Python DSA Notes

Built in Data structures

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

List, Tupple 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	List
Data Type	Homogeneous (same type)	Heterogeneous (different type)
Flexibility	Fixed size	Dynamic

		(can grow & shrink)
Performance	More memory efficient for large data	Slower for large data
Usage in python	Use array or numpy	Built-in

Searching and Sorting

Linear search

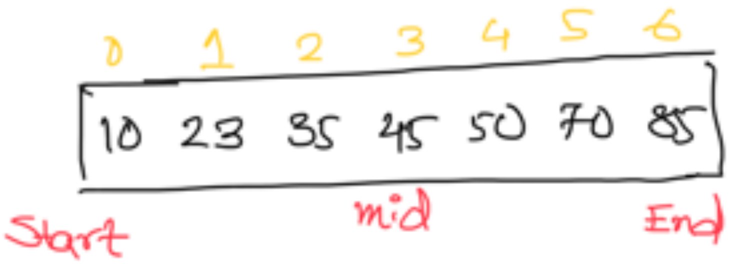
arr, target

return the target index if present else -1

implementation - Run a loop and compare.

Binary Search

arr, target
↓
* sorted



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 → mid = 3

iteration ⇒ mid – 3

arr[mid] = 45 < 50

2nd iteration ⇒ start – 4, mid – 5, end – 6

arr[mid] = 70 > 50

3rd iteration ⇒ start – 4, end – 4, mid – 4

arr[mid] = 50 == 50

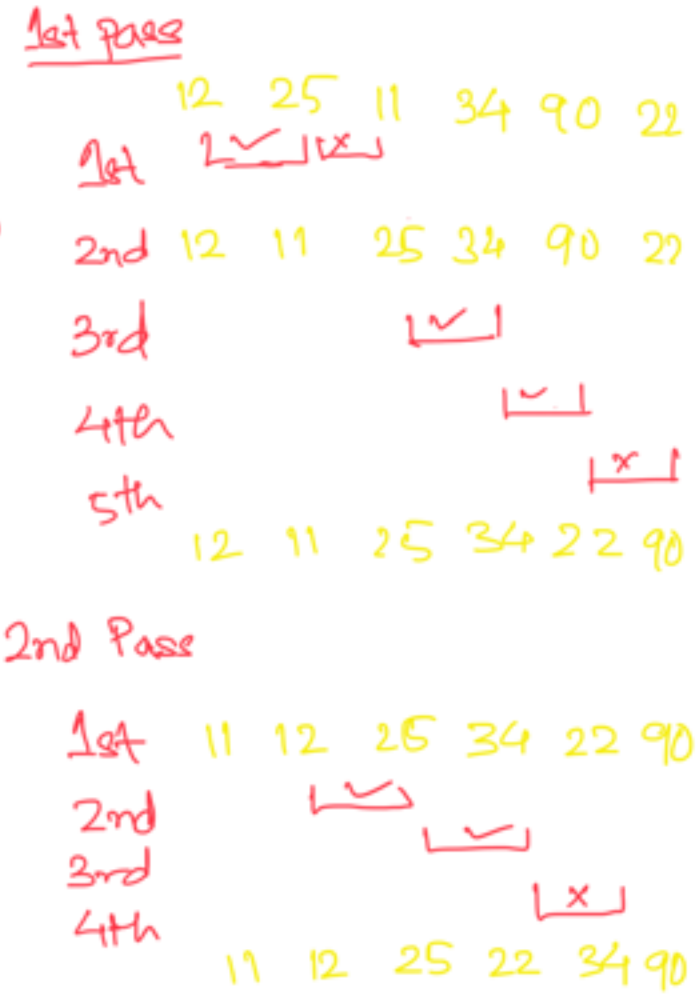
* Only works when array is sorted

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

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



Selection Sort

Select

In each pass we see

12 25 11 34 90 22

the array and select the minimum

we swap that to the right position

11 12 22 25 34 90

minimum select

n-1 pas

Pass 1 11 25 12 34 90 22

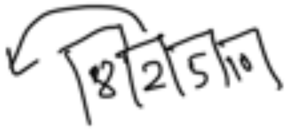
Pass 2 11 12 25 34 90 22

Pass 3

Insertion Sort

12 25 11 34 90 22

0 1 2 3 4 5



First round not required as 1st element always sorted

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)

