School of
Information Systems

SMU
SINGAPORE MANAGEMENT UNIVERSITY

COR-IS1702:

COMPUTATIONAL THINKING

WEEK 4: ITERATION & DECOMPOSITION

2020/21 Term 1

(04) Iteration & Decomposition Part 1 (Searching)

Video (14 mins):

https://youtu.be/-IBippztq-Y?list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&t=178

Road Map

Algorithm Design and Analysis

- Week 1: Introduction, Counting, Programming
- Week 2: Programming
- Week 3: Complexity
- This week → Week 4: Iteration & Decomposition
- Week 5: Recursion

Fundamental Data Structures

(Weeks 6 - 10)

Computational Intractability and Heuristic Reasoning

(Weeks 11 - 13)

Learning Outcomes

- Understand the concept of iteration and decomposition
  - as exemplified by searching and sorting algorithms
- Analyze the best cases and the worst cases of an algorithm
- Able to compare the complexity of algorithms

References

- You will need the supporting Python file for the tutorial exercises.
- Download SearchSortLab.py from eLearn
- Import this each time you open a new terminal

```
For Search and Sort functions:
py (or python or python3 if it does not work)
>>> from SearchSortLab import *

For Animation:
py (or python or python3 if it does not work)
>>> from SearchSortAnimation import *
```

Searching and Sorting

- Two of the most important tasks that kept computers busy since its invention.
  - It's estimated that 25% of CPU time are spent on sorting in early years.

- Why is searching important?

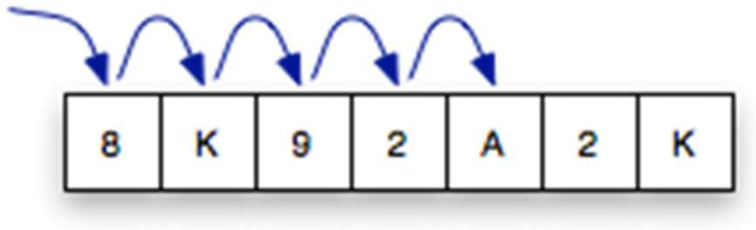
- Why is sorting important?
  - Once a set of items becomes sorted, many other problems become easy.

Defining Search

- "Search" can be very general
  - look for a book, either on a bookshelf at home or in a library
  - find a name in a phone book or a word in a dictionary
  - search a file drawer to find customer information or student records
  - Spotlight search in Mac OSX
- What these problems have in common:
  - a large collection of items
  - we need to search the collection to find a single item that matches a certain condition (e.g. name of book, name of a person)
- In this class, we will only focus on **exact matches**.
  - Search for an item that exactly matches given input.

Linear Search

- Linear search is the simplest, most straightforward search strategy
- As the name implies, the idea is to start at the beginning of a collection and compare items one after another



- Some terminology:
  - the item we are looking for is known as the ***key***
  - this type of search is also sometimes called a ***scan***
  - if the key is not found the search ***fails***

The `search` Function

- If `a` is an array object, we can call a function named **`lSearch`** to do a search and return the location of the item

```
>> a = ["apple", "lime", "kiwi", "orange", "ugli"]
=> ["apple", "lime", "kiwi", "orange", "ugli"]
>> lSearch(a, "kiwi")
=> 2
>> lSearch(a, "banana")
=> -1
```

Under the Hood

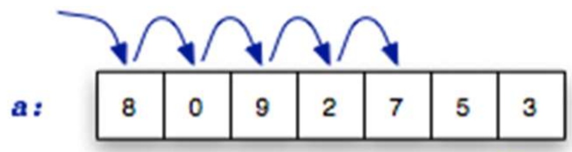
```
def lSearch(a, k):
    i = 0
    while i < len(a):
        if a[i] == k:
            return i
        i += 1
    return -1
```

`i` will (if necessary) go through every value from `0` to `len(a) - 1`

notice return values: either `i` or `-1` (not found)

