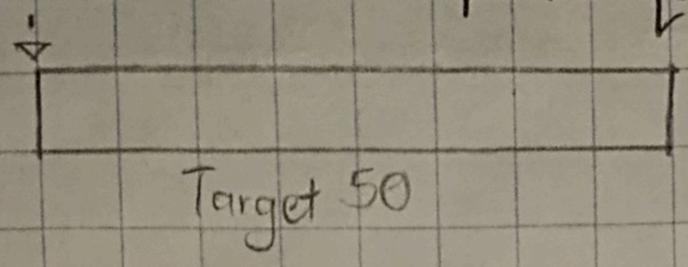
ALGORITHMS.

Algorithms - a set of steps a program takes to complete a task.
- An established body of knowledge on how to solve particular problems well.

Search Algorithms.

- Linear search / Simple / Sequential → Searching for a target value sequentially until u find it.

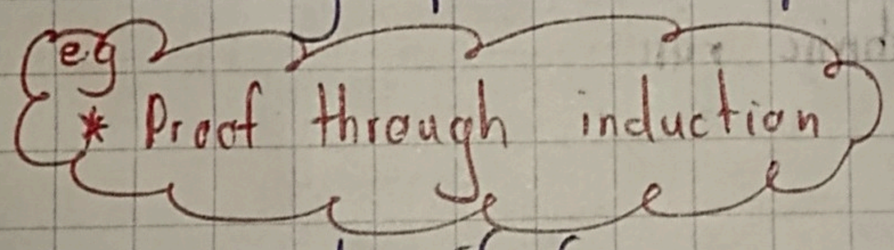


Input - list of values.
Output - target values.

Your algorithm:

Should be clear and concise.
1. Have a clearly defined problem statement (input & output)
2. Steps should be in a very specific order.
3. Should produce a result.
4. Should be complete and not take an infinite amount of time / complete in a finite amount of time.

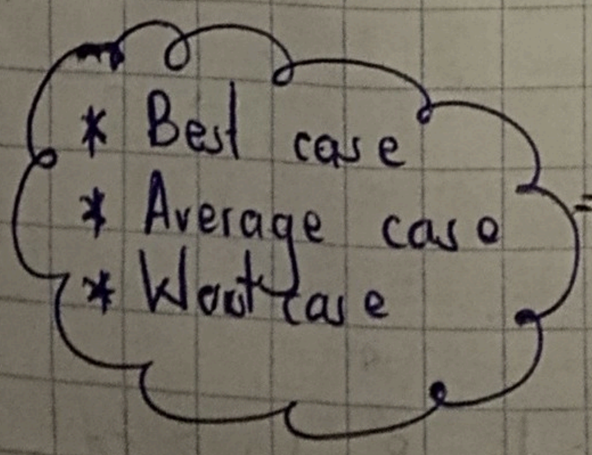
Should be correct and efficient
Correct → for any possible input, the algorithm should always terminate



An alg is correct if for every run of the algorithm against all possible values in the input data, we always get the output we expect

Efficiency → Should help us solve problems faster and delive a better end user experience.
Measures of efficiency → Time & Space.

Time complexity → time used to complete a task.
Space complexity → amount of memory used in the computer.



→ analyzing the efficiency of an algorithm.

Binary search.

- Works by eliminating the range of values by half until the target value is found.

* Input - a ~~list~~ sorted list of values.
* Output - ē position in the list of ē target values we are searching for
  Values need to be sorted.

Efficiency of Binary Search. → faster.

Big O → notation 4 describing complexity.

* Theoretical definition of the complexity of an algorithm as a function of ē size
- Measures complexity as the input size grows.

Big O → (upper bound of ē alg) → measures worst case scenario of an algorithm

Linear search → O(n) ~~→ can~~ → linear time
Binary search → O(log n) → logarithmic run

Linear search.

Its runtime is constant time → O(1)

Binary search.

Its runtime is logarithmic runtime O(log n)

e.g:

$2^{\downarrow 3} = 8$

exponent → defines how the no. grows.

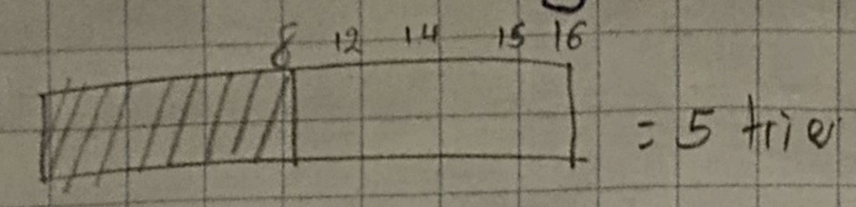
Logarithm → opposite of exponent.

$\log_2 8 = 3$ → How many times do i have to divide 8 by 2 to get 1

for Binary Search (BS), to find the no. of tries, we use:

$\log_2 n + 1$

e.g if 16 is our target.



In short, $\log_2 16 + 1 = 5$

We rep this runtime in Big O as $O(\log n)$ or $O(\ln n)$

logarithmic runtime.