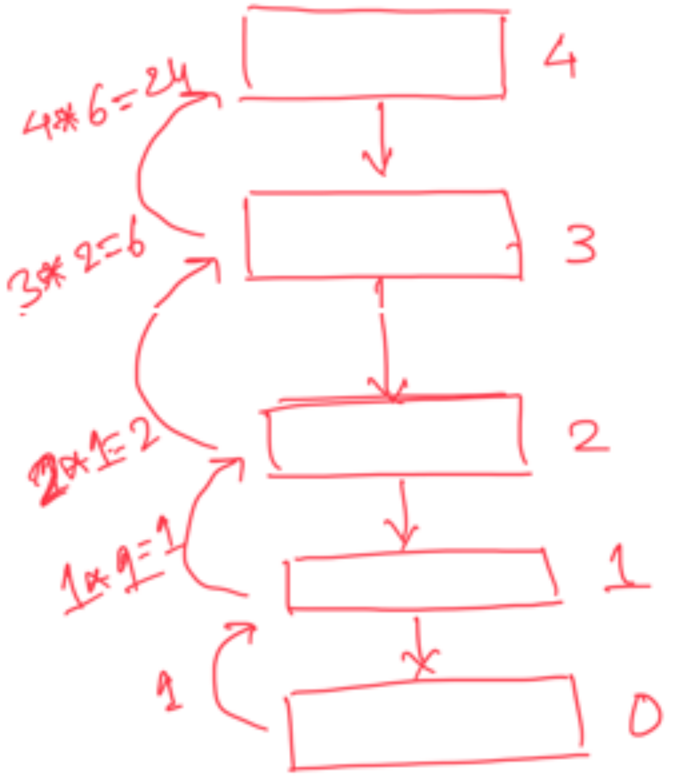
Factorial of a number

① Base case $0! = 1$

② Recursion relation: $fact(n) = n * fact(n-1)$

③ our work: multiply n with small ans

Recursion Tree (4!)