Performance

- How many comparisons will the linear search algorithm make as it searches through an array with n items?
  - another way to phrase it: how many iterations will our Python function execute?
- For an unsuccessful search:
  - compare every item before returning `false` or `nil` or `-1`
  - *i.e.*, make n comparisons
- For a successful search, anywhere between 0 and n-1 (inclusive)
  - search may get lucky and find the item in the first location
  - similarly, it might be in the last location
  - expect, on average: (n + 1) / 2 comparisons

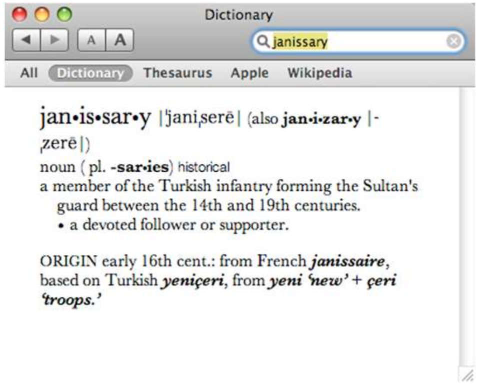


How Can We Improve the Performance?

- Observation:
  - After each comparison, we only manage to eliminate ONE candidate.

- Possible improvement:
  - Try to eliminate as many candidates as possible after each comparison.

Motivating Example – Dictionary Lookup



- A detailed specification of this process:

  Input: a word w and the dictionary

  Output: the meaning of w if found, and nil if not found

  1. the initial region is the entire dictionary
  2. at each step pick a word x in the middle of the current region
  3. there are now two smaller regions: the part <u>before x</u> and the part <u>after x</u>
  4. if w comes before x, repeat the search on the region before x, otherwise search the region following x (go back to step 3)

- This is called “Binary Search”:
  - We can eliminate half of all candidates after each comparison!
  - However, items must be arranged to certain order.

Binary Search

- The binary search algorithm uses the **divide-and-conquer** strategy to search through an array

- The array ***must be sorted***

  - The “zeroing in” strategy for looking up a word in the dictionary won’t work if the words are not in alphabetical order

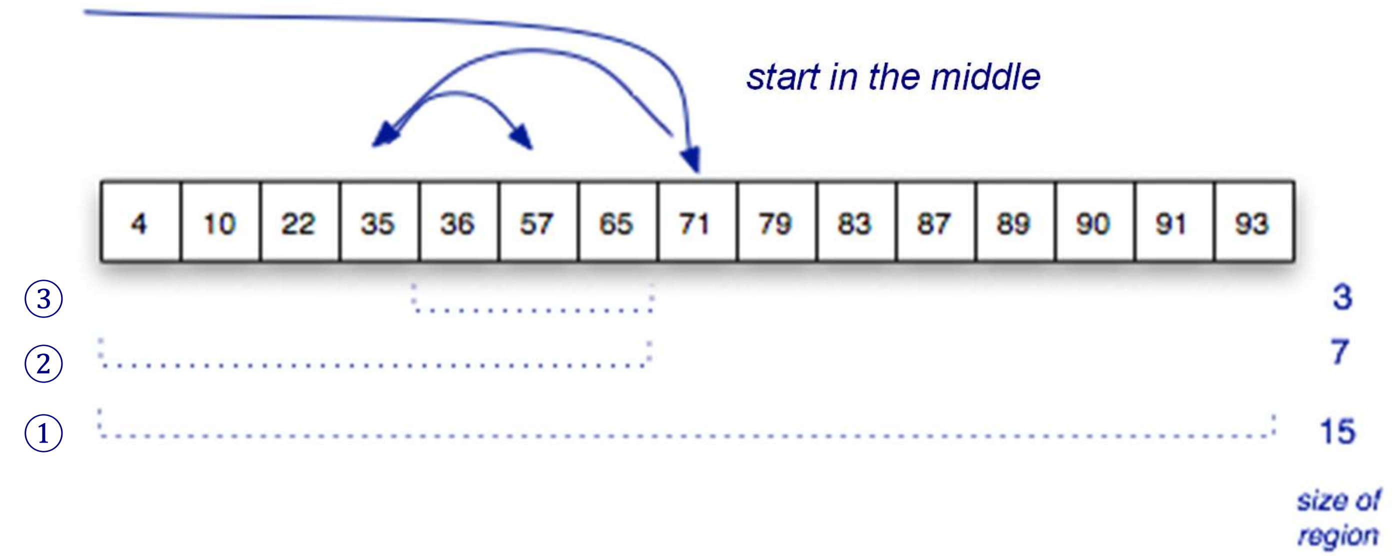
  - Binary search will not work unless the array is sorted

