

Python DSA Notes

Built in Data Structures

- List: Ordered, mutable collection of elements
- Tuple: Ordered, immutable collection of elements
- Dictionary: Unordered, Key-value pairs
- Sets: Unordered, unique collection of items

List, Tuple can have different data types as collections.

Arrays: Not built-in in python. Ordered collection of homogeneous elements

Drawbacks: - Fixed Length
- Homogeneous elements

Memory size in bytes

char - 1 byte
int - 2 bytes
Long - 4 bytes
float - 4 bytes
double - 8 bytes

Array

Data Type

Homogeneous
(same type)

Flexibility

Fixed size

List

Heterogeneous
(different type)

Dynamic
(can grow & shrink)

Performance

More memory efficient
for large data

(can grow & shrink)

Slower for large
data

Usage in python

Use array or numpy

Built-in

Searching and Sorting

Linear Search

arr, target

return the target index if
present else -1

10	23	45	70	11
0	1	2	3	4

Implementation - Run a loop and compare.

Binary Search

arr, target

* sorted

0	1	2	3	4	5	6
10	23	35	45	50	70	85
Start		mid		End		

1. I will consider the full array/list
Start = 0
End = n-1
2. See the middle element
middle = (end - start) / 2 [Floor division]
3. We compare element at middle with target.

1st iteration \rightarrow mid = 3

$$arr[mid] = 45 < 50$$

2nd iteration \Rightarrow start = 4, mid = 5, end = 6
 $arr[mid] = 70 > 50$

3rd iteration \Rightarrow start = 4, end = 4, mid = 4
 $arr[mid] = 50 == 50$

* Only works when array is sorted

Sorting Algorithm

Bubble Sort Algorithm

1. The largest element i.e, 90 is at the right position
2. We needed len-1 passes to that

1st pass

12 25 11 34 90 22
 1st 25 11
 2nd 12 11 25 34 90 22
 3rd 25 34
 4th 34 90
 5th 90 22
 12 11 25 34 22 90

In 2nd pass, the second largest is the the right position

2nd Pass

1st 11 12 25 34 22 90
 2nd 12 25
 3rd 25 34
 4th 34 22
 11 12 25 22 34 90

Selection Sort

Select

In each pass we see

12 25 11 34 90 22

1.

the array and select the minimum
we swap that to the right position

\downarrow
11 12 22 25 34 90
 minimum select

$n-1$ pas { Pass 1 11 25 12 34 90 22
 Pass 2 11 12 25 34 90 22
 Pass 3

Insertion Sort

8 | 2 | 5 | 10

2 | 8 | 5 | 10

First round not required as 1st element always sorted

12 25 11 34 90 22
 0 1 2 3 4 5

Round 1 12
 Round 2 12 25
 Round 3 11 12 25
 Round 4 11 12 25 34
 Round 5 11 12 25 34 90
 Round 6 11 12 22 25 34 90

Recursion

Principle of Mathematical Induction (PMI)

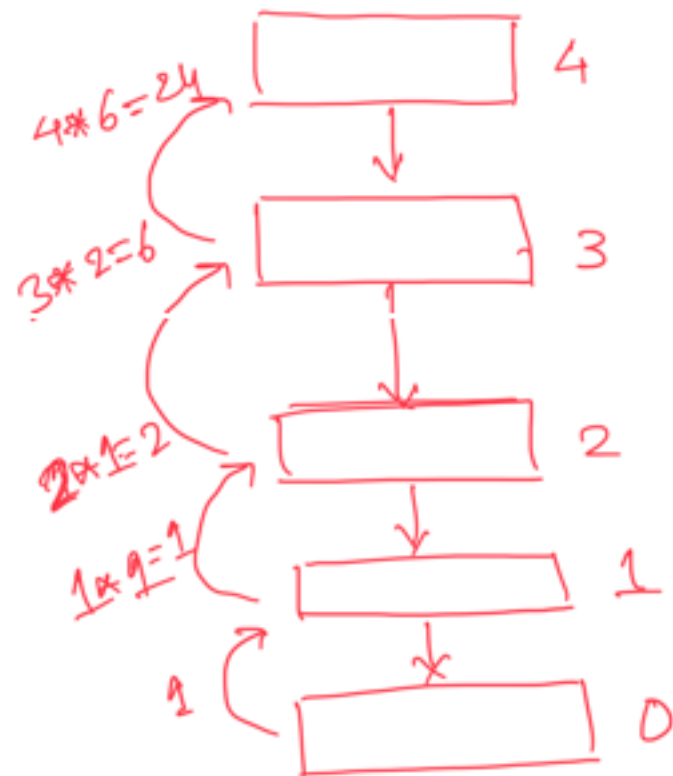
- ① Prove $F(0)$ & $F(1)$ are true (Base Case)
- ② Assume $F(k-1)$ is true or assume it is true for all k from 0 to $k-1$

③ Using ① & ②, calculate $F(k)$

Factorial of a number

- ① Base case $0! = 1$
- ② Recursion relation: $\text{fact}(n) = n * \text{fact}(n-1)$
- ③ our work: multiply n with small ans

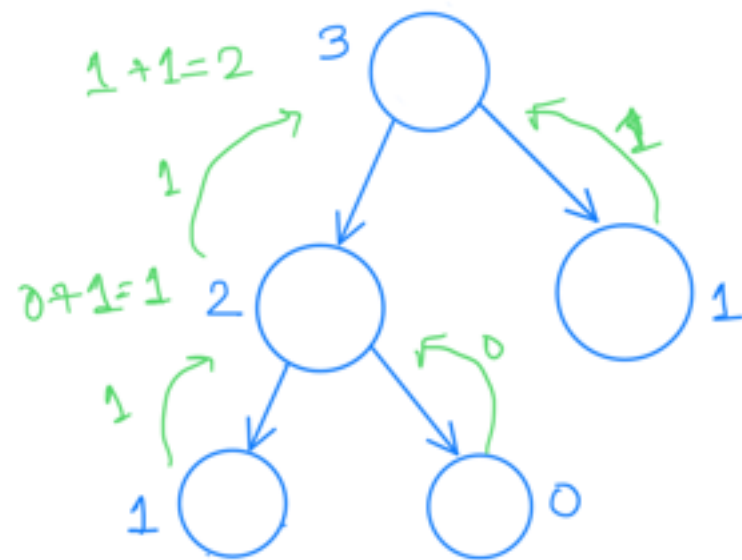
Recursion Tree ($4!$)



Fibonacci series

- ① Base Case: $F(0) = 0$, $F(1) = 1$
- ② Recursion Relation: $F(n) = F(n-1) + F(n-2)$
- ③ Do recursion call to get last and second last digit. Then add them up.

Recursion Tree (for $n=3$)



Head Recursion

- ① Define base case
- ② Do recursion call
- ③ Do your work

e.g. \rightarrow Print from 1 to N

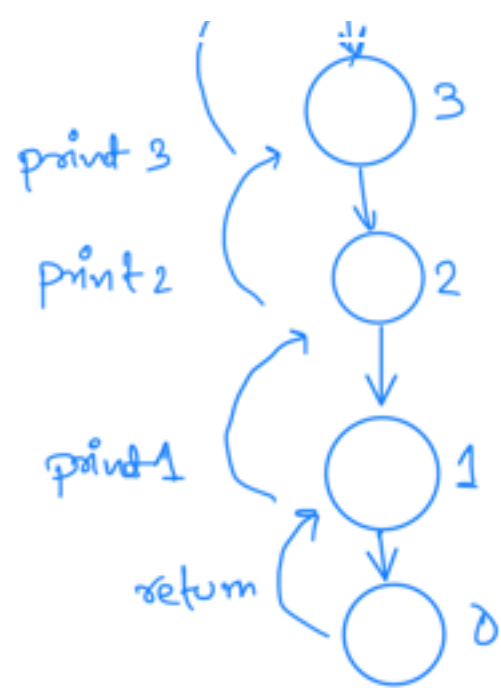
Base case: if $n == 0$:
return

Recursion call: $\text{print1toN}(n-1)$

Your work: $\text{print}(n)$

Recursion Tree ($n=4$)





Tail Recursion

- ① Define base case
- ② Do your work
- ③ Do recursion call

E.g. \rightarrow Print N to 1

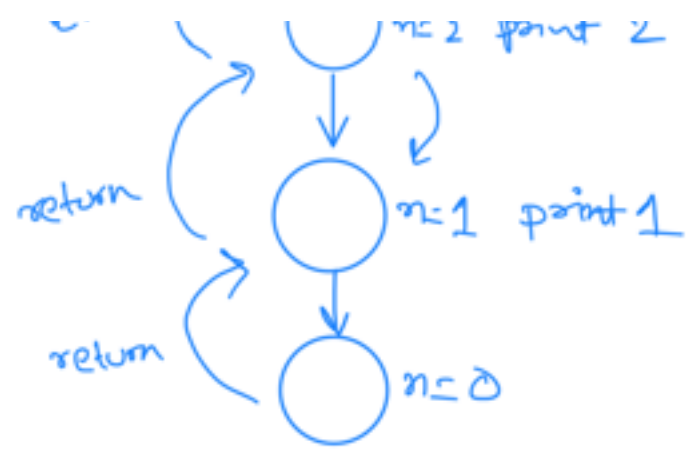
Base case : if $n == 0$;
return

Your work : print n

Recursion call : print N to 1 (n-1)

Recursion Tree (n: 3)





Recursion & Arrays

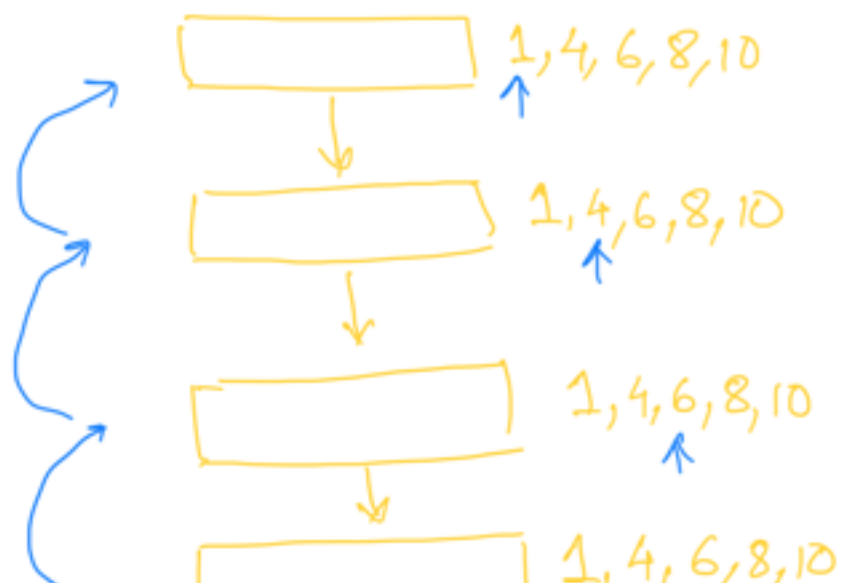
① Check if Array is sorted

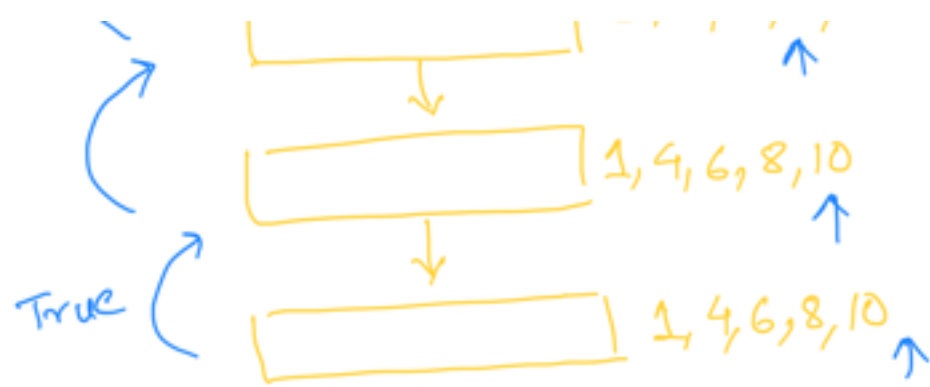
Base case : if $\text{len}(\text{Arr}) == 0$: or if $\text{len}(\text{Arr}) == \text{index}$:
return True