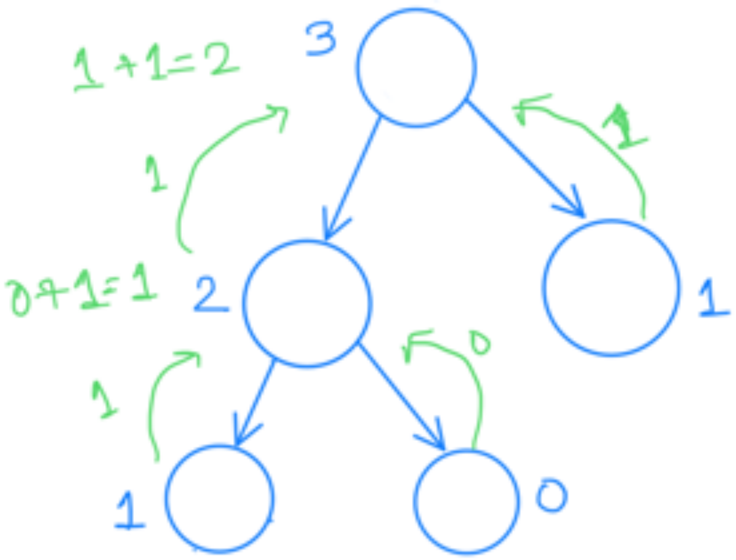
Fibonacci series

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

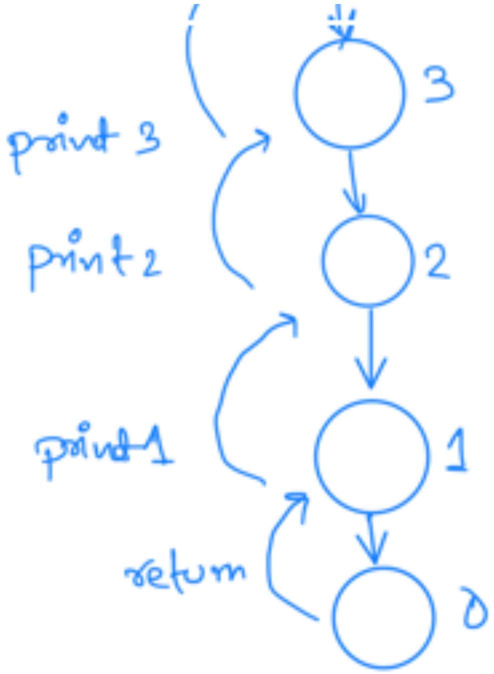
e.g. → Print from 1 to N

Base case: if n==0:
return

Recursion call: print1toN(n-1)

Your Work: print(n)

Recursion Tree (n=4)



Tail Recursion

① Define base case

② Do your work

③ Do recursion call

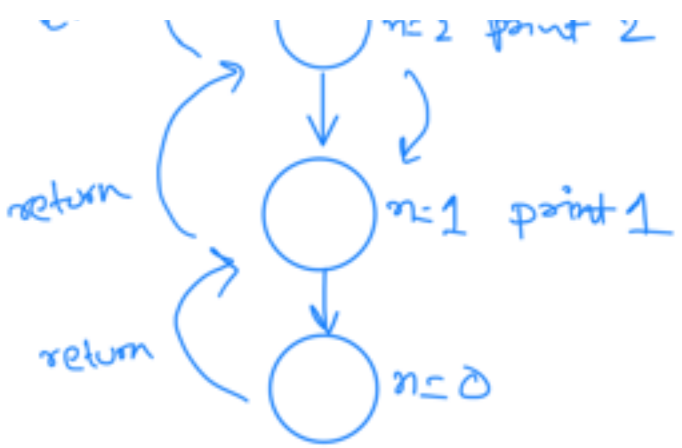
e.g. → Print N to 1

Base case : if n==0 :
return

Your work : print n

Recursion call : printNto1(n-1)

Recursion Tree (n=3)