Common Complexities

Alg Runtimes and Complexities

Algorithms with logarithmic runtime are preferred to linear

Also called sublinear.

Linear time.

Algorithms run in linear time when the worst case scenario is equal to the number of tries.

An algorithm that sequentially reads n input will have linear time
Big O of n = O(n)

Quadratic time.

Quadratic → when something is squared.

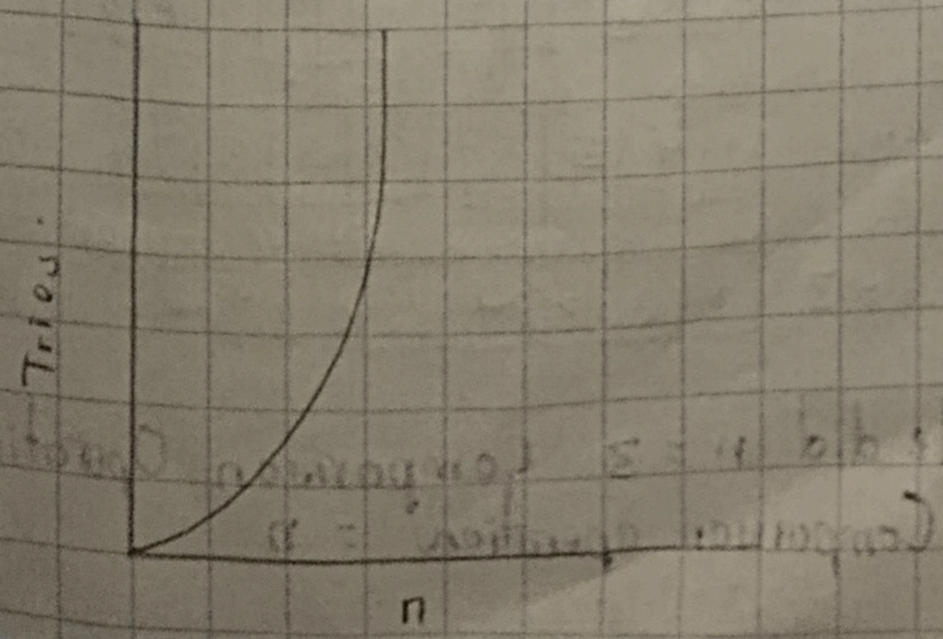
	4,1	4,2	4,3	4,4
n = 4	3,1	3,2	3,3	3,4
$n^2 = 16$	2,1	2,2	2,3	2,4
	1,1	1,2	1,3	1,4

Any number of tries is squared.

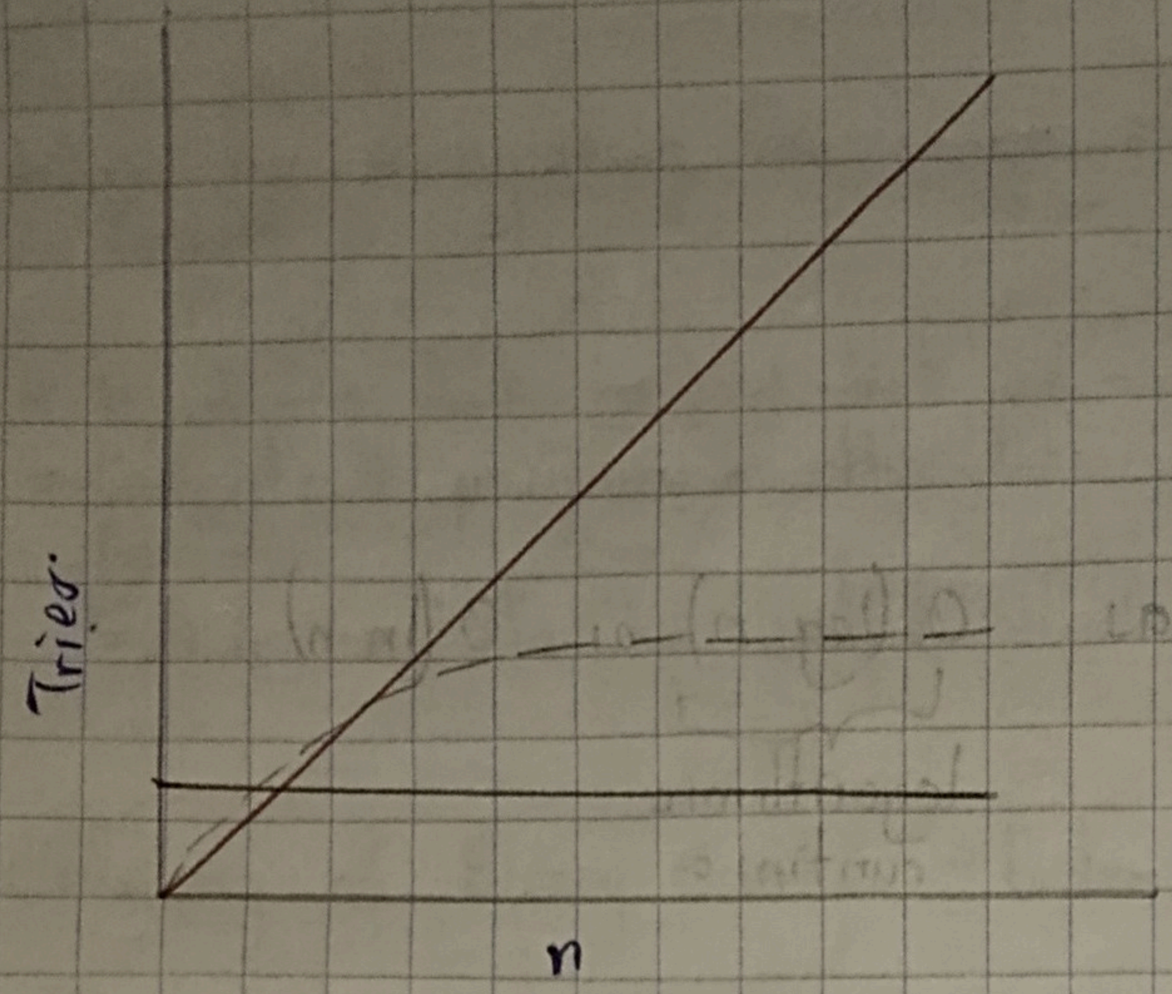
for any given value of n, we carry out n^2 no. of operations.

