UNIVERSITY OF LEICESTER

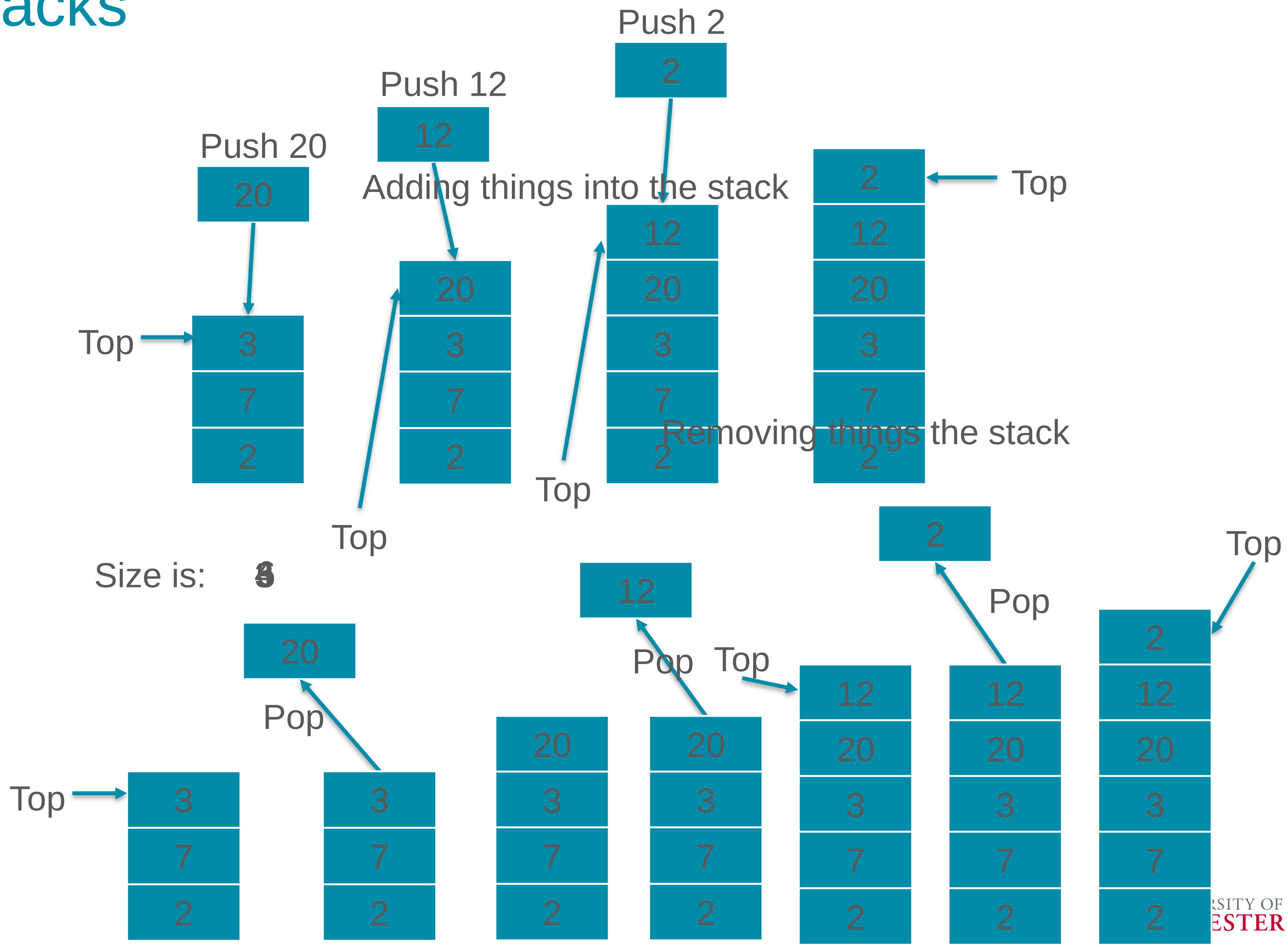
CO1107

Data Structure

Stacks

- A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**.
- This means the last element inserted inside the stack is removed first.
- A stack is like a list except…
  - You can **only** interact with the item at the 'top'
  - You can 'push' something to put it on the top of the stack
  - You can 'pop' to take something from the top of the stack
  - You can also check its size