Binary Search

- To search a list of n items, first look at the item in middle location n/2
  - then search either:

    the region from 0 to n/2-1, or
    the region from n/2+1 to n-1

- Example: searching for 57 in a sorted list of 15 numbers



Animation:
http://video.franklin.edu/Franklin/Math/170/common/mod01/binarySearchAlg.html
http://www.youtube.com/watch?v=k-eNRYdkBa8

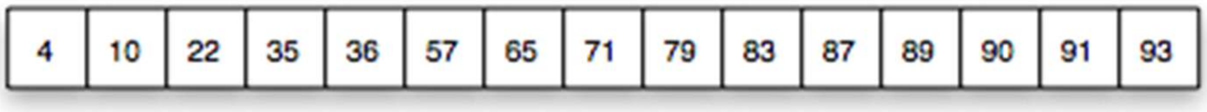
Detailed Description

- The algorithm uses two variables to keep track of the boundaries of the region to search

`lower`	index value ***one below*** the leftmost item in the region
`upper`	index value ***one above*** the rightmost region

-1 n

[0 n-1]

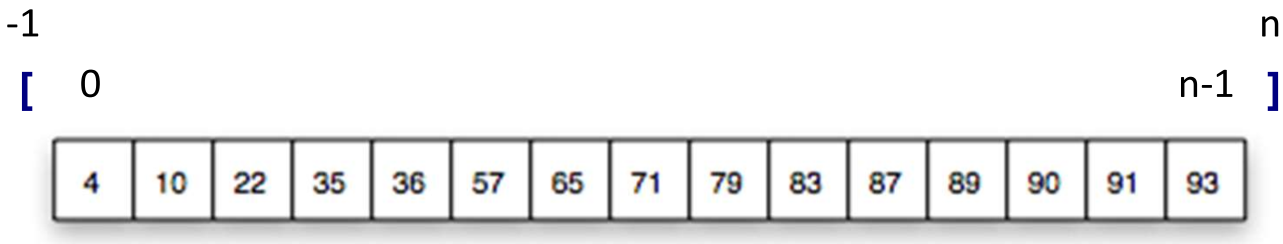


initial values when searching an array of n items:

```
lower = -1
upper = n
```

Detailed Description

- Perform iteration that keeps making the region smaller and smaller
  - the initial region is the complete array
  - the next one is either the upper half or lower half
  - the one after that is one quarter, then one eighth, then...



initial values when searching an array of n items:

```
lower = -1
upper = n
```

Detailed Description

- The first iteration when searching for 57 in a list of size 15:

```
[                                    *                                    ]
```

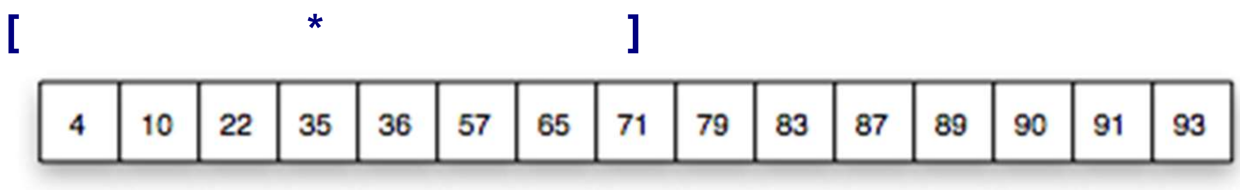
4	10	22	35	36	57	65	71	79	83	87	89	90	91	93

```
lower = -1        mid = 14 / 2 = 7        upper for next
upper = 15                                iteration: 7
```

