist))
```

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(datalist, 73,  -1,  15)
                        mid = 7 and 73 > datalist[7]
bs_helper(datalist, 73,   7,  15)
                        mid = 11 and 73 < datalist[11]
bs_helper(datalist, 73,   7,  11)
                        mid = 9 and 73 == datalist[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(datalist, 73,  -1,   15)
                        mid = 7 and 42 < datalist[7]
bs_helper(datalist, 73,  -1,    7)
                        mid = 3 and 42 > datalist[3]
bs_helper(datalist, 73,   3,    7)
                        mid = 5 and 42 < datalist[5]
bs_helper(datalist, 73,   3,    5)
                        mid = 4 and 42 > datalist[4]
bs_helper(datalist, 73,   4,    5)
                        lower+1 == upper
                        --->    return None
```

Analyzing Efficiency

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

In general...

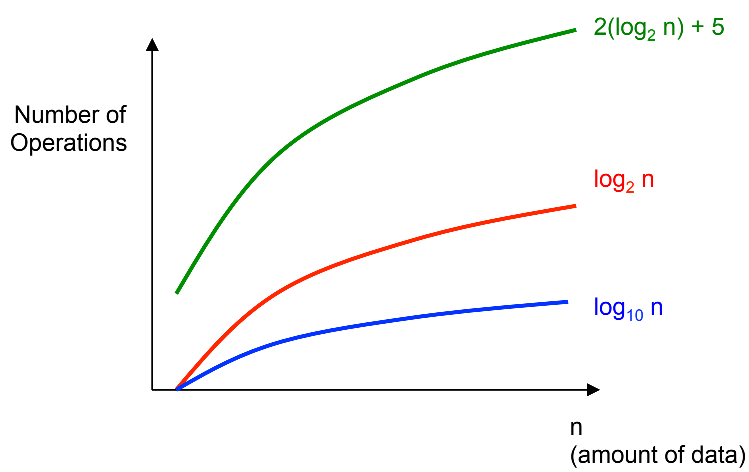
- In general, we can split search region in half $\lfloor \log_2 n \rfloor + 1$ times before it becomes empty.
- Recall the log function:
$\log_a b = c$ is equivalent to **$a^c = b$**
Examples:
$\log_2 128 = 7$
$\log_2 n = 5$ implies $n = 32$
- In our example: when there were 15 elements, we needed 4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

Big O

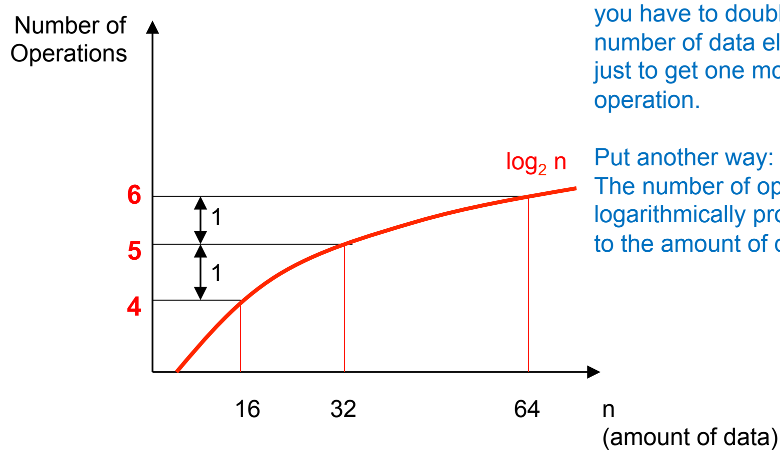
- In the worst case, binary search requires O(log n) time on a sorted list with n elements.
  – Note that in Big O notation, we do not usually specify the base of the logarithm. (It's usually 2.)
  – We only care that the relationship is logarithmic.
- <u>Number of operations</u> <u>Order of Complexity</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.

Put another way:
The number of operations is logarithmically proportional to the amount of data.

Searching (Worst Case)

Number of elements	Number of Comparisons	
	Linear Search	Binary Search
$15 \approx 2^4$	15	4
$31 \approx 2^5$	31	5
$63 \approx 2^6$	63	6
$127 \approx 2^7$	127	7
$255 \approx 2^8$	255	8
$511 \approx 2^9$	511	9
$1023 \approx 2^{10}$	1023	10
1 million$\approx 2^{20}$	1000000	20
1 billion $\approx 2^{30}$	1000000000	30

Binary Search Pays Off, but...

- Finding an element in a list with a billion elements requires only 30 comparisons!
- BUT....
  - The list must be sorted.
  - What if we sort the list 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...