Stacks



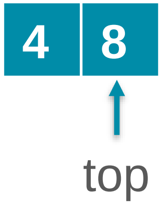
Adding things into the stack

Removing things the stack

Size is: [illegible]

Check your understanding

Assuming a stack started off with the following items in it



4 8

top

What would be at the top of the stack after the following operations?

push 10, push 2, push 5, pop pop

A. 4
B. 8
C. 10
D. 2
E. 5

Check your understanding (2)

Assuming a stack started off with the following items in it

What would be at the top of the stack after the following operations?

pop, pop, pop, pop

A. 4
B. 8
C. 5
D. 3
E. None of the above

Implementing stacks…

Which of the following statements are true

Given the list: L = [1,65,3,6]

I) L.append(4) gives [1,65,3,6,4]

II) L.append(4) gives [4,1,65,3,6]

III) L.pop() leaves L as [1,65,3,6]

IV) L.pop() leaves L as [65,3,6]

A. I and III are true
B. I and IV are true
C. II and III are true
D. II and IV are true

Stacks: A simple implementation

One way to implement is as a python list

Python lists have the following operations:

- Append – this will be our push operation
- Pop – this will be our pop operation
- Len – this will be our size operation

Stacks: Common uses

Stacks are often used to:

- Reverse a list of items
- In browsers

Reversing a list

```
def reverseListWithStack(aList):
    stack = []
    for item in aList:#fill the stack
        stack.append(item) #push item
    #stack can now reverse them
    position = 0
    while len(stack)>0:
        aList[position] = stack.pop()
#overwrite item at
#this position with what was popped off the
stack
        position += 1
    return aList
```

Queues

- Unlike stacks, a queue is open at both its ends. One end is always used to **insert data (enqueue)** and the other is used to **remove data (dequeue).**
- A Queue is like a list except…
  - You can **only** interact with the front and rear
  - You can ‘append’ something to put it at the end of the queue. (enqueue)
  - You can ‘serve’ to take from the front of the queue. (**dequeue)**
  - You can also check its size
- This kind of behaviour is commonly referred to as **FIFO (First in First Out)**

Checking understanding

Assuming a queue with the following contents

front 3 7 5 2 0 rear

what is the size of this queue after the following operations:

Serve, serve, append 20, serve, append 4

A. 2
B. 3
C. 4
D. 5
E. 6
F. 7

Queues: A simple implementation

One way to implement is as a python list

Python lists have the following operations:

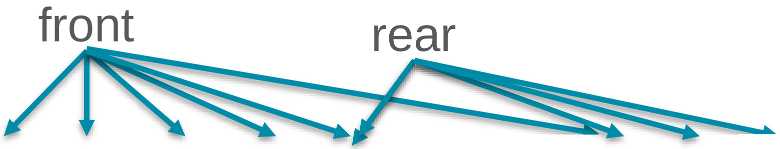
- Append – our append operation
- Pop(0) – this will be our serve operation
- Len – this will be our size operation

Queue: Common uses

- Anything that needs a buffer, where data MUST be processed in the same order it is received in:
  - Print queues
  - Streaming videos
  - CPU Scheduling
  - Call centre phone system

Removing negative numbers from a queue

```
"""
This function will take a queue of numbers and remove the items < 0
"""
def removeNegatives(aQueue):
    numberSeen = 0
    n = len(aQueue)
    while numberSeen <= n:
        item = aQueue.pop(0)
        numberSeen+=1
        if item >=0:
            #only positive numbers are put back in
            aQueue.append(item)
```



Brute Force Algorithm

Definition

- Brute Force algorithm is a typical problem solving technique go through each possibility to find the solutions.
- Example: Searching a word in a dictionary:
  - Look at each and every word in a dictionary
  - If it finds the match, returns its definition, etc
- Generate all candidate solutions
- Check if each candidate is a possible solution
- Find a (best) solution or all solutions.
- Can you think of any example that used Brute Force algorithm ?

Knapsack

Suppose you are in a treasure cave which contains 6 precious items, with the following weights and monetary value.

Item	**1**	**2**	**3**	**4**	**5**	**6**
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

You want to take as much treasure as you can carry.

However, you can only carry up to 20kg.

Which items do you take?