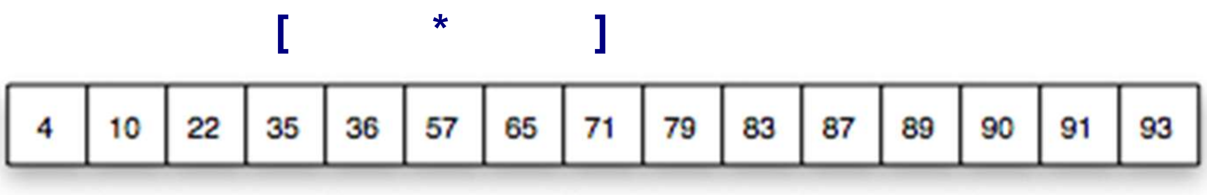
Detailed Description

- The remaining iterations when searching for 57:

```
lower = -1
upper = 7
mid = 3
lower = 3
```




```
lower = 3
upper = 7
mid = 5
found it!
```



This search required only 3 comparisons:

`a[7], a[3], a[5]`

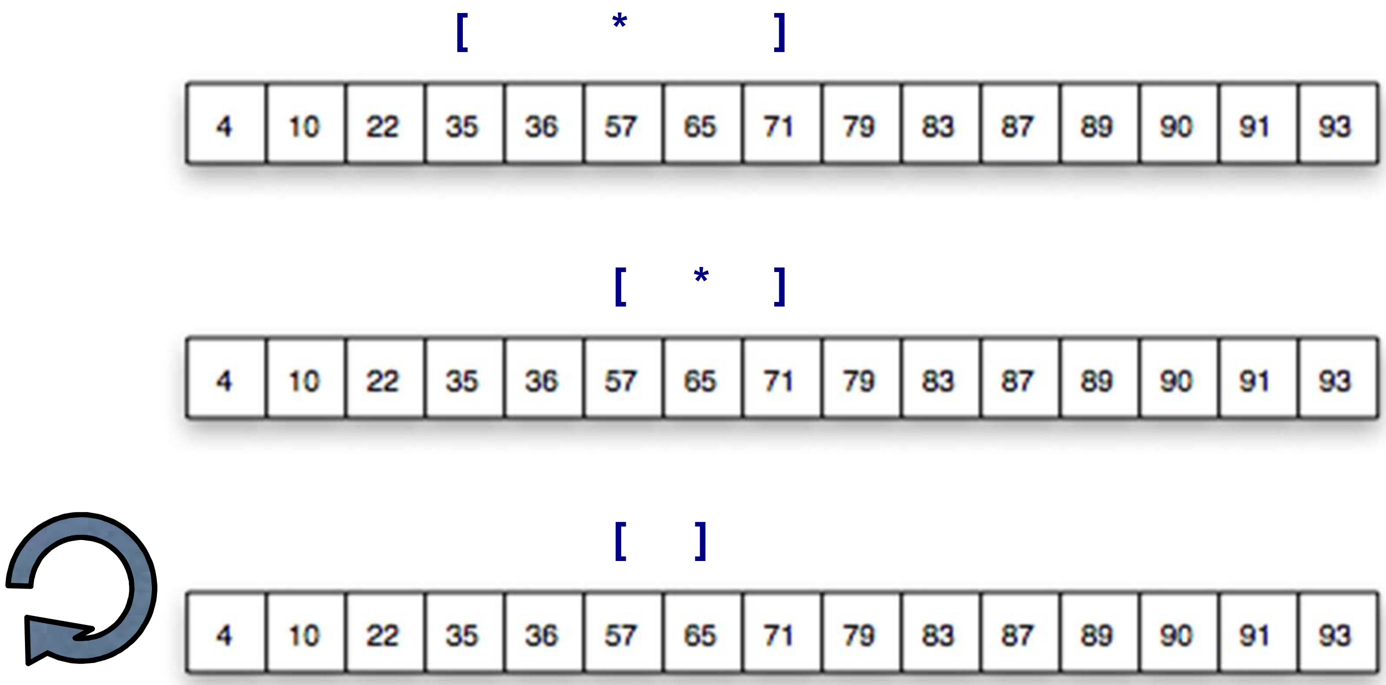
Unsuccessful Searches

- What happens in this algorithm if the item we're looking for is not in the array?
- Example: search for 58

```
lower = 3
upper = 7
mid = 5
lower = 5

lower = 5
upper = 7
mid = 6
upper = 6

lower = 5
upper = 6
mid = 5
lower = 5
```