My work : if $Arr[0] > Arr[1]$ or if $Arr[index] > Arr[index+1]$
return False return False

Recursion call : $\text{check_sorted_array}(\text{Arr}[1:])$
or
 $\text{check_sorted_array}(\text{Arr}, \text{index} + 1)$

Recursion Tree [1, 4, 6, 8, 10]





First Index of an element in array

Arr = [2, 4, 7, 3, 4, 8]
 0 1 2 3 4 5

elem = 4

first index = 1

Base case : if $\text{len}[\text{Array}] == \text{index}$:
 return -1

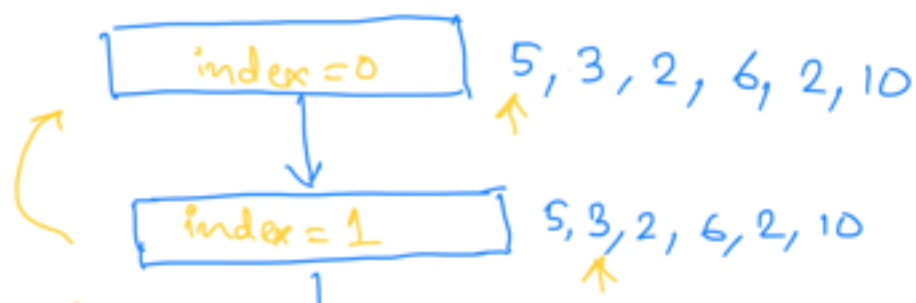
My work : if $\text{Array}[\text{index}] == \text{elem}$:
 return index

Recursive call : $\text{get_first_index}(\text{Arr}, \text{index} + 1)$

Recursion Tree

Arr = [5, 3, 2, 6, 2, 10]

elem = 2



2 ↗ index = 2 5, 3, 2, 6, 2, 10

Find last index of an element in an array

Arr: [2, 3, 4, 5, 3, 6]
0 1 2 3 4 5

elem = 3

last index = 4

Base case: if $\text{len}(\text{Arr}) == \text{index}$:
return -1

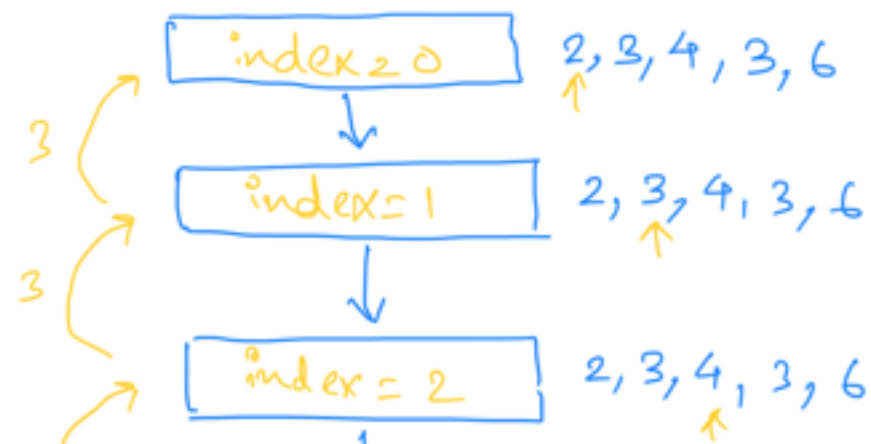
Recursion call: $\text{st_index} = \text{get_last_index}(\text{Arr}, \text{index} + 1)$

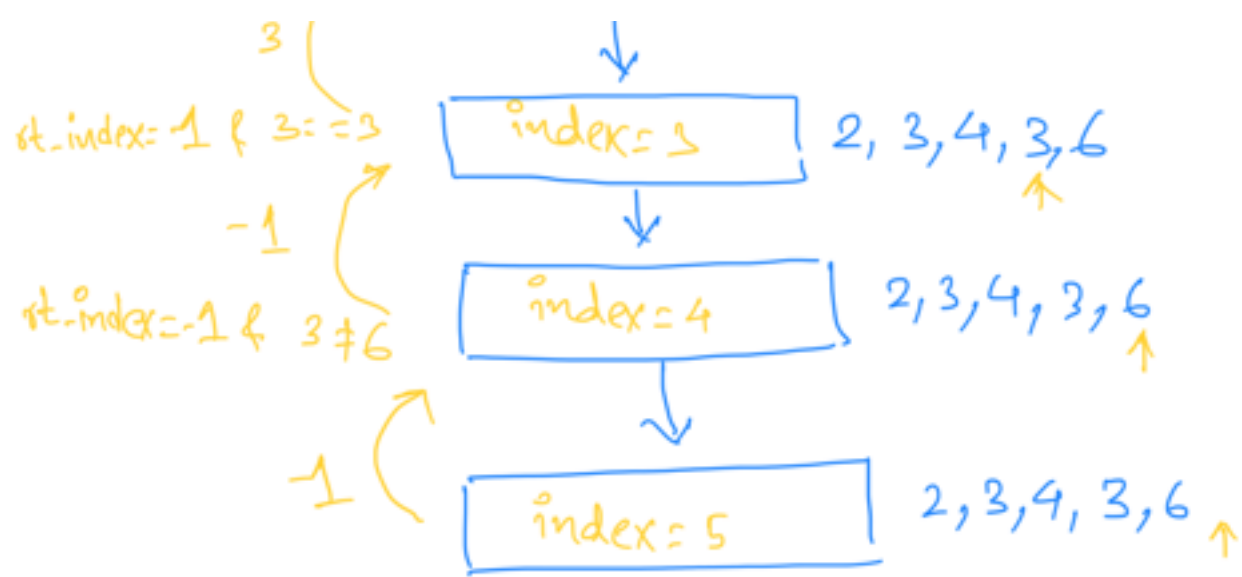
My work: if $\text{st_index} == -1$ and $\text{Arr}[\text{index}] == \text{elem}$:
return index
return st_index

Recursion Tree

Arr = [2, 3, 4, 3, 6]

elem = 3





Print all index of an element

Arr = [3, 2, 5, 2, 6]
 0 1 2 3 4

elem = 2

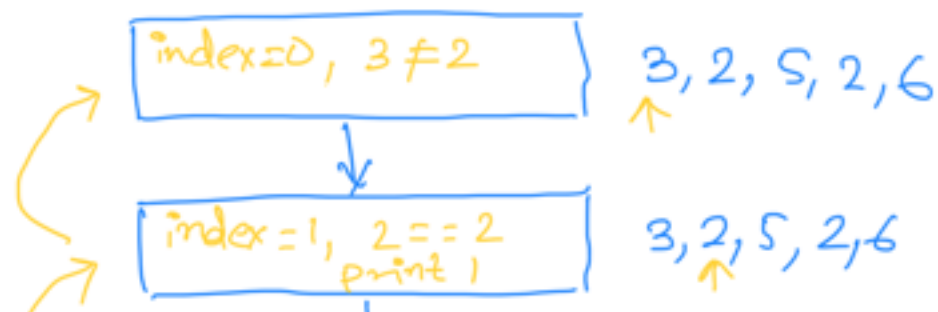
indices = 1 3

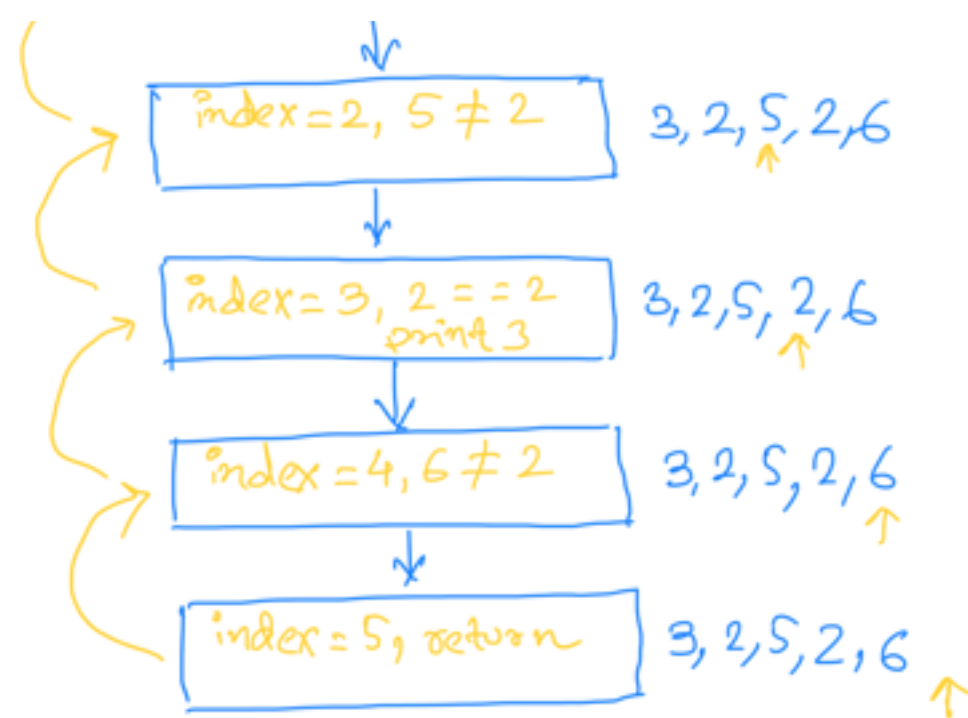
Base Case : if len(Arr) == index:
 return

My work : if Arr[index] == elem
 print index

Recursion call : print_index_of_elem(Arr, index+1)

Recursion Tree





Update all indices in the list provided in function
or Global list

Arr = [3, 2, 5, 2, 6]

elem = 2

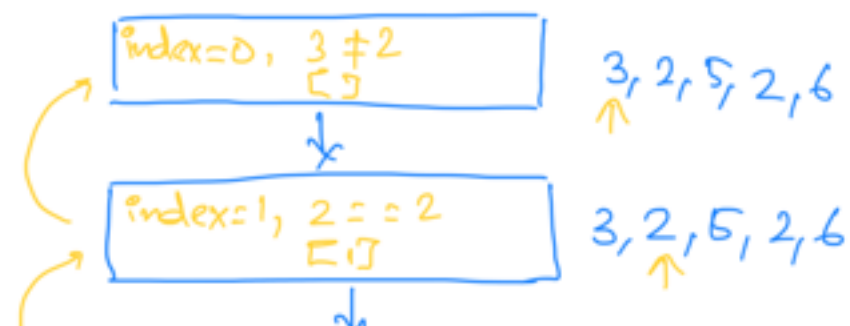
ans = [1, 3]