Solving Knapsack

Generate a list of all subsets of items

↓

Eliminate those subsets whose total weight is too much

↓

Calculate the total value for each of the sets of items

↓

Find the set of items with the greatest total value

List of set of items with total weight <= 20kg

0. Item 1 (20kg)
1. Item 2 (10kg)
2. Item 2 (10kg) & Item 3 (9kg)
3. Item 2 (10kg) & Item 3 (9kg) & Item 6 (1kg)
4. Item 2 (10kg) & Item 4 (4kg)
5. Item 2 (10kg) & Item 4 (4kg) & Item 5 (2kg)
6. Item 2 (10kg) & Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg)
7. Item 2 (10kg) & Item 4 (4kg) & Item 6 (1kg)
8. Item 2 (10kg) & Item 5 (2kg)
9. Item 2 (10kg) & Item 5 (2kg) & Item 6 (1kg)
10. Item 2 (10kg) & Item 6 (1kg)
11. Item 3 (9kg)
12. Item 3 (9kg) & Item 4 (4kg)
13. Item 3 (9kg) & Item 4 (4kg) & Item 5 (2kg)
14. Item 3 (9kg) & Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg)
15. Item 3 (9kg) & Item 4 (4kg) & Item 6 (1kg)
16. Item 3 (9kg) & Item 5 (2kg)
17. Item 3 (9kg) & Item 5 (2kg) & Item 6 (1kg)
18. Item 3 (9kg) & Item 6 (1kg)
19. Item 4 (4kg)
20. Item 4 (4kg) & Item 5 (2kg)
21. Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg)
22. Item 4 (4kg) & Item 6 (1kg)
23. Item 5 (2kg)
24. Item 5 (2kg) & Item 6 (1kg)
25. Item 6 (1kg)

List of possible set of items and values

Best Solution

0. Item 1 (20kg) **$4000**
1. Item 2 (10kg) **$3500**
2. Item 2 (10kg) & Item 3 (9kg) **$5300**
3. *Item 2 (10kg) & Item 3 (9kg) & Item 6 (1kg)* ***$5500***
4. Item 2 (10kg) & Item 4 (4kg) **$3900**
5. Item 2 (10kg) & Item 4 (4kg) & Item 5 (2kg) **$4900**
6. Item 2 (10kg) & Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg) **$5100**
7. Item 2 (10kg) & Item 4 (4kg) & Item 6 (1kg) **$4100**
8. Item 2 (10kg) & Item 5 (2kg) **$4500**
9. Item 2 (10kg) & Item 5 (2kg) & Item 6 (1kg) **$4700**
10. Item 2 (10kg) & Item 6 (1kg) **$3700**
11. Item 3 (9kg) **$1800**
12. Item 3 (9kg) & Item 4 (4kg) **$2200**
13. Item 3 (9kg) & Item 4 (4kg) & Item 5 (2kg) **$3200**
14. Item 3 (9kg) & Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg) **$3400**
15. Item 3 (9kg) & Item 4 (4kg) & Item 6 (1kg) **$2400**
16. Item 3 (9kg) & Item 5 (2kg) **$2800**
17. Item 3 (9kg) & Item 5 (2kg) & Item 6 (1kg) **$3000**
18. Item 3 (9kg) & Item 6 (1kg) **$2000**
19. Item 4 (4kg) **$400**
20. Item 4 (4kg) & Item 5 (2kg) **$1400**
21. Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg) **$1600**
22. Item 4 (4kg) & Item 6 (1kg) **$600**
23. Item 5 (2kg) **$1000**
24. Item 5 (2kg) & Item 6 (1kg) $**1200**
25. Item 6 (1kg) **$200**

Representation of items using bit-lists

Items from 1 to 6, indices beginning from 1

0. Item 1 (20kg) **100000**
1. Item 2 (10kg) **010000**
2. Item 2 (10kg) & Item 3 (9kg) **011000**
3. Item 2 (10kg) & Item 3 (9kg) & Item 6 (1kg) **011001**
4. Item 2 (10kg) & Item 4 (4kg) **010100**
5. Item 2 (10kg) & Item 4 (4kg) & Item 5 (2kg) **010110**
6. Item 2 (10kg) & Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg) **010111**
7. Item 2 (10kg) & Item 4 (4kg) & Item 6 (1kg) **010101**
8. Item 2 (10kg) & Item 5 (2kg) **010010**
9. Item 2 (10kg) & Item 5 (2kg) & Item 6 (1kg) **010011**
10. Item 2 (10kg) & Item 6 (1kg) **010001**
11. Item 3 (9kg) **001000**
12. Item 3 (9kg) & Item 4 (4kg) **001100**
13. Item 3 (9kg) & Item 4 (4kg) & Item 5 (2kg) **001110**
14. Item 3 (9kg) & Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg) **001111**
15. Item 3 (9kg) & Item 4 (4kg) & Item 6 (1kg) **001101**
16. Item 3 (9kg) & Item 5 (2kg) **001010**
17. Item 3 (9kg) & Item 5 (2kg) & Item 6 (1kg) **001011**
18. Item 3 (9kg) & Item 6 (1kg) **001001**
19. Item 4 (4kg) **000100**
20. Item 4 (4kg) & Item 5 (2kg) **000110**
21. Item 4 (4kg) & Item 5 (2kg) & Item 6 (1kg) **000111**
22. Item 4 (4kg) & Item 6 (1kg) **000101**
23. Item 5 (2kg) **000010**
24. Item 5 (2kg) & Item 6 (1kg) **000011**
25. Item 6 (1kg) **000001**