$= O(n^2)$ → Quadratic runtime.

Cubic Runtime → n^3



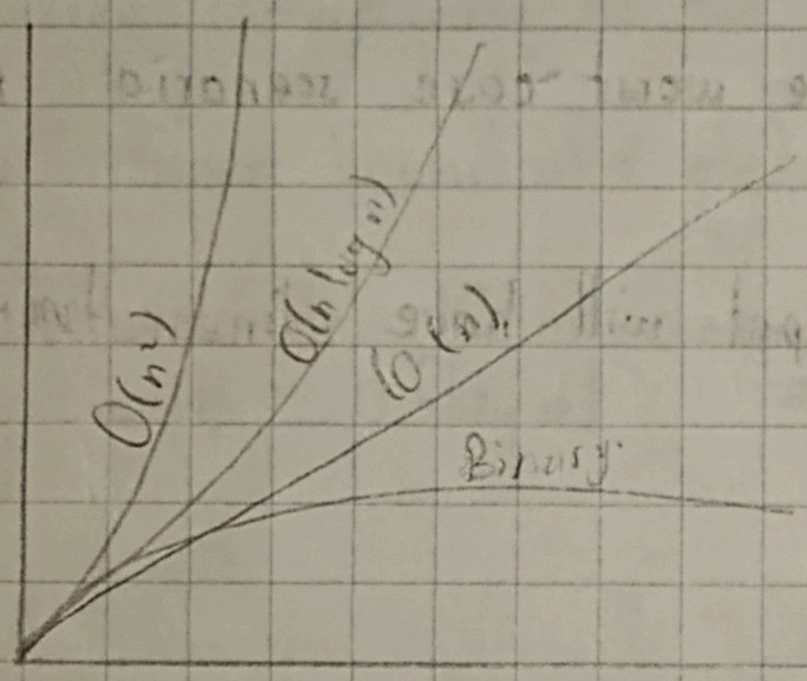
Graph of cubic and quadratic runtime



— logarithmic time.
— linear time
— constant time

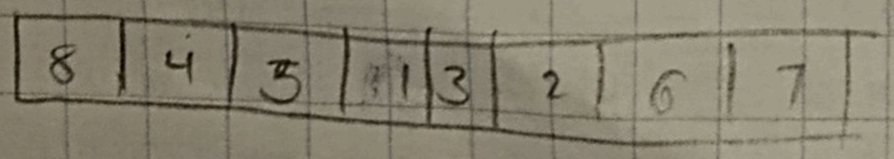
Quasilinear Runtimes → $O(n \log n)$

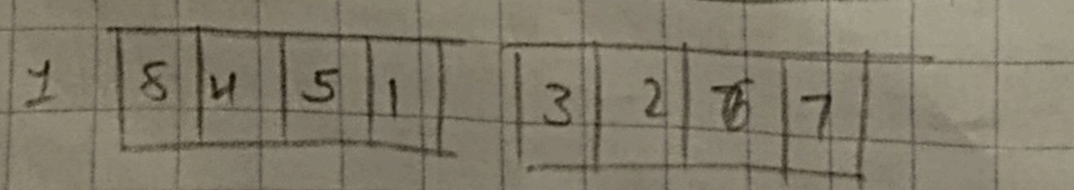
for every value of n, we execute a log n number of operations
Lies between $O(n^2)$ and $O(n)$



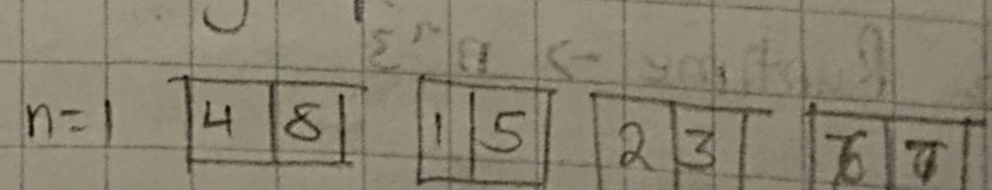
e.g Merge Sort.