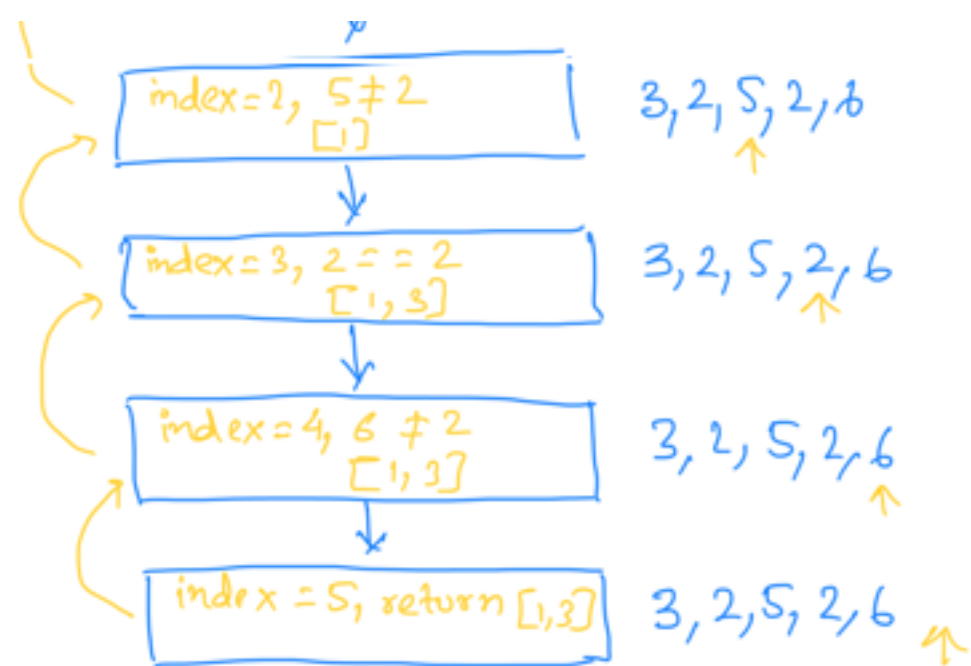
Base Case : if $\text{len}(\text{Arr}) == \text{index}$:
return lst

My work : if $\text{Arr}[\text{index}] == \text{elem}$:
lst.append(index)

Recursion Call : update_indices(Arr, index+1, lst)

Recursion Tree





Return indices of element in a new list

Arr = [3, 2, 5, 2, 6]
 0 1 2 3 4

elem = 2

lst = [1, 3]

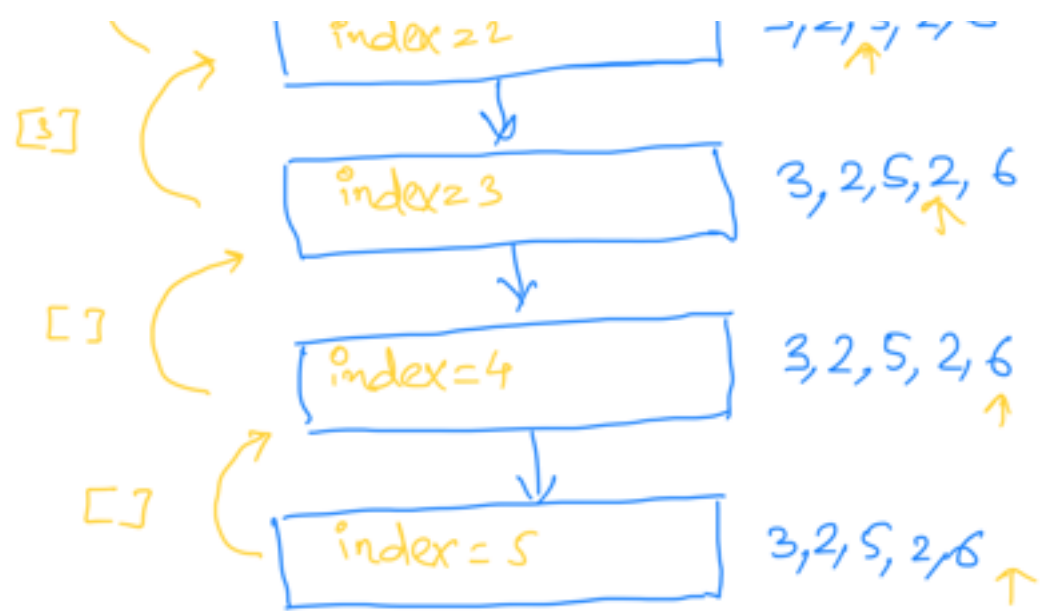
Base case : if $\text{len}(\text{Arr}) == \text{index}$:
 return []

Recursion call : ans_lst = get_indices_of_elem(Arr, index+1)

My work : if $\text{Arr}[\text{index}] == \text{elem}$:
 ans_lst.insert(0, index)

Recursion Tree





Searching and Sorting using Recursion

Linear Search

Arr = [1, 3, 5, 2, 6]

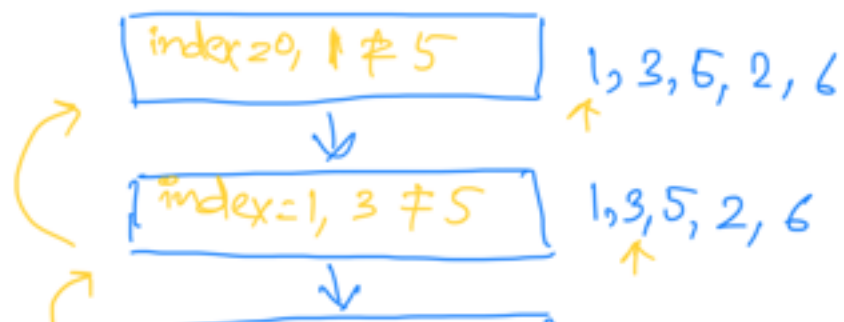
target = 5

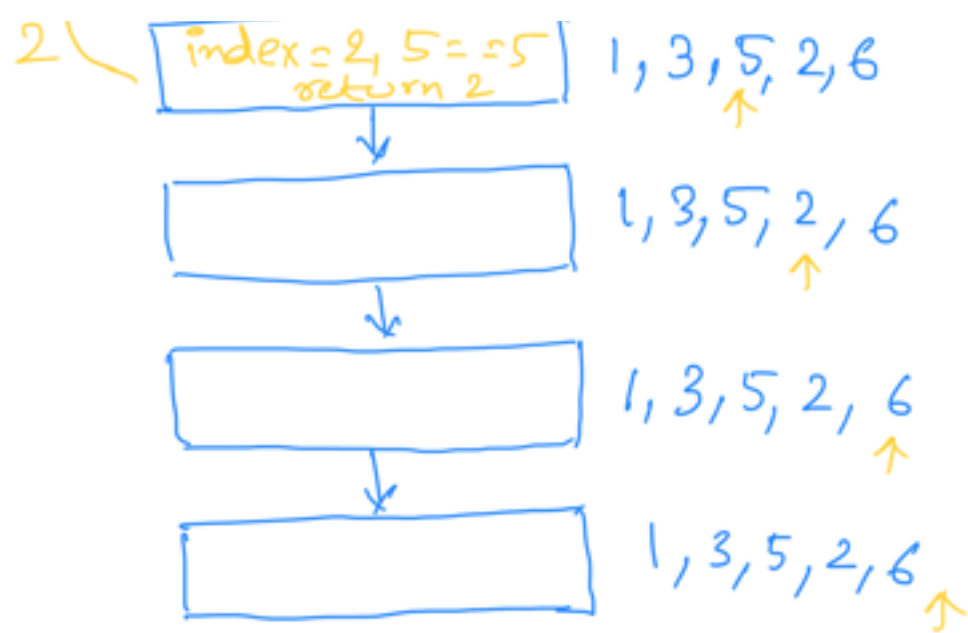
Base Case : if $\text{len}(\text{Arr}) == 0$:
return -1

My work : if $\text{Arr}[\text{index}] == \text{index}$:
return index

Recursion call : $\text{Linear_search}(\text{Arr}, \text{index} + 1)$

Recursion Tree





Binary Search.

Arr = [2, 4, 5, 6, 8, 10]

target = 5

Base Case : if start > end :
return -1

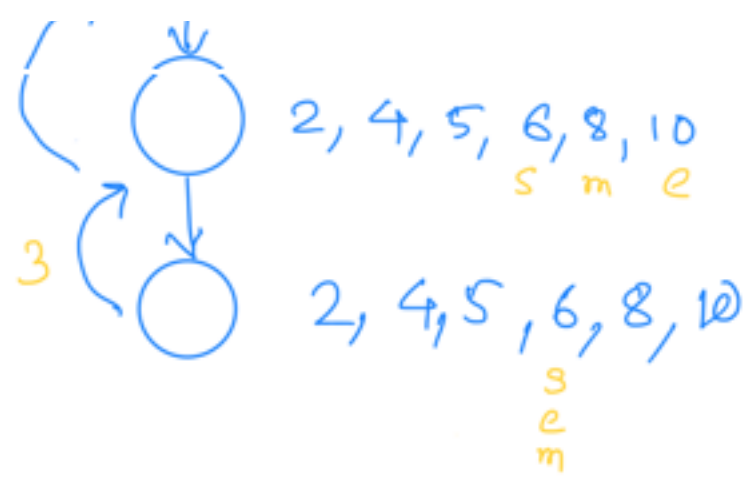
My work : mid = start + (end - start) // 2
if Arr[mid] == target : return mid

Recursion call : if Arr[mid] > target :
return binary_search(Arr, start, mid-1)
return binary_search(Arr, mid+1, end)

Recursion Tree

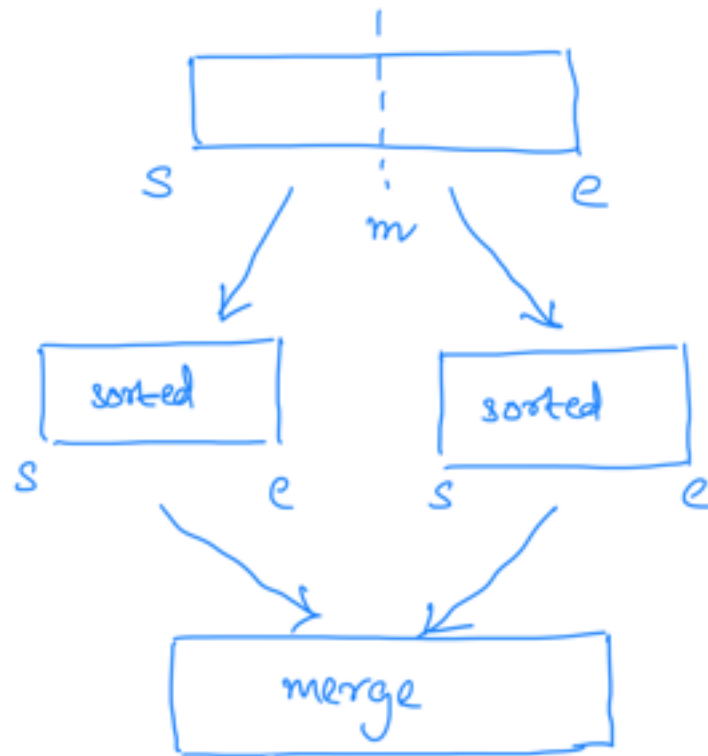
[2, 4, 5, 6, 8, 10] → 6





Merge Sort

Arr = $[5, 8, 9, 3, 4]$



Base Case : if $start \geq end$:
return

Recursion Call : $mid = start + (end - start) // 2$
merge_sort (Arr, start, mid)
merge_sort (Arr, mid+1, end)

My work : merge (Arr, start, mid, end)

$i = \text{start}$
 $j = \text{mid} + 1$ $\text{new_lst} = []$

while $i \leq \text{mid}$ and $j \leq \text{end}$:

if $\text{Arr}[i] < \text{Arr}[j]$:

$\text{new_lst.append}(\text{Arr}[i])$
 $i = i + 1$

elif $\text{Arr}[i] > \text{Arr}[j]$:

$\text{new_lst.append}(\text{Arr}[j])$
 $j = j + 1$

elif $\text{Arr}[i] == \text{Arr}[j]$:

$\text{new_lst.append}(\text{Arr}[i])$
 $\text{new_lst.append}(\text{Arr}[j])$
 $i = i + 1$
 $j = j + 1$

while $i \leq \text{mid}$:

$\text{new_lst.append}(\text{Arr}[i])$
 $i = i + 1$

while $j \leq \text{end}$:

$\text{new_lst.append}(\text{Arr}[j])$
 $j = j + 1$

Remaining
elements from
list

$\text{existing_list_index} = \text{start}$

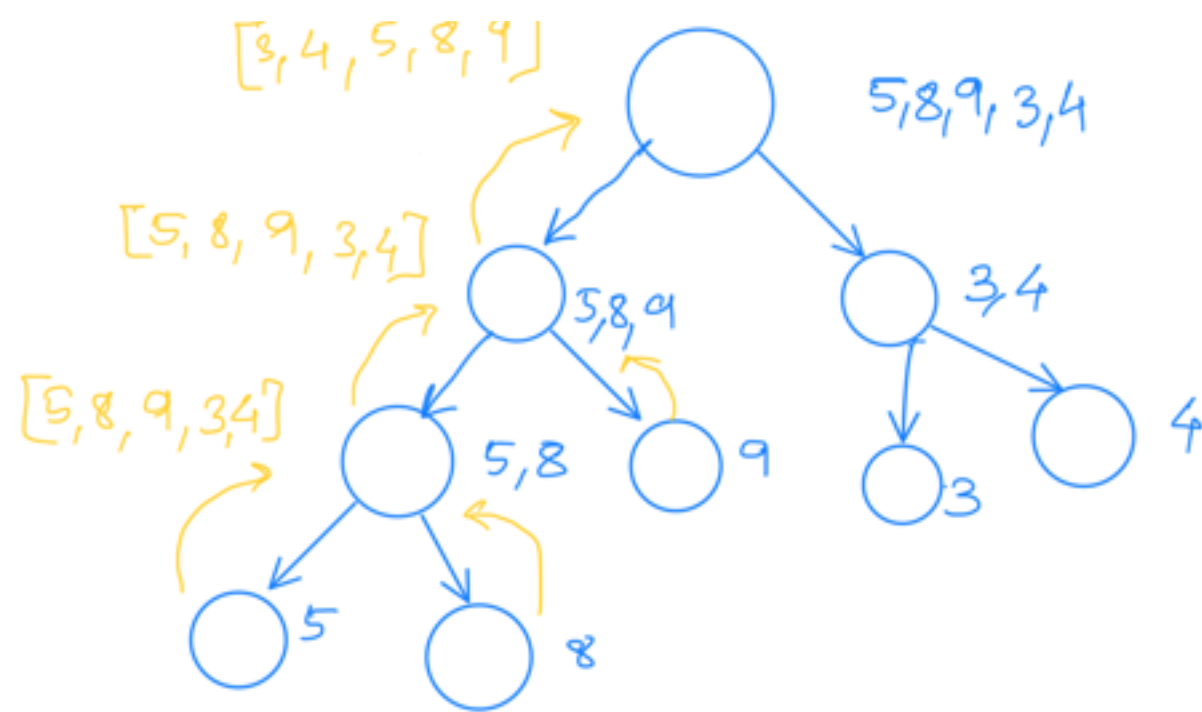
$\text{new_list_index} = 0$

while $\text{existing_list_index} \leq \text{end}$:

$\text{Arr}[\text{existing_list_index}] = \text{new_lst}[\text{new_list_index}]$

$\text{existing_list_index} += 1$

$\text{new_list_index} += 1$



Quick Sort

Arr = [4, 8, 3, 9, 5]

- ① Take last elem of array as pivot elem
- ② Find the right position of the pivot elem in the array
- ③ Swap the pivot elem with right position elem.
- ④ Move all elements smaller than pivot elem to the left and all elements larger than pivot elem to the right
- ⑤ Apply quick sort to the arrays to the left and right of the pivot elem.

Base case : if start \geq end:
return

My work : partition(Arr, start, end)

pivotElem = Arr[end]

pivotIndex = start

```

i = start
while i < end:
    if Arr[i] < pivotElem:
        pivotIndex += 1
    i += 1

Arr[pivotIndex], Arr[end] = Arr[end], Arr[pivotIndex]

j = start
k = end
while j <= pivotIndex and k > pivotIndex:
    if Arr[j] < pivotElem:
        j += 1
    elif Arr[k] > pivotElem:
        k -= 1
    else:
        Arr[j], Arr[k] = Arr[k], Arr[j]
        j += 1
        k -= 1

```

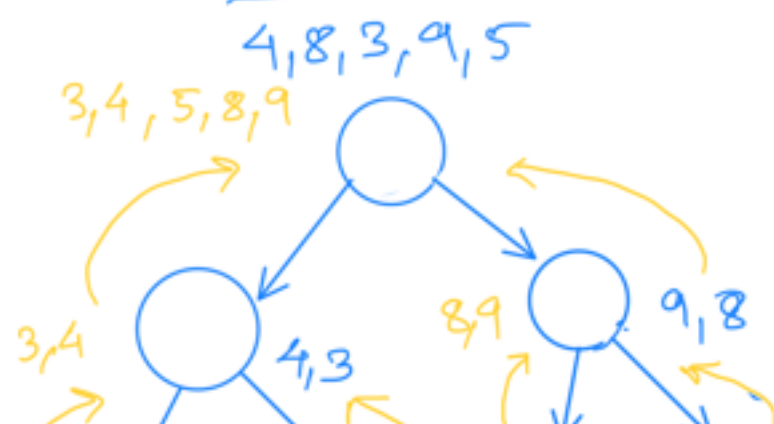
Recursion Call :

```

quick_sort(Arr, start, pivotIndex-1)
quick_sort(Arr, pivotIndex+1, end)

```

Recursion Tree





Recursion and Strings

Palindrome Check with Recursion

nitin → palindrome n i t i n e
 abc → not palindrome s e

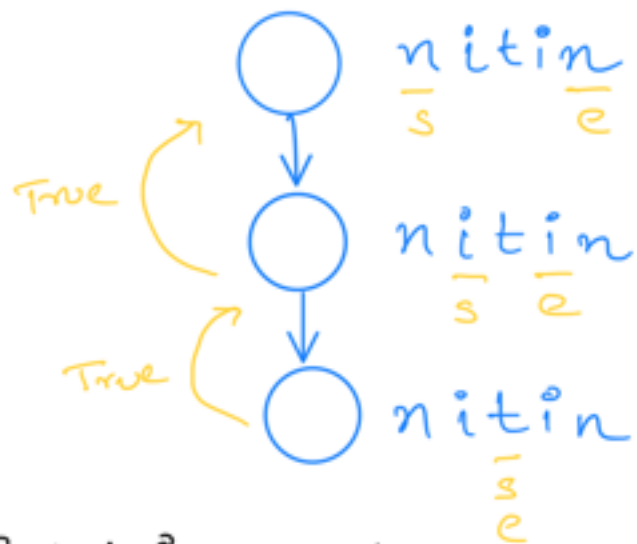
malayalam → palindrome

Base Case : if start > end:
 return True

My work : if str[start] != str[end]:
 return False

Recursion Call : palindrome_check(str, start+1, end-1)

Recursion Tree



String 0 1 2 3 4 5

Strings, Substrings and Subsequences

string = 'abc'

substring = ordering to be maintained
and it should be continuous

a¹ b² c³

ab⁴ bc⁵ ~~ac~~

abc⁶

$$\frac{n(n+1)}{2}$$

$$= \frac{3(3+1)}{2}$$

$$= 6$$

Subsequence : order maintained and they can be
non-continuous as well

a b c

ab bc ac

abc ϕ



either take it or
not take it

Return Subsequences of a given string

abc \rightarrow a b c

ab bc ac

abc ϕ

Base Case : If $\text{len}(\text{str}) == 0$ or if $\text{len}(\text{str}) == \text{index}$
return ['']

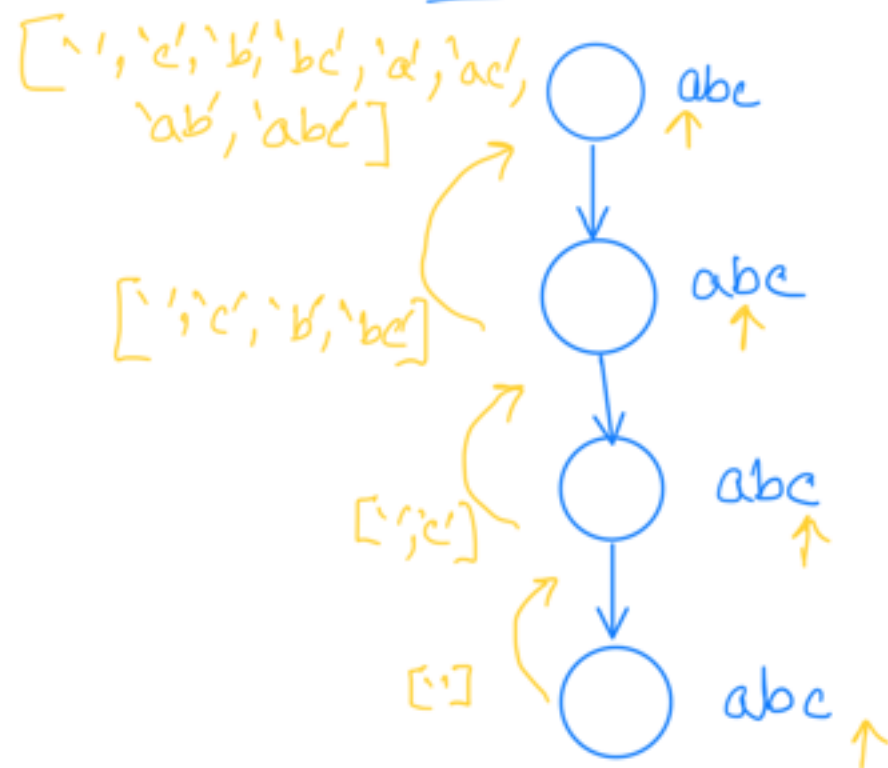
Logic : I will keep 'a' and will pass 'bc' to
recursion call and expect recursion call to give all

subsequences of 'bc'. Then will concatenate 'a' to all those subsequences to get all combinations.

Recursion Call : $\text{return_subsequences}(\text{str}[1:])$
or
 $\text{return_subsequences}(\text{str}, \text{index}+1)$

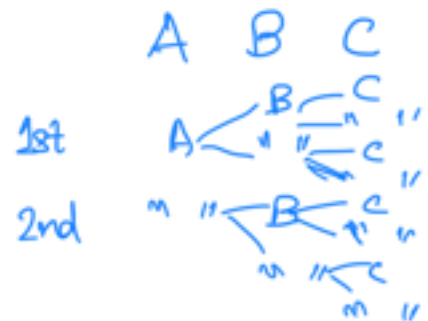
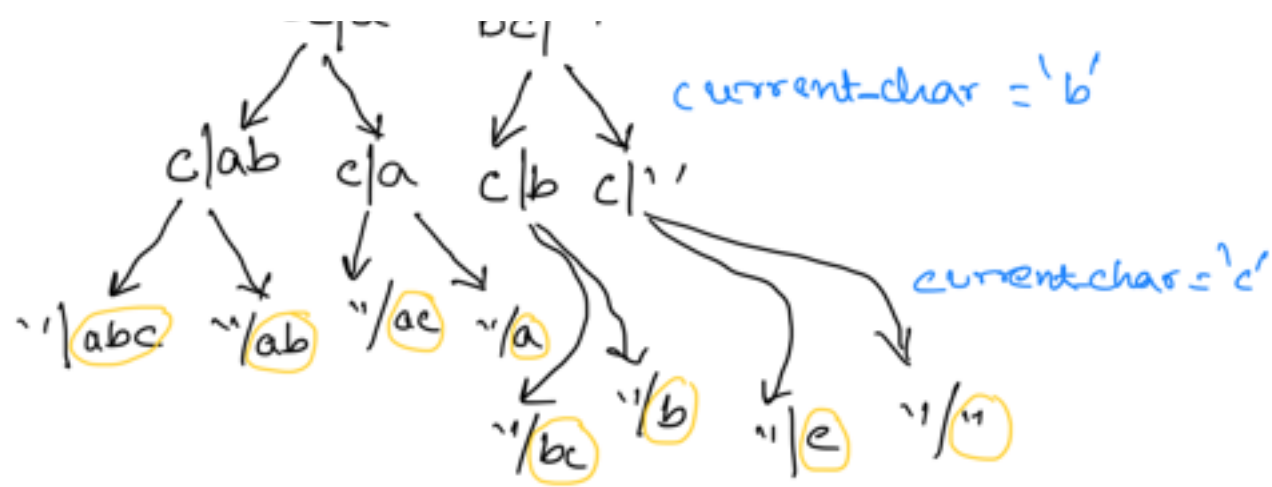
My work : $\text{myChar} = \text{str}[0]$ or $\text{str}[\text{index}]$
for subsequence in subsequences:
 $\text{subsequences.append}(\text{myChar} + \text{subsequence})$

Recursion Tree



Print Subsequences of a given String





Print Permutations of a given string

string → "abc"

permutations → abc acb
bac bca
cab cba

↓ ↓ ↓
1 2 3

$n!$ ways

abc|""

↓
bc|a_

↙ ↘
c|_b_a_ c|_a_b_

↙ ↘ ↘ ↙ ↘ ↘ ↙ ↘ ↘
""|cba ""|bca ""|bac ""|cab ""|acb ""|abc

Return Permutations of a String

String \rightarrow 'abc'

Approach : Keep current character and send smaller string to recursion call. Assume recursion call gives the permutations for smaller string. Then add current character in all possible positions to the returned permutations.