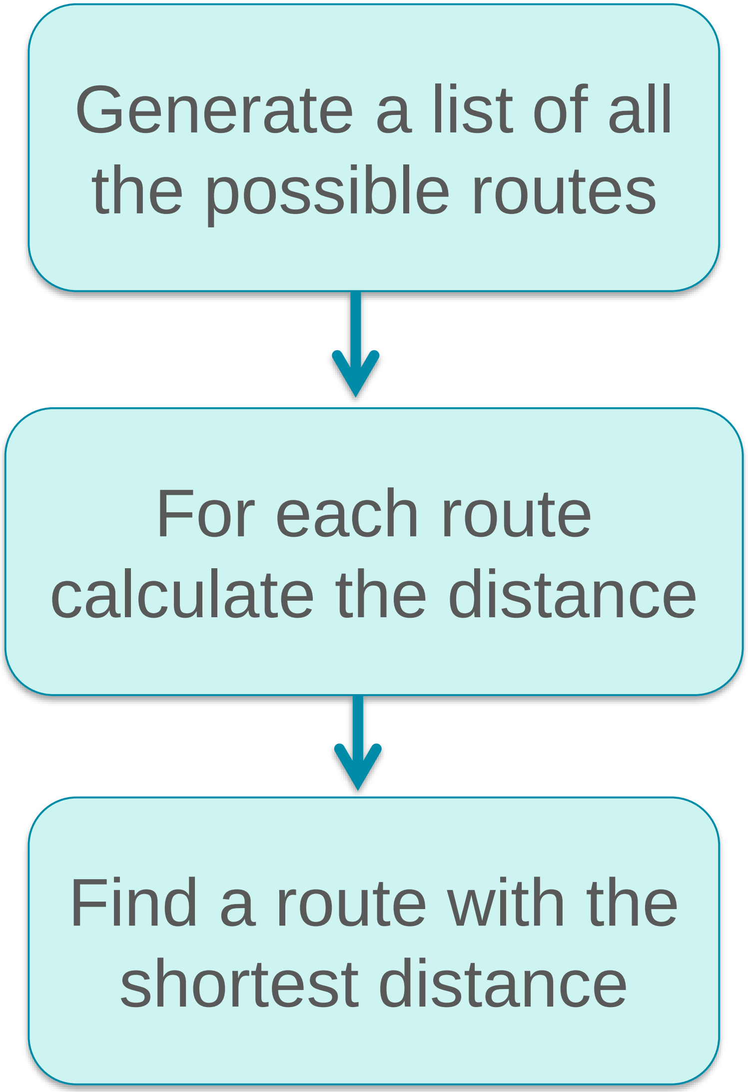
Travelling Salesman Problem

- Suppose you are given the following driving distances in miles between the following cities:

	London	Leicester	Manchester	Birmingham	Coventry
London		100	192	120	95
Leicester	100		95	42	24
Manchester	192	95		78	98
Birmingham	120	42	78		22
Coventry	95	24	98	22	

- Find the shortest route that enables a salesman to start at London, visit all the other cities, before returning to London.

Solving Travelling Salesman



Generate a list of all the possible routes

For each route calculate the distance

Find a route with the shortest distance

Difficulties with Brute Force

- Need to list all the possible solutions.
- The number of possible solutions increases for some problems very quickly as the size of the problem increases.

Search Algorithm:

Sequential Search

Finding a phone number

Consider the problem of trying to find a telephone number in a phone book.