Worst case → $O(n \log n)$
Sorts by cutting down in halves
- Splitting operation
- Merge operation
- Comparison operation.





Merge operations = n



We did n = 3 Comparison Operations
Comparison operation = n

Polynomial Runtime → Quadratic
Cubic

An algorithm is considered a polynomial Runtime if for a given value of n, its worst case runtime is n raised to the k power where k = any value.

e.g $O(n^k)$ where $k = 2$ or $k = 3$

~~Are~~ Algorithms with this are considered efficient.

Exponential Runtimes

Is an algorithm with a Big O ~~notation~~ number of some value raised to the nth power.

as n increases slightly, ē number of tries increase exponentially.

Examples:

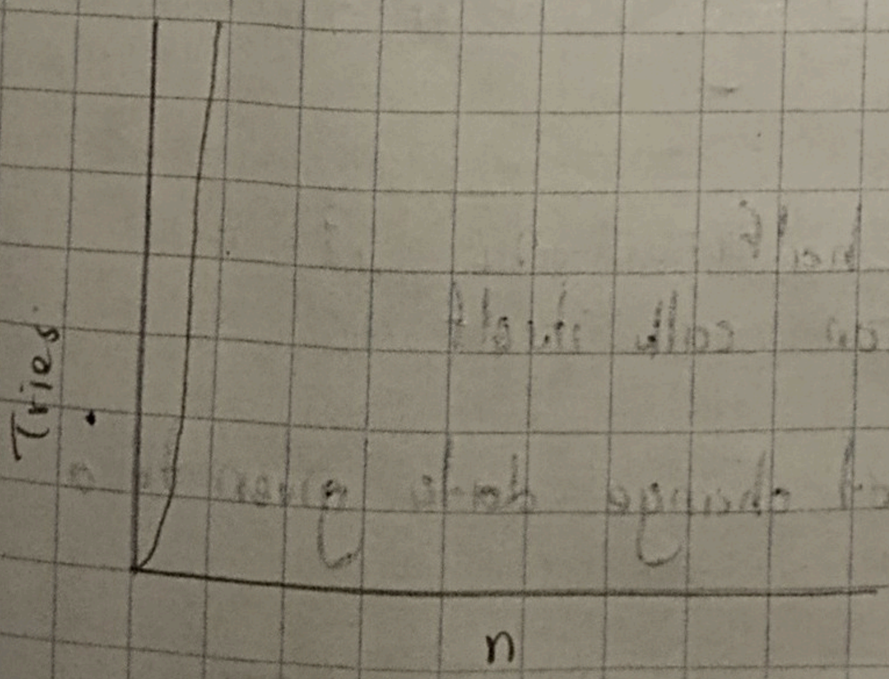
1. Brute force Algorithm.
   - Search thraugh each individual value to get an answer.
   - As n increases, ē no. of operations increase exponentially to a point where it is unsolvable.
   - So inefficient and useless.

2. Traveling Saleman.

Uses the factorial → n!
Basically $n(n-1)(n-2)\ldots(2)(1)$ repeated until you reach the ~~nxt~~ number 1.

e.g $3!$ $3 \times 2 \times 1 = 6$
$4!$ $4 \times 3 \times 2 \times 1 = 24$

Here ē most efficient algorithm will have a factorial/combinatorial RT.
Used for low n values.



Worst case scenario of Traveling Saleman problem is $O(n!)$

WEEK 2

How to determine the Complexity of an Algorithm.

For Binary Search: Assuming the list is sorted:

Step 1. - Determine the middle position (O(1))
Step 2. - Comparison Operation O 1
Step 3 (success case) → If middle value matches → Best case scenario: 1

Step 4 (if values don't match) → Alg will keep splitting the list into sub list to get the value. → O(log n)

Its upper bound → runtime of the least efficient step in the algorithm

INTRODUCTION TO ALGORITHMS.

a) Linear seach algorithm
   * Converted .py to its Cequivalent for easier understanding.

b) Binary search algorithm
c) Recursive binary search.
   * Different from the first,
   * Uses a shorter method to check if list is empty then returns false instead of None as seen in Binary search algorithm (the 1st one)

Difference between the two implementations
1. A recursive function is one that calls itself, but the other is simply direct

The recursive function calls itself inside it (if that even makes sense)
It includes a return statement, for e called out function

Instructing a function to ~~recall~~ return a value back to the function that called it.

Recursion and Space Complexity
For recursive functions:
With each call to itself, the size is cut in half
Recursive Depth → The number of times a function calls itself

Recursive solution → most preferred as it does not change data given to a function.