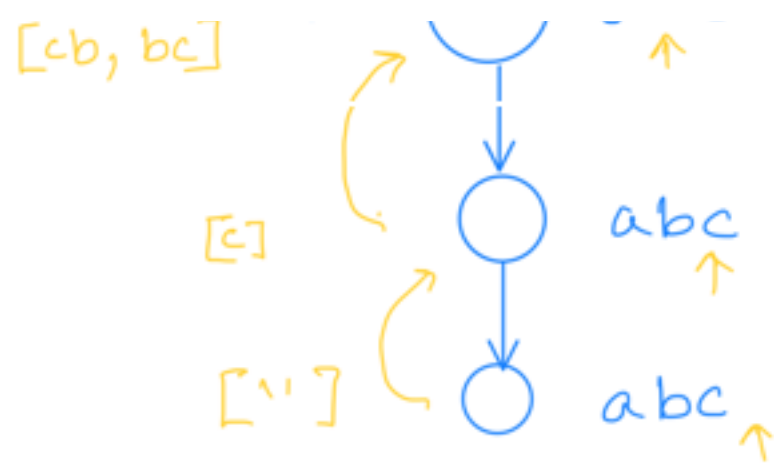
Base Case : if $\text{len}(s) == \text{index}$ or $s == ''$:
return ['']

Recursion call : $\text{permutation_list} = \text{permutations}(s, \text{index})$

My work : $\text{current_char} = s[\text{index}]$, $\text{new_list} = []$
for permutation in permutation_list :
 for position in range(len(permutation)) :
 $\text{new_list.append}(s[0:\text{position}] + \text{current_char} + s[\text{position}:])$
return new_list

Recursion Tree



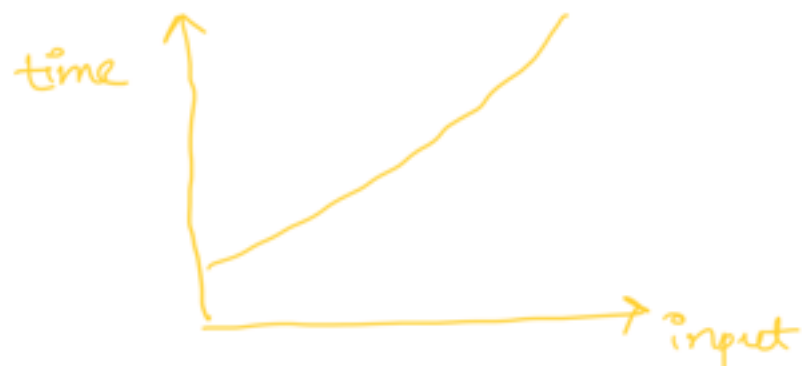


Time Complexity Analysis

Problems with experimental analysis

1. Different times on different run
2. Few changes, time can be different
3. Time actually varies with input, but we are not getting or establish a relationship.
4. Generate test cases. (worst cases)
5. Check for each and every implementation.
6. Large inputs are very time consuming.

Time complexity \neq Time taken
 Time complexity is actually how time is related or dependent on input size.



Quantifying the Time Complexity

1. We want time complexity when input is very large.
2. Worst Test case
3. We want to look for biggest factor in the run time

$n^2 + n \rightarrow n^2$ is the dominating factor for large values of n and n can be ignored

nested loop $(n^2) >$ single loop (n)

4. We want to express runtime as input size and we don't want to look for precision but 'order' of works



if the relationship is $20n^2 + 5n + 1$

we are fine with most dominant term: $20n^2$

No. of unit operations: $+$, $-$, $>$, \div , $=$

3 major things:

1. Talk in terms of no. of operations not time
2. Focus only on the highest power
3. Don't care about coeff. much.

Asymptotic Notation

Main idea of our analysis is to do have a measure of the efficiency of algorithm that don't depend on.

- machine specific constants
- require algorithm to be implemented.

Asymptotic notation are mathematic tools to represent time complexity of our algorithm

There are mainly 3 types of asymptotic notation

1. Big O notation (O notation, worst case)
2. Omega notation (Ω notation, best case)
3. Theta notation (Θ notation, avg case)

Big O notation

Function of an algorithm based on no. of operations

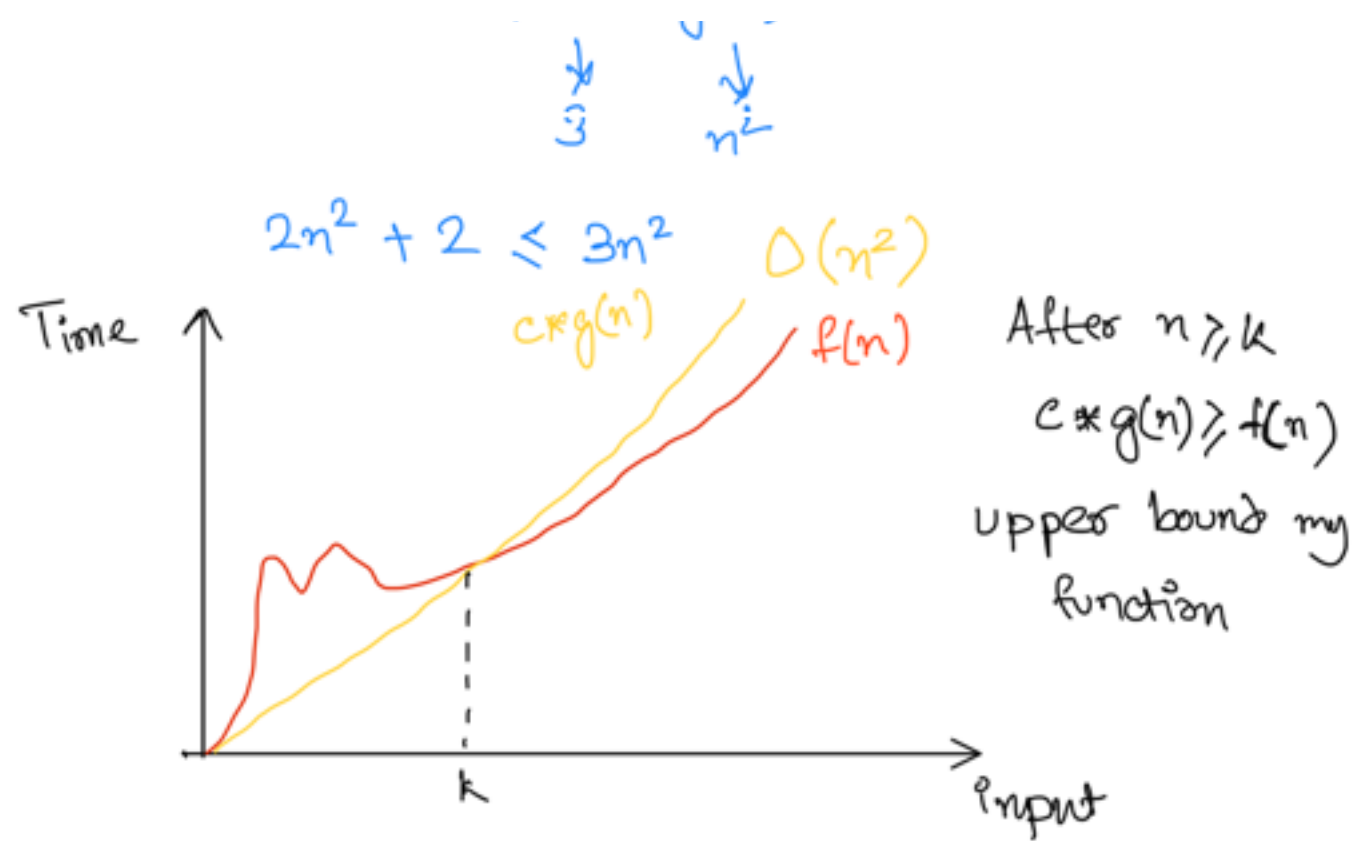
$$f(n) : 20n^2 + 15n + 2$$

A algo $f(n) \rightarrow O$ complexity

$$f(n) \leq c * g(n)$$

then time complexity = $O(g(n))$

$$f(n) = 2n^2 + 2 \leq c * g(n)$$



Big Omega (Ω)

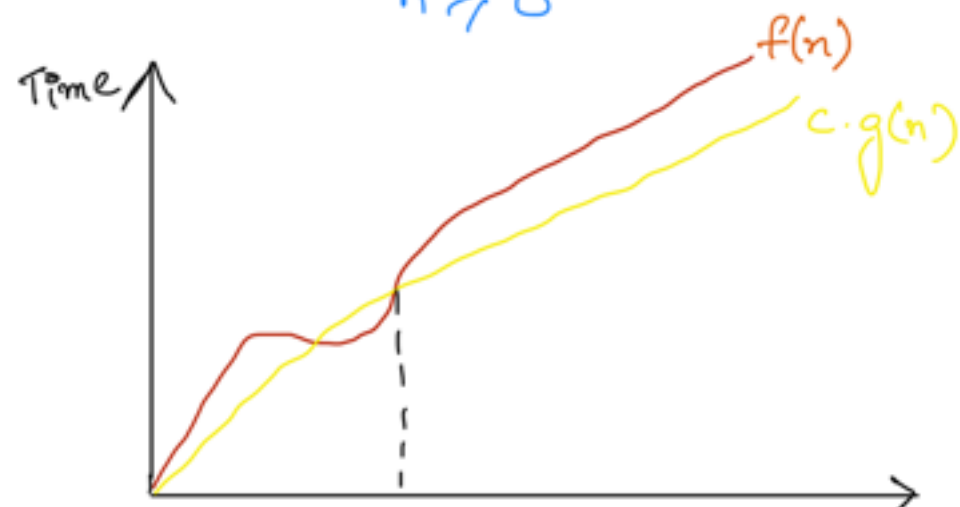
→ Best case scenario

$$f(n) = \Omega(g(n))$$

$$\text{when } f(n) \geq c \cdot g(n)$$

$$2n^2 + 2 \geq 2n^2$$

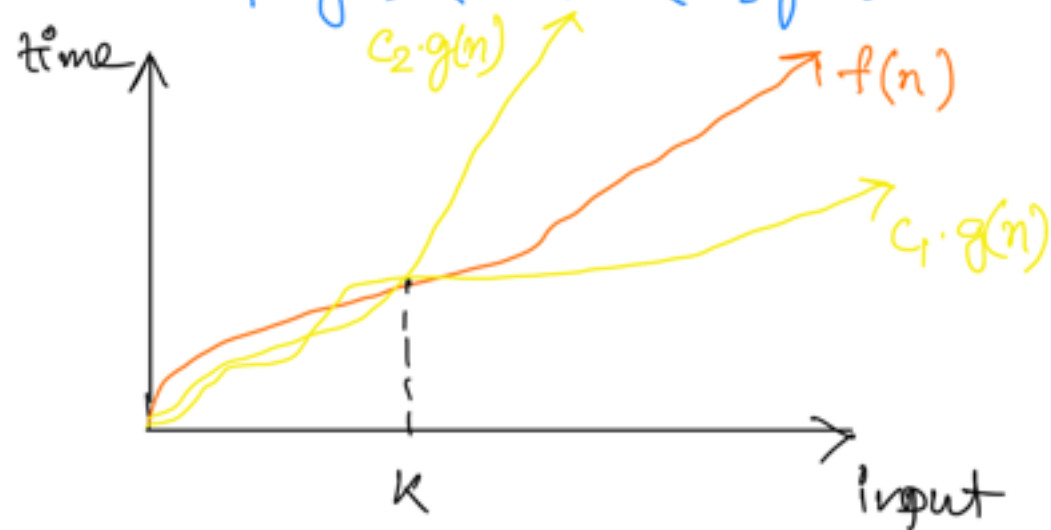
$$n \geq 0$$



Theta (Θ)