Reza 0463935372

Alex 0411484152

Emmanuel 0418721183

Tom 0436242684

Karim 0479753034

Roy 0445778949

Michael 0436756947

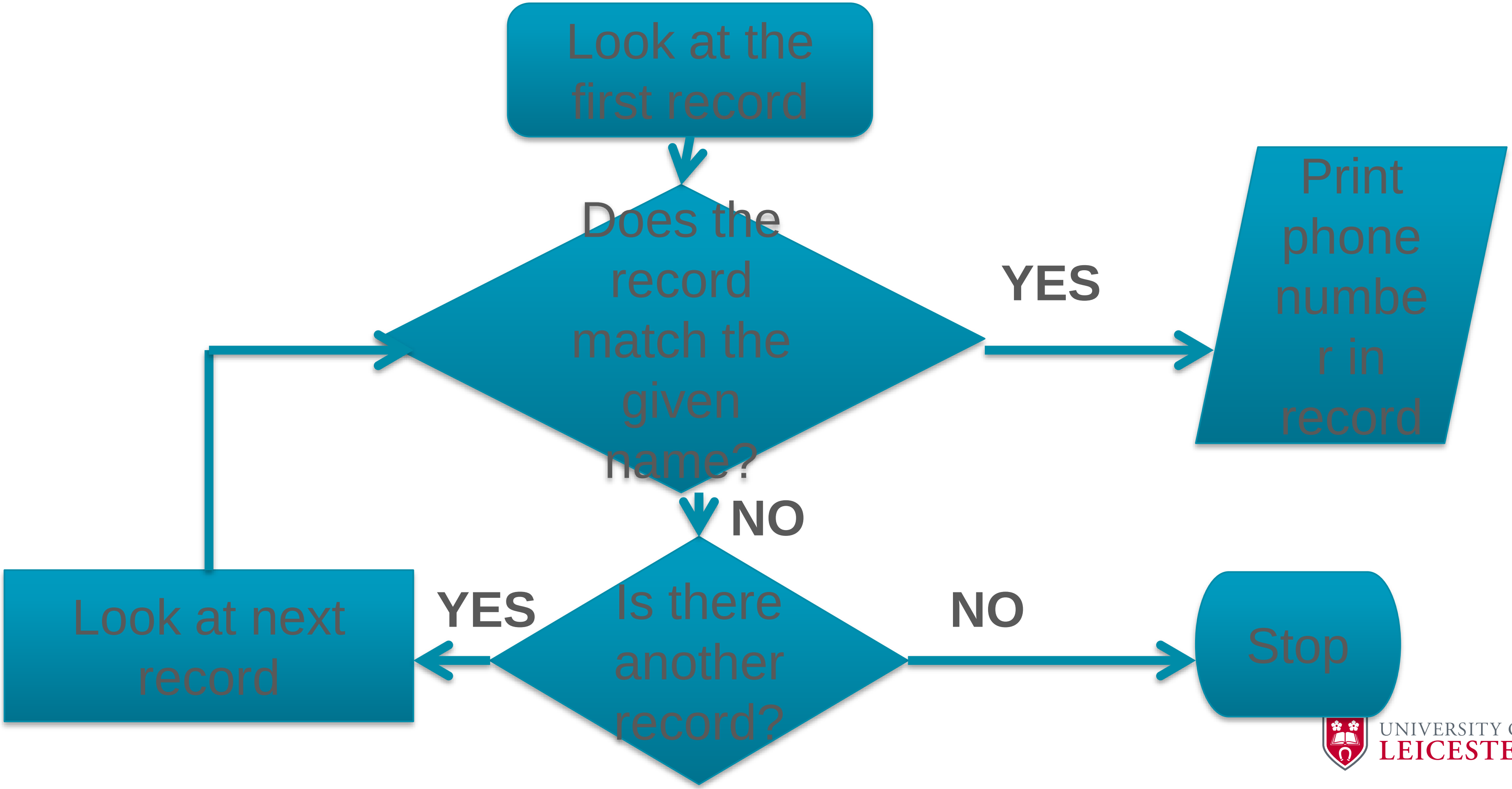
Richard 0483503919

…

Approach

// Input: A name

// Output: The number corresponding to the phone number.



Look at the first record

Does the record match the given name?

YES

Print phone number in record

NO

Is there another record?