Space Complexity

Re A measure of how much working storage or extra storage is needed as an algorithm grows.
Measured in the worst case scenario using Big O Notation.

For the Iterative version of Binary search, for any value of N at e start, middle or end of the process, the amount of storage required does not get larger than N, thus → O(1)

Recursive binary search → Log O (log n)

Yay!!!

Iterative functions → Binary search.
Linear search.

Recursive functions → Recursive binary search.

Codes are in my laptop → will send.

WEEK 3

INTRODUCTION TO DATA STRUCTURES.

* Arrays
* Linked lists.

Arrays

for representing data values

Data Structure → a data storage format. It is a collection of value and the format they are stored in, the relationships between the value in the collection as well as the operations applied on the data stored in the structure

Arrays in:
Java → homogenous containers → can only store one data type. e.g int only
Python → heterogenous structures → can mix numbers and text.

Here e index is used for: accessing, inserting, updating, deleting.

Array → a contiguous data structure.

Homogenous memory → C, Swift, Java
→ Uses contiguous memory only.
Heterogenous memory → Python
→ Uses pointers and contiguous memory
(almost like linked lists in C)

Operations on Data Structures:
- Access and read values
- Search for an arbitrary value.
- Insert values at any point into the structure.
- Delete.

Implementation in arrays
Access: Uses index. position of that value
As long as it has the first index, it is easy to calculate the rest.

eg space in memory → M
size of array → N

space required = N * M

~~* Code *~~

new_list = [1, 2, 3]
result = new_list[0]

" " " Expected output: 1 " " "

Inserting Values
3 types of operations

i) Using an index value → to insert a value anywhere in the list.
its runtime → O(n) → Linear Runtime

ii) Appending → Simply adds items to the end of the list.
→ Constant time operation (O(1))

Eg:

1. numbers = []

size of list is equal to n+1
space starts of with space for 1 element since n=0

2. len(numbers)
   0
   → This means that ē list doesn't use memory allocation as an
   • indicator of its size.

3. numbers.append(2) → only has space 4. One element.
4. numbers.append(200) → forced to adjust to accommodate 2

• ē list, at this point, but it can add a new element to the list.
  it needs to increase the memory allocation, and ∴ ē size of ē
  list.
• Append operations take constant space ⇒ it has an ammortize
  constant space complexity. 2
  Also happens in insert operations.

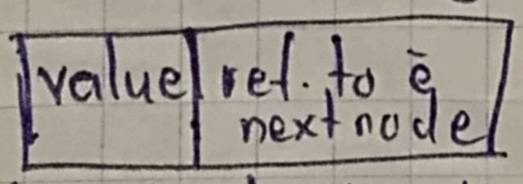
iii) A extend (in py) → ē ability to add one list to another.
  makes a series of append call to completely add all elements
  ~~to~~ of the new list to the original list.
  Runtime O(k)
  k → no. of elements in the list being added.

Delete Operation
• When a delete occurs, element shift to ē left for correct indexing
  Runtime → O(n)

Building a Linked list.

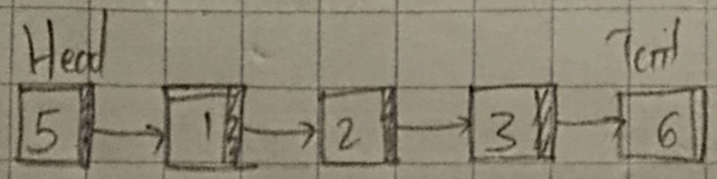
Why we build data structures.

Linked list → a linear data structure where each element in ē list is
contained in a separate object called a node.

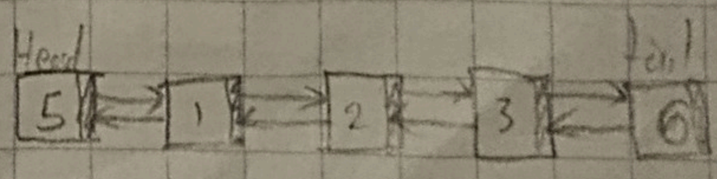
Node → 

↳ Self referential objects

Singly linked list.



Doubly linked list.



self.head = None → Default value of head is None so that new lists created are always empty

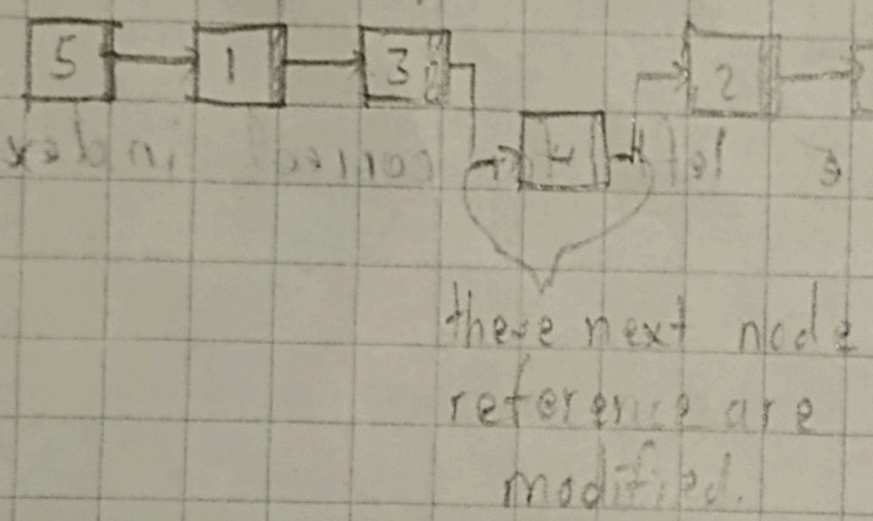
return self.head == None
if it returns True, it means the list is empty