→ Average case complexity

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$



Middle of a list



$$mid = n // 2$$

$$k \cdot n^0 \leq (k+1) \cdot (n^0) \rightarrow g(n)$$

Complexity is $O(1) \rightarrow$ constant time

Largest element in an array



worst case \rightarrow largest elem at the end
 $\text{max} = a[0]$

for $i \rightarrow n$
if $a[i] > \text{max}$
 $\text{max} = a[i]$ } k operations

1	2	3	n
k	k	k		k

$$\Sigma = kn \leq \frac{c}{(k+1)} \cdot g(n) \cdot n$$

$$O(g(n)) = O(n)$$

Linear complexity

Bubble Sort Time Complexity

$n = \text{len(arr)}$

for i in $\text{range}(n)$

 for j in $\text{range}(n-i-1)$

k { if $\text{arr}[j] > \text{arr}[j+1]$:
 $\text{arr}[j], \text{arr}[j+1] = \text{arr}[j+1], \text{arr}[j]$

i	j
0	$(n-1)k \rightarrow 1^{st} \text{ elem}$
1	$(n-2)k \rightarrow 2^{nd} \text{ elem}$
2	$(n-3)k \rightarrow 3^{rd} \text{ elem}$
\vdots	
$n-1$	$k \rightarrow \text{last elem}$

$$\begin{aligned}
 \Sigma &= k(n-1) + k(n-2) + k(n-3) + \dots + k \\
 &= k[(n-1) + (n-2) + (n-3) + \dots + 1] \\
 &= k \frac{(n-1)(n-1+1)}{2} \quad \begin{array}{l} 1+2+3+\dots+n \\ = \frac{n(n+1)}{2} \end{array} \\
 &= \frac{kn(n-1)}{2} = k \frac{n^2}{2} - \cancel{\frac{kn}{2}} \rightarrow 0 \\
 &= k \frac{n^2}{2} \leq \underset{\substack{\downarrow \\ (k+1) \\ \frac{1}{2}}}{c} \cdot g(n) \Rightarrow n^2
 \end{aligned}$$

$$g(n) = n^2$$

$$O(n^2)$$

We will have complexity for bubble sort as $O(n^2)$ which is quadratic time complexity

Insertion Sort Time Complexity

def insertionSort(array):

for step in range(1, len(array)) :

key = array[step]

j = step - 1

while j > 0 and key < array[j] :

array[j+1] = array[j]

j = j - 1

array[j+1] = key

key element →



↑

Assume sorted

step

j

1

2 + 1.k

2

2 + 2.k

3

2 + 3.k

⋮

⋮

⋮

n-1

2 + (n-1).k

$$\Sigma = 2n + k[1 + 2 + \dots + n-1]$$

$$= 2n + k \frac{(n-1)(n-1+1)}{2}$$

$$= 2n + k \frac{n(n-1)}{2}$$

$$= 2n + \frac{kn^2}{2} - \frac{kn}{2}$$

$$= \frac{kn^2}{2} + n \left(2 - \frac{k}{2} \right) \xrightarrow{0}$$

$$\frac{kn^2}{2} \lesssim \underset{\left(\frac{k}{2}+1\right)}{\underset{\downarrow}{c}} \cdot \underset{\downarrow}{g(n)} \approx O(n^2)$$

Complexity $\approx O(n^2)$ i.e. quadratic

Best case (Ω) time complexity of Insertion Sort

1	2	3	4	5
---	---	---	---	---

1st elem - 0

2nd elem - k

3rd elem - k

⋮

n elem - k operation

$$\Sigma = k + k + k + \dots + (n-1)$$

$$= k(n-1) = kn - k \xrightarrow{0} \leq (k+1)n$$

$O(n)$ is the best case time complexity for insertion sort.

Selection Sort Time Complexity



key elem

for 1st elem $(n-1)k$ operations
for 2nd elem $(n-2)k$ operations
:
:
for $(n-1)$ elem $1 \cdot k$ operations

$$\begin{aligned}\Sigma &= (n-1)k + (n-2)k + \dots + k \\ &= \frac{k(n-1)(n-1+1)}{2} = \frac{kn^2}{2} - \frac{kn}{2}\end{aligned}$$

$$\frac{kn^2}{2} \leq \left(\frac{k}{2} + 1\right) n^2$$

Worst case complexity is $O(n^2)$

Best case time complexity is also same as same number of operations happening even if array completely sorted.

Time Complexity : Recursive Algorithm

for recursive algo, we use a diff. approach

— Recurrence relation method

Factorial of a number

def fact(n):

if n == 0:

return 1

} k_1

return $n * \text{fact}(n-1)$

$\underbrace{\hspace{1cm}}_{k_2}$

$$\text{fact}(n) = n * \text{fact}(n-1)$$

$$T(n) = k_1 + k_2 + T(n-1)$$

$$T(n) = k + T(n-1)$$

$$T(n-1) = k + T(n-2)$$

$$T(n-2) = k + T(n-3)$$

\vdots

$$T(0) = k$$

$$Q = P + 5$$

$$P = 5$$

$$\hline Q + P = P + 10$$

$$Q = 10$$

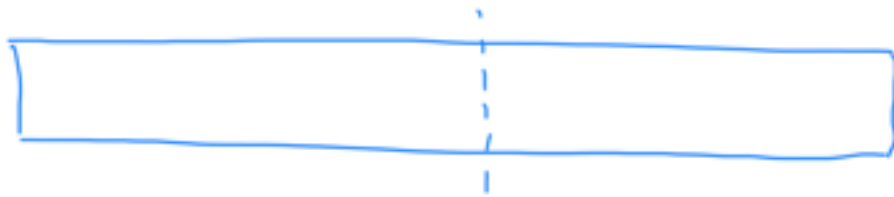
$$T(n) = k + k + k + \dots + (n+1) \text{ times}$$

$$= k(n+1) = kn + k^0$$

$$= kn \lesssim (k+1)n$$

Time complexity of factorial is $O(n)$ i.e. linear.

Binary Search Time Complexity



$$T(n) = k + T(n/2) \quad \text{Problem gets half}$$

$$T(n/2) = k + T(n/4)$$

$$T(n/4) = k + T(n/8)$$

\vdots

$$T(1) = k$$

← some constant work
(base case)

$$T(n) = k + k + k + \dots \quad ? \text{ how many times}$$

$$n \quad \frac{n}{2} \quad \frac{n}{4} \quad \frac{n}{8} \quad \dots \quad 1 \quad ?$$

$$n \quad \frac{n}{2^1} \quad \frac{n}{2^2} \quad \frac{n}{2^3} \quad \dots \quad 1 \quad ?$$

Say x times to reach 1

$$\frac{n}{2^x} = 1$$

$$\Rightarrow n = 2^x$$

$$\Rightarrow \log_2 n = \log_2 2^x$$

$$\Rightarrow \log_2 n = x \log_2 2 = x$$

$$\Rightarrow x = \log_2 n$$

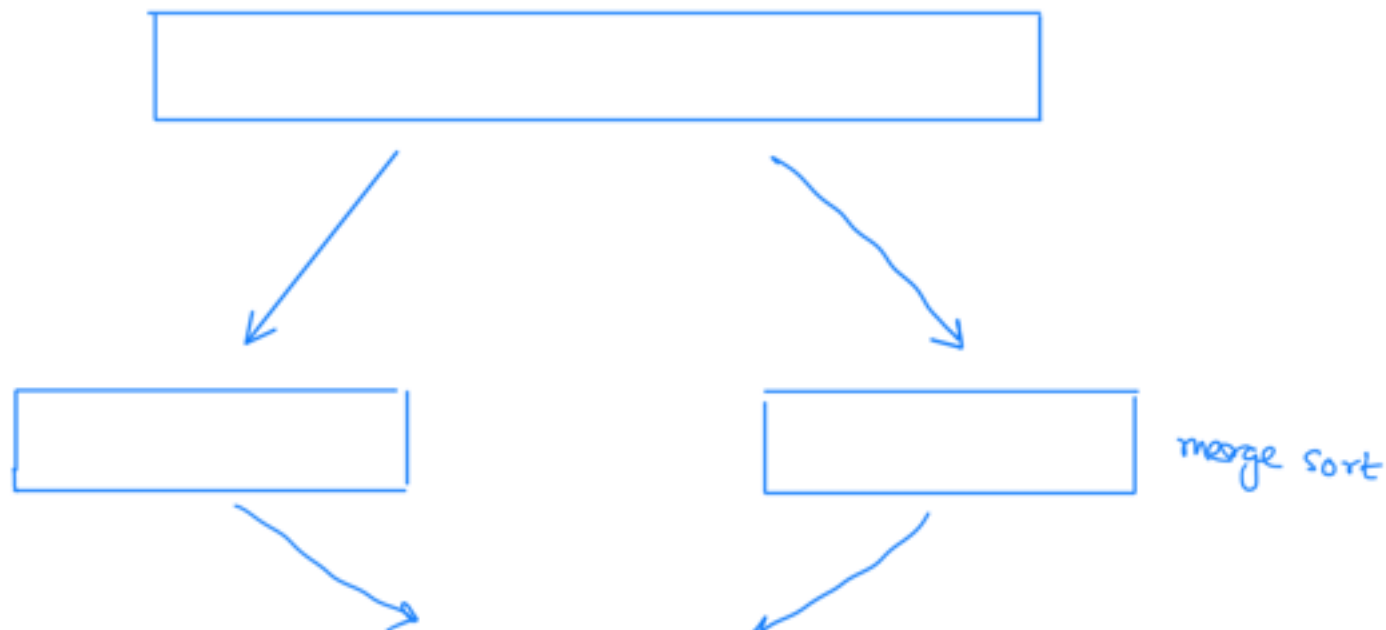
$$k + k + k + \dots \log_2 n$$

$$k \cdot \log_2 n \leq (k+1) \log_2 n$$

Time complexity of binary search is $\log_2 n$

10^6 elements : 20 comparisons

Merge Sort Time Complexity





merge

$$T(n) = k_1 + 2T(n/2) + k_2 n$$

$$= \cancel{k_1}^0 + 2T(n/2) + k_2 n$$

$$T(n) = kn + 2T(n/2)$$

$$2T(n/2) = \cancel{\frac{kn}{2}} + \textcircled{4} 2T(\frac{n}{4}) \times 2$$

$$4T(n/4) = \cancel{\frac{kn}{4}} + \textcircled{8} 2T(\frac{n}{8}) \times 4$$

$$\vdots \times 8$$

$$T(1) = kn$$

$$T(n) = kn + kn + kn + \dots \times \text{times}$$
$$n \quad \frac{n}{2} \quad \frac{n}{4} \quad \frac{n}{8} \quad \dots \quad 1$$