YES

Look at next record

NO

Stop

Divide and Conquer

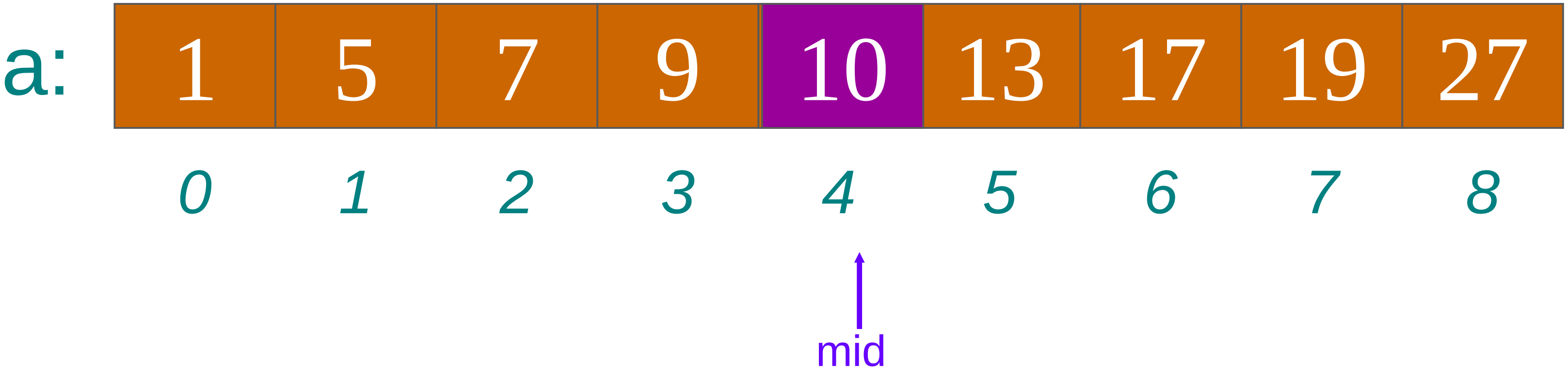
- Divide instance of a problem into 2 or more smaller instances
- Conquer (solve) smaller instances and combine solutions to obtain solution to bigger instances

Binary Search

```
If ( target == middle item )
    target is found
else if ( target < middle item )
    search left-half of list with the same method
else
    search right-half of list with the same method
```

Binary Search Case 1:

target = 10 (middle item)