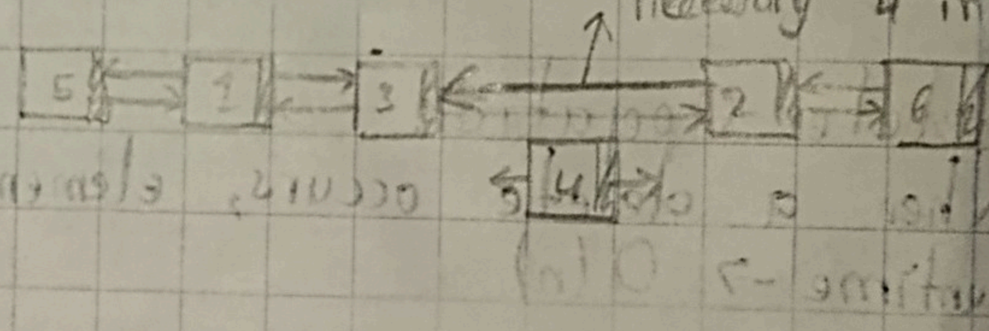
Ways of adding data to a list.

1. Adding nodes at the head of the list. → prepend
2. Adding nodes at the ~~end~~ tail of the list. → append
3. Insert nodes anywhere on the list → insert

Insert operation

~~We~~ Here, instead of shifting indices, we just need to change a few reference to next.
We can insert any node in the list at O(1) time.
Finding the position in the list takes O(n) time





Remove Operation.

1st need to search for data that matches the key.
The next node references also need to be modified, just like in insert.

Almost there.

WEEK 4

HeHeHe, → already did merge sort.

Used code.

Recursive function.

verify_sorted → checks if ē 2nd elemen (at index 1) is less than index 0 element-

1	2	3	4	5	False
2	3	4	5	False	
3	4	False			
4	5	False			
5	True				

Cost of This algorithm-

Split function
→ finding midpoint and splitting list at midpoint-
→ is recursive.
→ Takes overall $O(\log n)$ time.

Merge step
→ split into single elements
→ make comparison operations
→ merge them back together.

for a list of size $\underline{n}$, we have $\underline{n}$ number of merge operations.
→
Overall runtime $O(n \log n)$

Runs in overall linear runtime $O(n)$

∴ for Merge Sort, it takes $O(n \log n)$ time.

Split function takes $k * O(\log n)$
$= O(k \log n)$

OVERALL top level function take
$O(kn \log n)$

To fix this, we ~~use~~ remove ē slicing part

Space Complexity

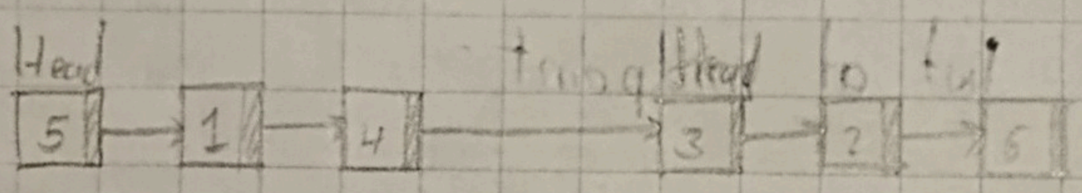
Takes O(n) space.

Coz the code does not execute every part simultaneously
They The left size is run first then merge, then the right side is also sorted and merged.
Thus, no new memory is allocated and it uses its original memory

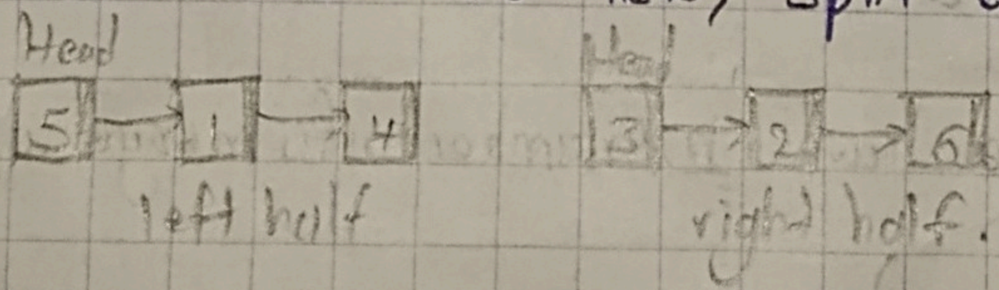
Additional space required by the algorithm at a given time.
=n
At diff point in the algorithm, we require
= log n amount of space.