Unsuccessful Searches

- To fix this problem we have to add another condition to the loop
  - we want the result to be `nil` if the region shrinks to 0 items
  - this happens when `upper` equals `lower + 1`

```
mid = (lower + upper) // 2
if (lower + 1 == upper):
  return -1
if k == a[mid]:
  return mid
if k < a[mid]:
  upper = mid
else:
  lower = mid
```

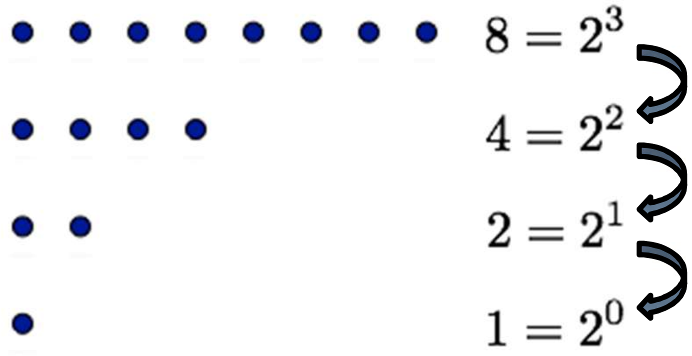
Binary Search Function in Python

```
def bSearch(array, target):
    lower = -1
    upper = len(array)
    while not (lower + 1 == upper):      #fail if region empty
        mid = (lower + upper)//2         #find middle of region
        if target == array[mid]:         #succeed if k is at mid
            return mid
        elif target < array[mid]:
            upper = mid                  #search lower region
        else:
            lower = mid                  #search upper region
    return -1                            #not found
```

Complexity of Binary Search

- When we're searching we're reducing an area of size n down to an area of size 1.
  - e.g. n = 8 in this diagram.
  - Let m be the number of steps required:

$$\frac{n}{2^m} = 1 \;\rightarrow\; n = 2^m \;\rightarrow\; m = \log_2 n$$



- Best case: a successful search might return after the first comparison.

- Worst case: reduce original area of size n to an area of size 1, and perform the final comparison to conclude search failure:

steps = $(\log_2 n + 1)$

Big O Complexity of Binary Search

- In the worst case, $(\log_2 n + 1)$ steps, so we have $O(\log_2 n)$
- Does the base of the log matter in Big O notation?
- Change of base:
  - $\log_2 n = (\log_k n) / (\log_k 2)$, for any constant k
- Since $\log_k 2$ is a constant, we have:
  - $O(\log_2 n) = O(\log_k n / \log_k 2) = O(\log_k n)$

- The base of the log does not matter in terms of Big O notation
- The base of the log matters if we want the exact number of steps

Recap: Search

- The linear search algorithm looks for an item in an array
  - starts at the beginning (`a[0]`, or “the left”)
  - compares each item, moving systematically to the right (`i += 1`)
  - Best Case: 1 comparison when the item is in the first position
  - Worst Case: do n comparisons when the search is unsuccessful
  - Complexity: $O(n)$

- The binary search algorithm uses divide and conquer
  - array must be sorted beforehand
  - starts at the middle, systematically halves the search space
  - Best Case: 1 comparison when the item is in the mid position
  - Worst Case: $(\log n) + 1$ comparisons, when the item is not found
  - Complexity: $O(\log n)$

In-Class Exercises

The following sub-questions concern the array below:

```
x = [0, 2, 5, 7, 29, 36, 46, 55, 62, 67,
72, 73, 81, 88]
```

- ❖ How many elements will be checked by linear search and by binary search respectively to find the number 7?
- ❖ How many elements will be checked by linear search and by binary search respectively to find the number 50?