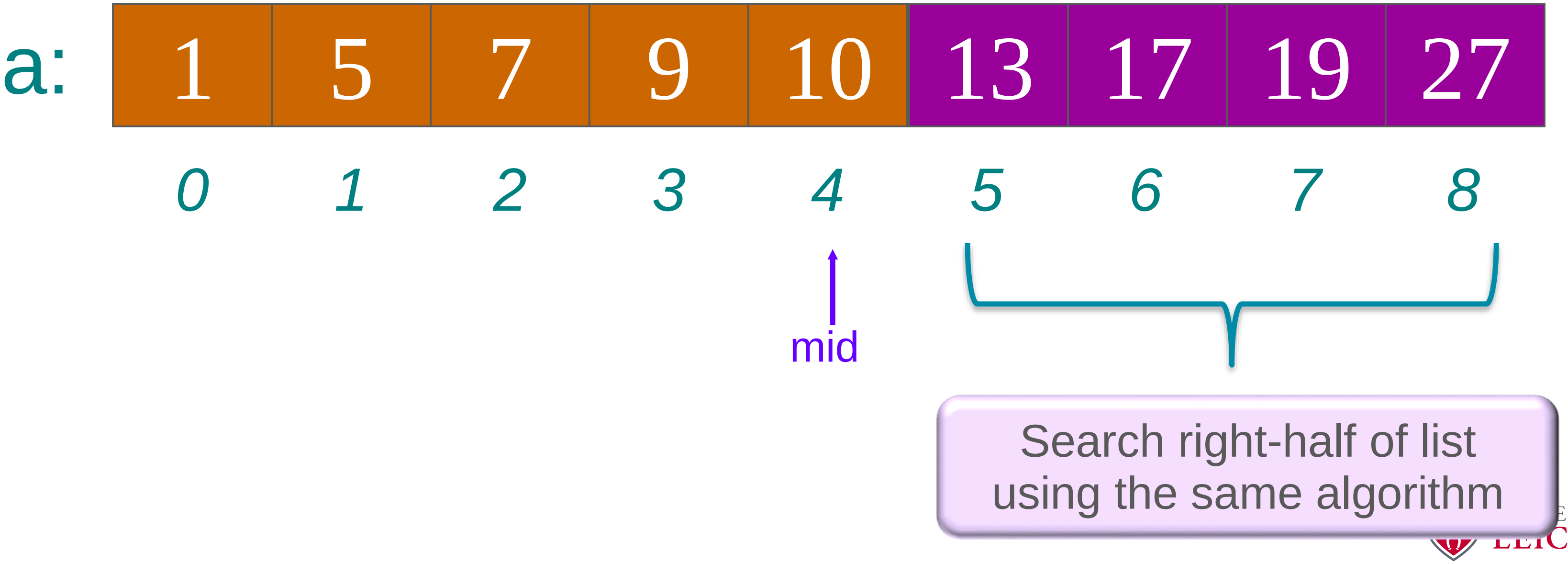


a:	1	5	7	9	10	13	17	19	27
	0	1	2	3	4	5	6	7	8

Return `mid`

Binary Search Case 2:

target = 19

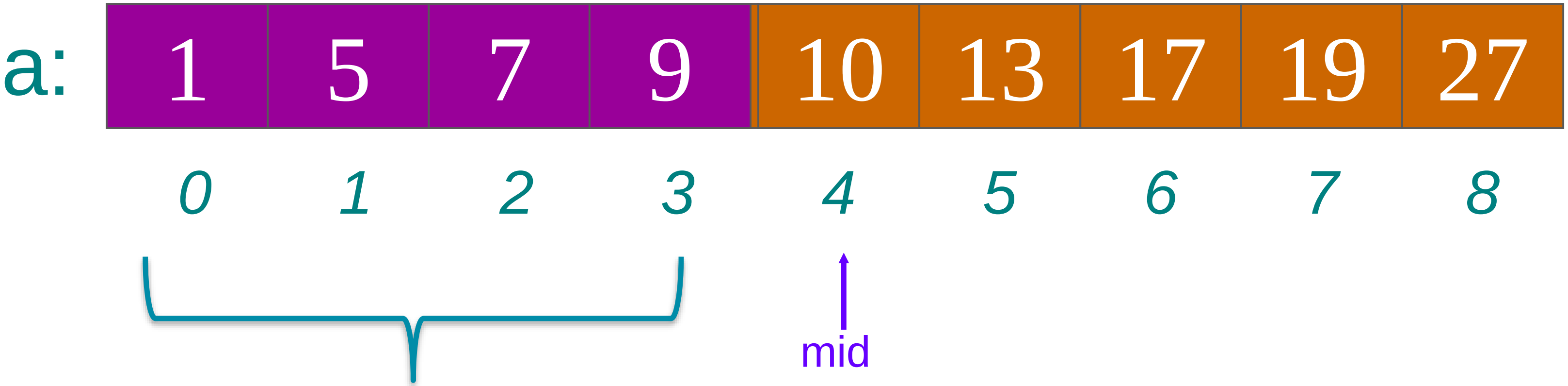


a:	1	5	7	9	10	13	17	19	27
	0	1	2	3	4	5	6	7	8

Search right-half of list using the same algorithm

Binary Search Case 3:

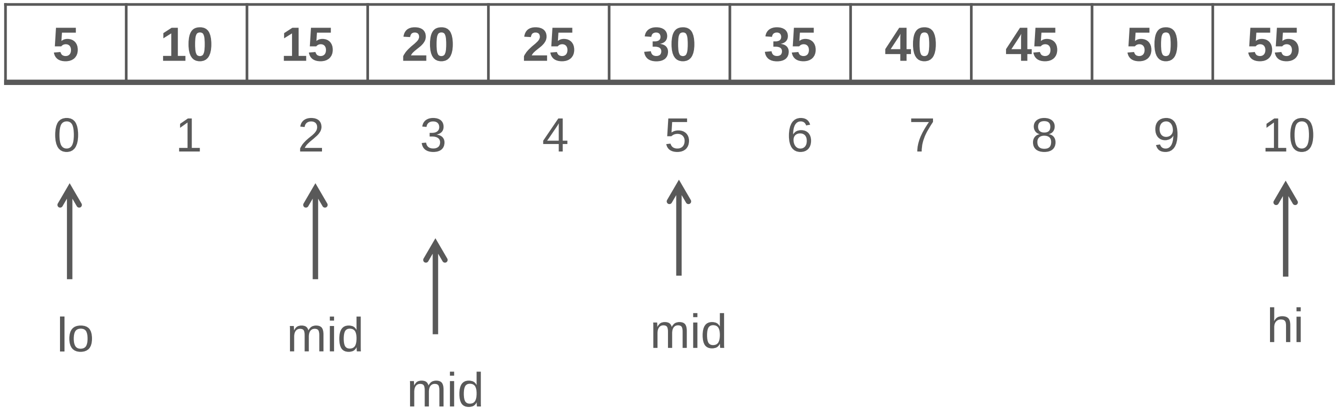
target = 7



a:	1	5	7	9	10	13	17	19	27
	0	1	2	3	4	5	6	7	8

Search left-half of the list using the same algorithm

Binary Search



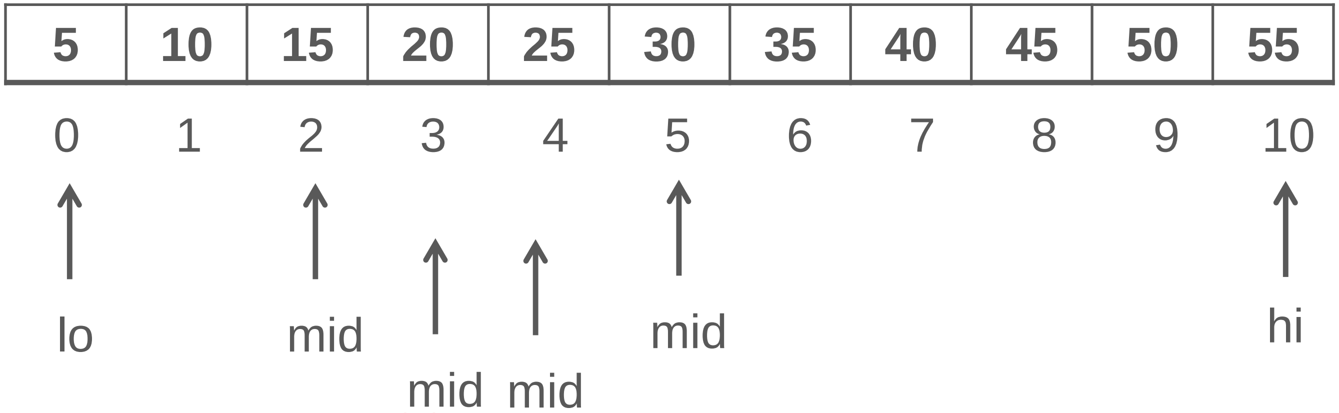
5	10	15	20	25	30	35	40	45	50	55
0	1	2	3	4	5	6	7	8	9	10

Searching **target** in range (lo to hi)

Assume **target** = 20

- Repeat until range is not empty (i.e., while lo <= hi)
  - mid = (lo+hi)//2
  - if **target** < array[mid],
    - Search from lo to mid-1 (e.g., move hi to mid-1)
  - if **target** > array[mid]
    - Search from mid+1 to hi (e.g., move lo to mid+1)
  - if **target** == array[mid],
    - Return mid

What will be lo and hi when while loop quits?



5	10	15	20	25	30	35	40	45	50	55
0	1	2	3	4	5	6	7	8	9	10

Assume **target** = 23

- Repeat until range is not empty (i.e., while lo <= hi)
    - mid = (lo+hi)//2
    - if **target** < array[mid],
        - Search from lo to mid-1 (e.g., move hi to mid-1)
    - if **target** > array[mid]
        - Search from mid+1 to hi (e.g., move lo to mid+1)
    - if **target** == array[mid],
        - return mid
- return -1

A. lo = 4, hi = 4
B. lo = 4, hi = 3
C. lo = 3, hi = 3
D. None of the abov

Binary Search

```
# returns the index of target if target is found, otherwise returns -1
def binarySearch(aList, target):
      low = 0
      high = len(aList)-1
          # continue until range is not empty
      while low <= high:
            mid = (low + high) // 2
            # return the index if target is found
            if aList[mid] == target:
                 return mid
            # otherwise update the range
            elif aList[mid] > target:
                 high = mid-1
            else:
                 low = mid+1
      # if range is empty and target is not found, return -1
      return -1
aList = [5,10,15,20,25,30,35,40,45,50,55]
print(binarySearch(aList,20))
```