Implementing Merge Sort on a Linked List.

The implementation differs, and so does the runtime

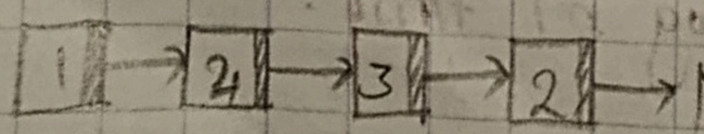


find the midpoint
The node after the node at ē midpoint is assigned to the head of a newly created linked list. and ē connection between the midpoint node and the one after is removed.
We now have two lists, split at the midpoint.

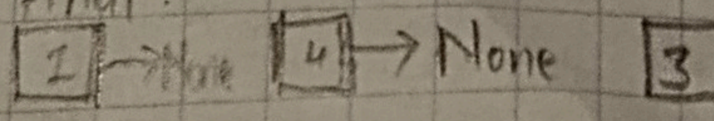


def merge(left, right)

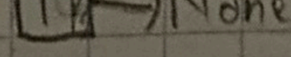
merges two linked lists, sorting by data in nodes.
returns a new, ~~linked list.~~ merged list



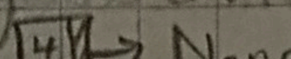
keep calling splits until we have a single head
final:



left_head

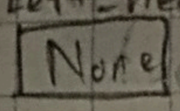


right_head

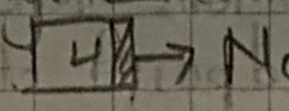


Comparison for the lower value leaves left_head empty.

left_head

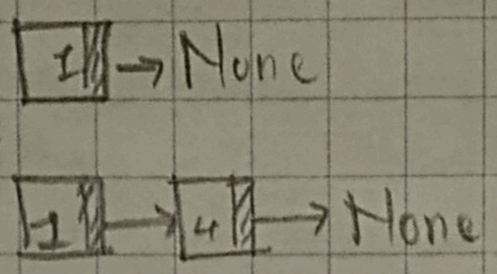


right_head



∴

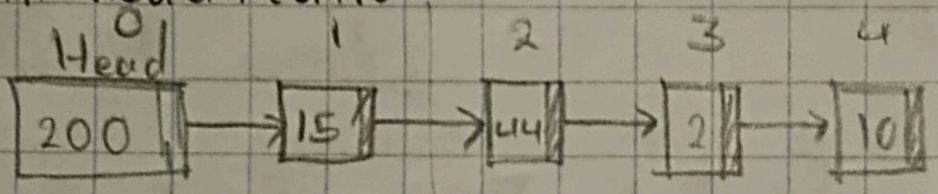
Final Merged List:



split operation

Now both lists are empty, thus the ~~alg~~ terminates
Vice versa is true.

Full visualization

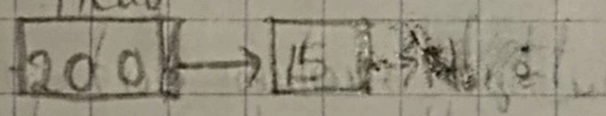


1. Split operation -

midpoint = 2

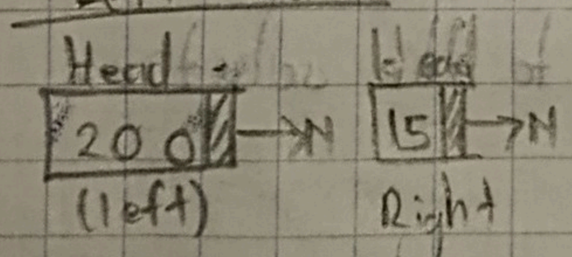
Assign ē list to left and right halves

Left

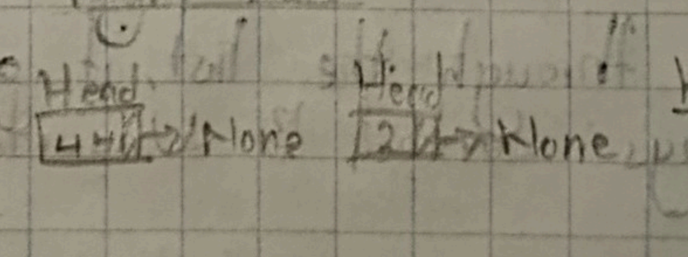


Split operation is carried out multiple times, first on the Left half till we get single nodes then on the right.

Left half

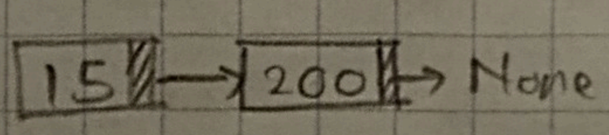


Right half

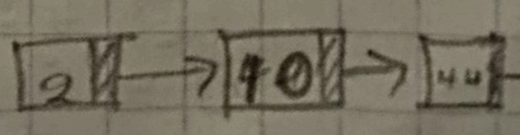


Do a merge sort on them where we create a new linked list then check on which is smaller and assign it to the head.