$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

$$\log_2 n = x \log_2 2$$

$$x = \log_2 n$$

→ n = 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864, 134217728, 268435456, 536870912, 1073741824, 2147483648, 4294967296, 8589934592, 17179869184, 34359738368, 68719476736, 137438953472, 274877906944, 549755813888, 1099511627776, 2199023255552, 4398046511104, 8796093022208, 17592186044416, 35184372088832, 70368744177664, 140737488355328, 281474976710656, 562949953421312, 1125899906842624, 2251799813685248, 4503599627370496, 9007199254740992, 18014398509481984, 36028797018963968, 72057594037927936, 144115188075855872, 288230376151711744, 576460752303423488, 1152921504606846976, 2305843009213693952, 4611686018427387904, 9223372036854775808, 18446744073709551616, 36893488147419103232, 73786976294838206464, 147573952589676412928, 295147905179352825856, 590295810358705651712, 1180591620717411303424, 2361183241434822606848, 4722366482869645213696, 9444732965739290427392, 18889465931478580854784, 37778931862957161709568, 75557863725914323419136, 151115727451828646838272, 302231454903657293676544, 604462909807314587353088, 1208925819614629174706176, 2417851639229258349412352, 4835703278458516698824704, 9671406556917033397649408, 19342813113834066795298816, 38685626227668133590597632, 77371252455336267181195264, 154742504910672534362390528, 309485009821345068724781056, 618970019642690137449562112, 1237940039285380274899124224, 2475880078570760549798248448, 4951760157141521099596496896, 9903520314283042199192993792, 19807040628566084398385987584, 39614081257132168796771975168, 79228162514264337593543950336, 158456325028528675187087900672, 316912650057057350374175801344, 633825300114114700748351602688, 1267650600228229401496703205376, 2535301200456458802993406410752, 5070602400912917605986812821504, 10141204801825835211973625643008, 20282409603651670423947251286016, 40564819207303340847894502572032, 81129638414606681695789005144064, 162259276829213363391578010288128, 324518553658426726783156020576256, 649037107316853453566312041152512, 1298074214633706907132624082305024, 2596148429267413814265248164610048, 5192296858534827628530496329220096, 10384593717069655257060992658440192, 20769187434139310514121985316880384, 41538374868278621028243970633760768, 83076749736557242056487941267521536, 166153499473114484112975882535043072, 332306998946228968225951765070086144, 664613997892457936451903530140172288, 1329227995784915872903807060280344576, 2658455991569831745807614120560689152, 5316911983139663491615228241121378304, 10633823966279326983230456482242756608, 21267647932558653966460912964485513216, 42535295865117307932921825928971026432, 85070591730234615865843651857942052864, 170141183460469231731687303715884105728, 340282366920938463463374607431768211456, 680564733841876926926749214863536422912, 1361129467683753853853498429727072845824, 2722258935367507707706996859454145691648, 5444517870735015415413993718908291383296, 10889035741470030830827987437816582766592, 21778071482940061661655974875633165533184, 43556142965880123323311949751266331066368, 87112285931760246646623899502532662132736, 174224571863520493293247799005065324265472, 348449143727040986586495598010130648530944, 696898287454081973172991196020261297061888, 1393796574908163946345982392040522594123776, 2787593149816327892691964784081045188247552, 5575186299632655785383929568162090376495104, 11150372599265311570767859136324180752990208, 22300745198530623141535718272648361505980416, 44601490397061246283071436545296723011960832, 89202980794122492566142873090593446023921664, 178405961588244985132285746181186892047843328, 356811923176489970264571492362373784095686656, 713623846352979940529142984724747568191373312, 1427247692705959881058285969449495136382746624, 2854495385411919762116571938898990272765493248, 5708990770823839524233143877797980545530986496, 11417981541647679048466287755595961091061972992, 22835963083295358096932575511191922182123945984, 45671926166590716193865151022383844364247891968, 91343852333181432387730302044767688728495783936, 182687704666362864775460604089535377456991567872, 365375409332725729550921208179070754913983135744, 730750818665451459101842416358141509827966271488, 1461501637330902918203684832716283019655932542976, 2923003274661805836407369665432566039311865085952, 5846006549323611672814739330865132078623730171904, 11692013098647223345629478661730264157247460343808, 23384026197294446691258957323460528314494920687616, 46768052394588893382517914646921056628989841375232, 93536104789177786765035829293842113257979682750464, 187072209578355573530071658587684226515959365500928, 374144419156711147060143317175368453031918731001856, 748288838313422294120286634350736906063837462003712, 1496577676626844588240573268701473812127674924007424, 2993155353253689176481146537402947624255349848014848, 5986310706507378352962293074805895248510699696029696, 11972621413014756705924586149611790497021399392059392, 23945242826029513411849172299223580994042798784118784, 47890485652059026823698344598447161988085597568237568, 95780971304118053647396689196894323976171195136475136, 191561942608236107294793378393788647952342390272950272, 383123885216472214589586756787577295904684780545900544, 766247770432944429179173513575154591809369561091801088, 1532495540865888858358347027150309183618739122183602176, 3064991081731777716716694054300618367237478244367204352, 6129982163463555433433388108601236734474956488734408704, 12259964326927110866866776217202473468949912977468817408, 24519928653854221733733552434404946937899825954937634816, 49039857307708443467467104868809893875799651909875269632, 98079714615416886934934209737619787751599303819750539264, 196159429230833773869868419475239575503198607639501078528, 392318858461667547739736838950479151006397215279002157056, 784637716923335095479473677900958302012794430558004314112, 1569275433846670190958947355801916604025588861116008628224, 3138550867693340381917894711603833208051177722232017256448, 6277101735386680763835789423207666416102355444464034512896, 12554203470773361527671578846415332832204710888928069025792, 25108406941546723055343157692830665664409421777856138051584, 50216813883093446110686315385661331328818843555712276103168, 100433627766186892221372630771322662657637687111424552206336, 200867255532373784442745261542645325315275374222849104412672, 401734511064747568885490523085290650630550748445698208825344, 803469022129495137770981046170581301261101496891396417650688, 1606938044258990275541962092341162602522202993782792835301376, 3213876088517980551083924184682325205044405987565585670602752, 6427752177035961102167848369364650410088811975131171341205504, 12855504354071922204335696738729300820177623950262342682411008, 25711008708143844408671393477458601640355247900524685364822016, 51422017416287688817342786954917203280710495801049370729644032, 102844034832575377634685573909834406561420991602098741459288064, 205688069665150755269371147819668813122841983204197482918576128, 411376139330301510538742295639337626245683966408394965837152256, 822752278660603021077484591278675252491367932816789931674304512, 1645504557321206042154969182557350504982735865633579863348609024, 3291009114642412084309938365114701009965471731267159726697218048, 6582018229284824168619876730229402019930943462534319453394436096, 13164036458569648337239753460458804039861886925068638906788872192, 26328072917139296674479506920917608079723773850137277813577744384, 52656145834278593348959013841835216159447547700274555627155488768, 105312291668557186697918027683670432318895095400549111254310977536, 210624583337114373395836055367340864637790190801098222508621955072, 421249166674228746791672110734681729275580381602196445017243910144, 842498333348457493583344221469363458551160763204392890034487820288, 1684996666696914987166688442938726917102321526408785780068975640576, 3369993333393829974333376885877453834204643052817571560137951281152, 6739986666787659948666753771754907668409286105635143120275902562304, 13479973333575319897333507543509815336818572211270286240551805124608, 26959946667150639794667015087019630673637144422540572481103610249216, 53919893334301279589334030174039261347274288845081144962207220498432, 107839786668602559178668060348078522694548577690162289924414440996864, 215679573337205118357336120696157045389097155380324579848828881993728, 431359146674410236714672241392314090778194310760649159697657763987456, 862718293348820473429344482784628181556388621521298319395315527974912, 1725436586697640946858688965569256363112777243042596638790631055949824, 3450873173395281893717377931138512726225554486085193277581262111899648, 6901746346790563787434755862277025452451108972170386555162524223799296, 13803492693581127574869511724554050904902217944340773110325048447598592, 27606985387162255149739023449108101809804435888681546220650096895197184, 55213970774324510299478046898216203619608871777363092441300193790394368, 110427941548649020598956093796432407239217743554726184882600387580788736, 220855883097298041197912187592864814478435487109452369765200775161577472, 441711766194596082395824375185729628956870974218904739530401550323154944, 883423532389192164791648750371459257913741948437809479060803100646309888, 1766847064778384329583297500742918515827483896875618958121606201292619776, 3533694129556768659166595001485837031654967793751237916243212402585239552, 7067388259113537318333190002971674063309935587502475832486424805170479104, 14134776518227074636666380005943348126619871175004951664972849610340958208, 28269553036454149273332760011886696253239742350009903329945699220681916416, 56539106072908298546665520023773392506479484700019806659891398441363832832, 113078212145816597093331040047546785012958969400039613319782796882727665664, 226156424291633194186662080095093570025917938800079226639565593765455331328, 452312848583266388373324160190187140051835877600158453279131187530910662656, 904625697166532776746648320380374280103671755200316906558262375061821325312, 1809251394333065553493296640760748560207343510400633813116524750123642650624, 3618502788666131106986593281521497120414687020801267626233049500247285301248, 7237005577332262213973186563042994240829374041602535252466099000494570602496, 14474011154664524427946373126085988481658748083205070504932198000989141204992, 28948022309329048855892746252171976963317496166410141009864396001978282409984, 57896044618658097711785492504343953926634992332820282019728792003956564819968, 115792089237316195423570985008687907853269984665640564039457584007913129639936, 231584178474632390847141970017375815706539969331281128078915168015826259279872, 463168356949264781694283940034751631413079938662562256157830336031652518559744, 926336713898529563388567880069503262826159877325124512315660672063305037119488, 1852673427797059126777135760139006525652319754650249024631321344126610074238976, 3705346855594118253554271520278013051304639509300498049262642688253220148477952, 7410693711188236507108543040556026102609279018600996098525285376506440296955904, 14821387422376473014217086081112052205218558037201992197050570753012880593911808, 29642774844752946028434172162224104410437116074403984394101141506025761187823616, 59285549689505892056868344324448208820874232148807968788202283012051522375647232, 118571099379011784113736688648896417641748464297615937576404566024103044751294464, 237142198758023568227473377297792835283496928595231875152809132048206089502588928, 474284397516047136454946754595585670566993857190463750305618264096412179005177856, 94856879503209427290989350919117134113398771438092750061123652

$$\rightarrow n \log_2 n \leq (k+1) n \log_2 n$$

Time complexity = $O(n \log n)$

(best, avg, worst case)