Left half

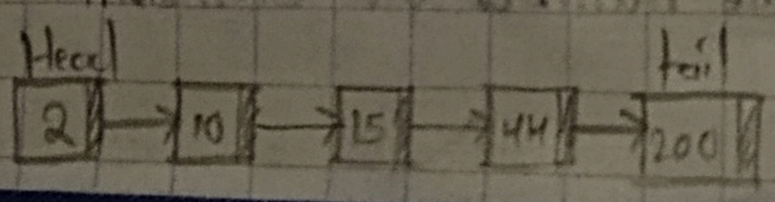


Right



We recursively split and repeatedly merge sublists to get one final sorted list.

Final



A true implementation of merge runs in Quasilinear time or log linear time ie. $(n \times \log n)$

Swapping operation and comparison operation take $O(1)$ time
Split operation ⇒ $O(k)$ where k is the midpoint of the list.
⇒ take $O(\log n)$ time
∴ Overall, it takes $O(k \log n)$ time
Merge function ⇒ $O(n)$ time

Finally overall runtime
$= O(kn \log n)$

FINAL LAP : WEEK 5

SORTING ALGORITHMS

Bogo Sort

It randomizes the order of the list until it is sorted.
If time the algorithm is run, it takes a random number of attempts to sort values
Might generate a list with either one or multiple values out of order
Problem ⇒ does not get close to a solution with each operation.

Selection Sort.

With each pass through the list, it gets closer to the solution.
Uses two arrays: Sorted & Unsorted.

Unsorted	Sorted
[8, 5, 1, 4, 7]	[]
[8, 5, 4, 7]	[1]
[8, 5, 7]	[1, 4]
[8, 7]	[1, 4 5]
[8]	[1, 4, 5, 7]
[]	[1, 4, 5, 7, 8]

Disadv: Takes longer time to sort as the number of elements increases

These next two rely on Recursion. → Quicksort
→ ~~Recur~~ Merge Sort -

Base case → when there is no element left to add
Recursive case → Where there are still elements to add

This function relies on calling itself until we either get the target value or we run out of elements

Quicksort.py

Reduces the number of comparisons it makes
It relies on recursion.
It is recursive because it keeps calling itself with smaller and smaller subsets of the elements needed to be sorted.

How it works

[4, 6, 3, 2, 9, 7, 3, 5]

Pick any value from the list and it becomes the pivot.

[4]
pivot

Break the list into two sublists for values smaller than or greater than the pivot value.

less than pivot
[3, 2, 3]

> pivot
[6, 9, 7, 5]

We now call the quicksort function recursively

< pivot.

[3]
pivot
[2, 3] [3]

[2, 3]

[2] pivot

[ ] [2] [3]

[2, 3] + [3]
[2, 3, 3]

pivot.
[4]

> pivot.
[6] pivot.
[5] [6] [7, 9]
[7] [9]

eventually

[5, 6, 7, 9]

Combine everything now.

[2, 3, 3, 4, 5, 6, 7, 9]

Better workflow for quicksort.

[4, 6, 3, 2, 9, 7, 3, 5]

[3, 2, 3] 4 [6, 9, 7, 5]

[2, 3] 3 [ ]

[ ] [2] [3]

[5] 6 [9, 7]

[2, 3, 3, 4, 5, 6, 7, 9]

B Clearer workflow

[4, 6, 3, 2, 9, 7, 3, 5]

[3, 2, 3] 4 [6, 9, 7, 5]

[2, 3] 3 [ ]

[ ] 2 [3]

[5] 6 [9, 7]

[7] 9 [ ]

[2, 3, 3, 4, 5, 6, 7, 9]

NB: Trying running this on a file with repeated values will generate an error.

Faster than Selection Sort (very much faster)

* Merge sort is also recursive.

Recursive merge sort workflow

[4, 6, 3, 2, 9, 7, 3, 5]

[4] [6]

[3] [2]

[4, 6] [2, 3]

[9] [7]

[3] [5]

[7, 9] [3, 5]

[2, 3, 4, 6] [3, 5, 7, 9]

[2, 3, 3, 4, 5, 6, 7, 9]

Very fast for veeery large numbers

for 10000 values, Quicksort is the quickest, then merge sort, then selection sort.
→ too slow for very very large files thus barely used.

Big O Notation

As a way of quickly describing how an algorithm performs as the size of its data set increases

Selection sort ⇒ $O(n^2)$

Quicksort ⇒ O (log n) times ⇒ for ~~selecting~~ dividing a list in half.
best case ⇒ O (n log n) ⇒
worst case ⇒ $O(n^2)$ times.

Merge Sort ⇒ O (n log n)

ALGORITHMS: SORTING AND SEARCHING

If we have a list of names and we are looking for a specific one:
Best choice:
Step 1: Use quicksort to sort the list in order
Step 2: Use binary search to get the name.

THE END.