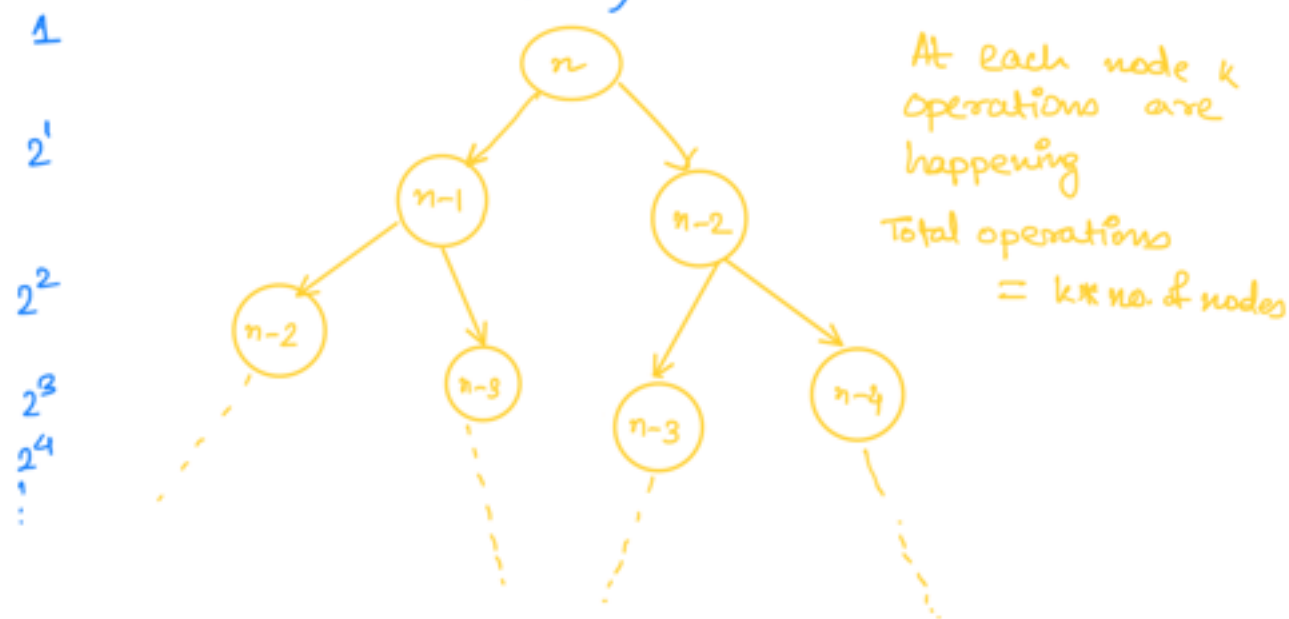
Fibonacci Number - Time Complexity

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
```

} k operations

return fib(n-1) + fib(n-2)

$$T(n) = k + T(n-1) + T(n-2)$$



* we are assuming that each subtree will in worst case end at n depth / bottom most depth

∴ No. of Nodes

$$= 1 + 2 + 4 + 8 + 16 + \dots$$

$$= 1 + 2^1 + 2^2 + 2^3 + 2^4 + \dots$$

$$= \frac{2(2^n - 1)}{2 - 1} \quad [\text{Sum of } n \text{ terms in G.P.}]$$

$$= 2^{2^1} - 1$$

$$\begin{aligned} \therefore \text{No. of operations} &= k * \text{No. of nodes} \\ &= k * 2(2^n - 1) \\ &= 2^{n+1}k - 2k \end{aligned}$$

$$\begin{aligned} T(n) &= k 2^{n+1} \\ &= 2k 2^n \leq 3k 2^n \end{aligned}$$

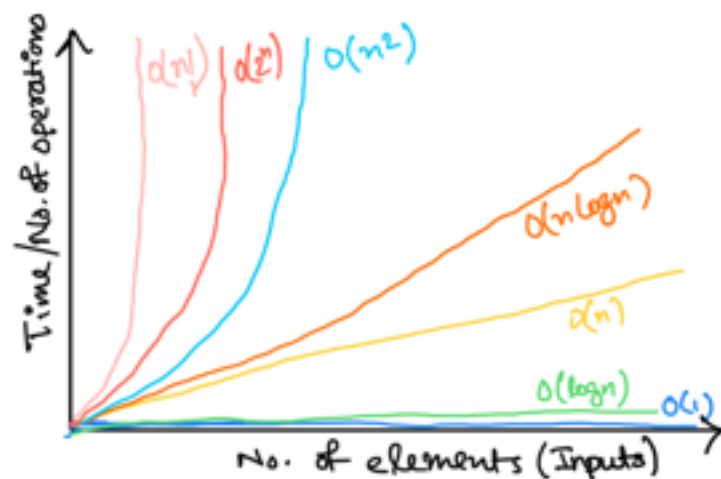
\therefore Complexity is $O(2^n)$ i.e. exponential

Space Complexity

Space complexity is space required as a function $f(n)$ of input size

$$\begin{aligned} \text{Space complexity} &= \text{Input space} \\ &+ \\ &\text{Auxiliary space} \end{aligned}$$

Visualizing Time Complexities



While doing questions on Leetcode, OA, codechef, we face time limit exceeded error.

Brute force normally not accepted

⇒ No of operations allowed per second $\sim 10^8$ ops/sec

In problems on internet, the same is also provided in question defn
e.g. Constraints : $1 \leq \text{nums}, \text{length} \leq 5 \times 10^4$

if we apply, algorithms like bubble sort for max length array, no. of operations = $O(n^2) = (5 \times 10^4)^2 = 25 \times 10^8 > 10^8$
Hence will give time limit exceeded error.

How to determine the solution by looking at constraint?

<u>Constraint</u>	<u>Worst Time Complexity</u>	<u>Algo</u>
$n \leq 12$	$O(n!)$	Backtracking Recursion
$n \leq 25$	$O(2^n)$	Bit manipulation, Recursion/Backtracking
$n \leq 100$	$O(n^4)$	DP
$n \leq 500$	$O(n^3)$	DP
$n \leq 10^4$	$O(n^2)$	DP, graph, tree
$n \leq 10^6$	$O(n \log n)$	Sorting, Divide & Conquer
$n \leq 10^8$	$O(n)$	Mathematical Greedy
$n > 10^8$	$O(1)/O(\log n)$	Mathematical, Greedy

One more way to remember:

Whatever algo we choose, if we plug in n
we should get around $\sim 10^8$

Above is a hint to...

... we help in remembering.

Space Complexity

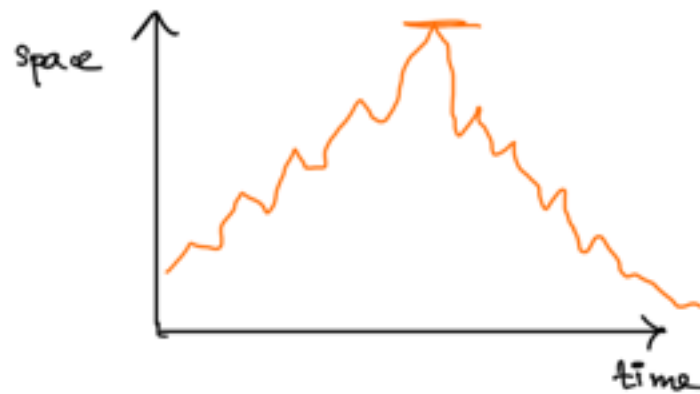
Space complexity is space required as a function $f(n)$ of input size

$$\text{Space complexity} = \text{input space} + \text{auxiliary space}$$

* We do not consider variables created as extra space
eg. $\rightarrow i, j, k$ etc

Merge

sorted array $= []$ dependent on input size



We are only concerned about the maximum space that is required during the entire algorithm

8gb
install 2gb } +
install 2gb } +
delete 4gb } -
Install 6gb } +

$n = 100000$

while $(i \leq n)$: space \uparrow

$$a = 5$$

$$a \neq 1$$



Space complexity of algorithms

Insertion Sort : $O(n^2) = O(1)$

Bubble Sort : $O(n^2) = O(1)$

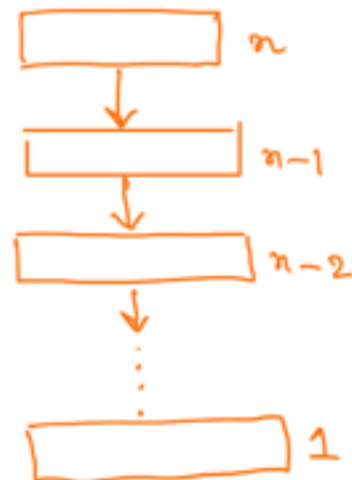
Selection Sort : $O(n^2) = O(1)$

No such variables created which takes space depending on input size.

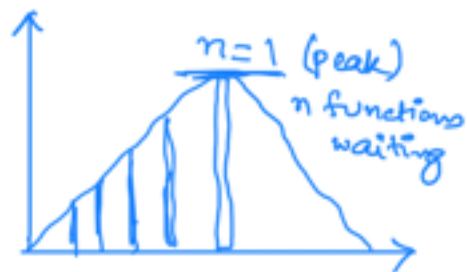
Space Complexity - Recursive Algorithm

```
def fact(n):
    if (n <= 1):
        return 1
    return fact(n-1)
```

space (k)



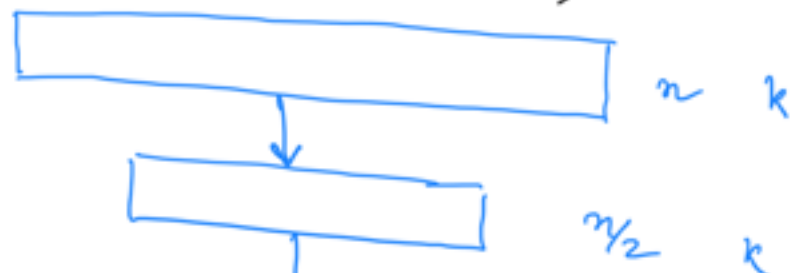
Each function takes k space



Recursion is not free. The function waiting for answers, takes up some space

kn is the maximum space taken
 \therefore Space complexity $= O(n)$

Binary Search (Recursion)



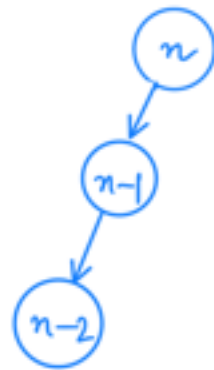
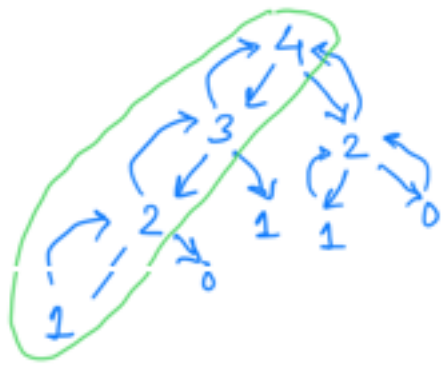


$k * (\text{no. of functions}) \Rightarrow k \cdot \log_2 n$

We will have $\log_2 n$ functions waiting

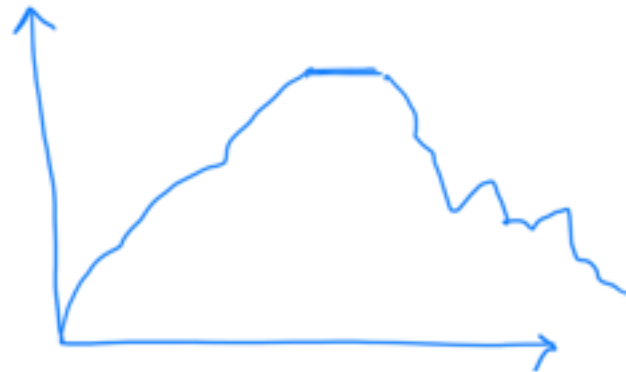
Fibonacci Number : Space Complexity

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



maximum number of calls in memory = n

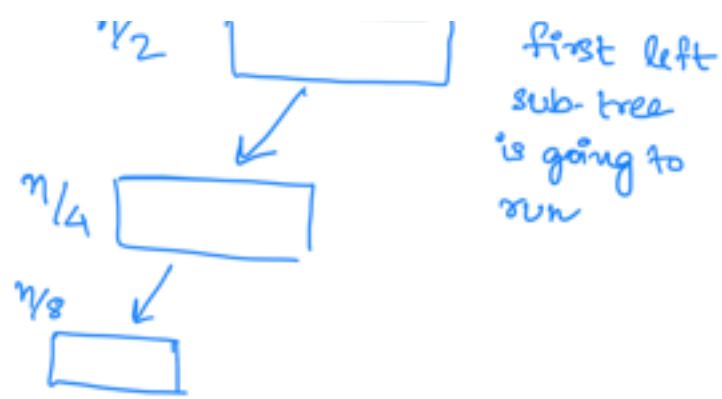
so space complexity = kn



Merge Sort - Space Complexity



$\log_2 n$ functions at max waiting in the memory



def merge sort(arr)

$\log_2 n$ { merge sort (left)
merge sort (right)

n { merge (left, right)

Space complexity
 $= O(\log_2 n + n)$

for a very large n

$n \gg \log_2 n$
major term

My major term is n

Space complexity $\rightarrow kn$

$O(n)$ is the space complexity of Merge Sort.



bigoocheatsheet.com