Recursion & Arrays

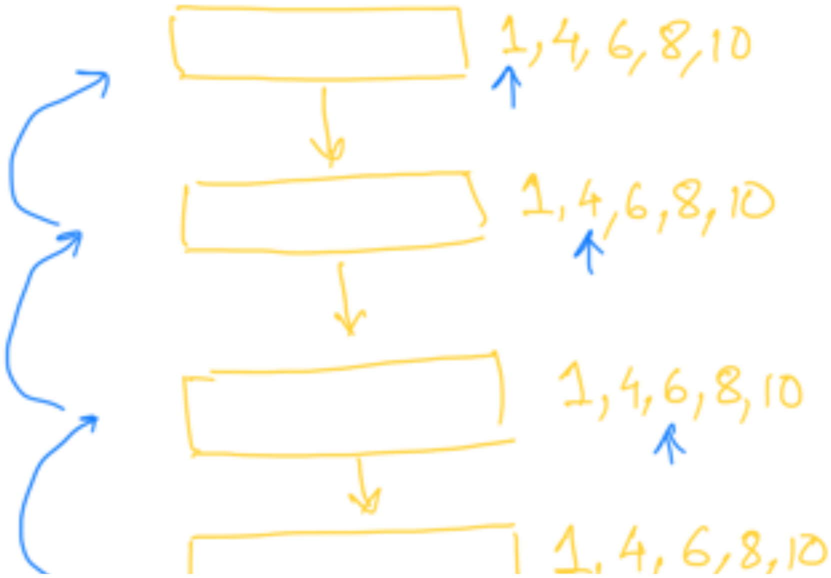
① Check if Array is sorted

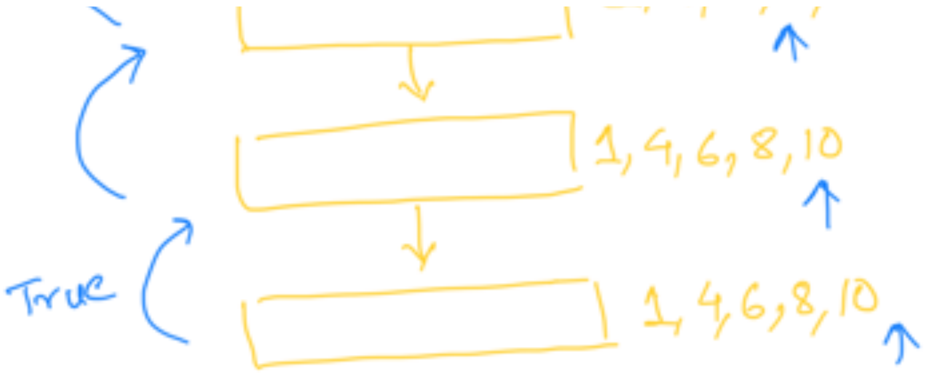
Base case : if len(Arr) == 0 : or if len(Arr) == index :
return True return True

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

Recursion call : check_sorted_array(Arr[1:])
or
check_sorted_array(Arr, 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 len[Array] == index :
return -1

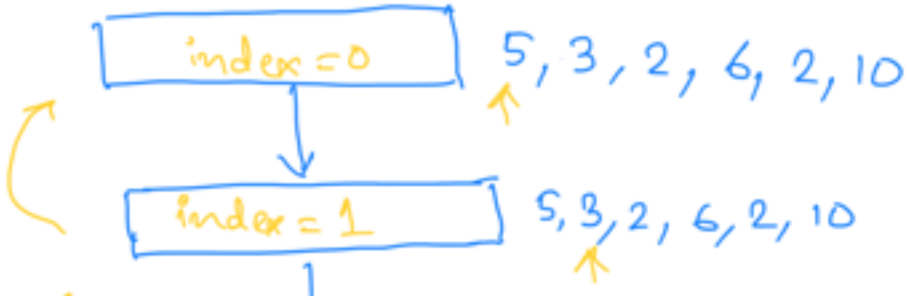
My work : if Array[index] == elem:
return index

Recursive call : get_first_index (Arr, index+1)

Recursion Tree

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

elem = 2



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 len(Arr) == index:
    return -1
```

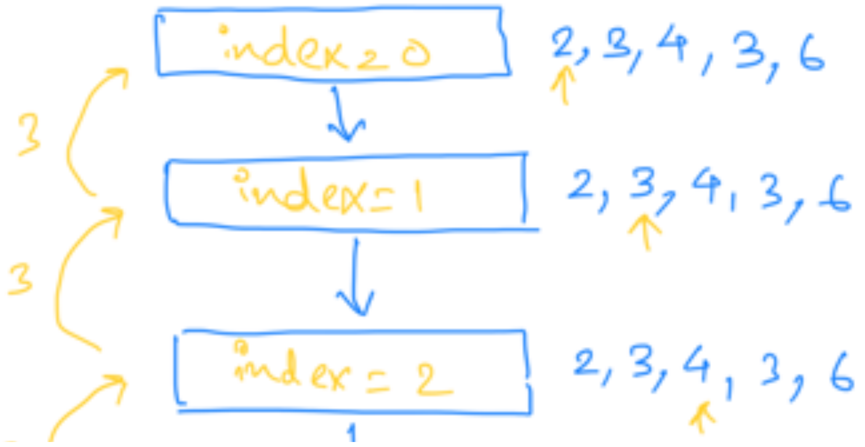
Recursion call: st_index = get_last_index(Arr, index+1)

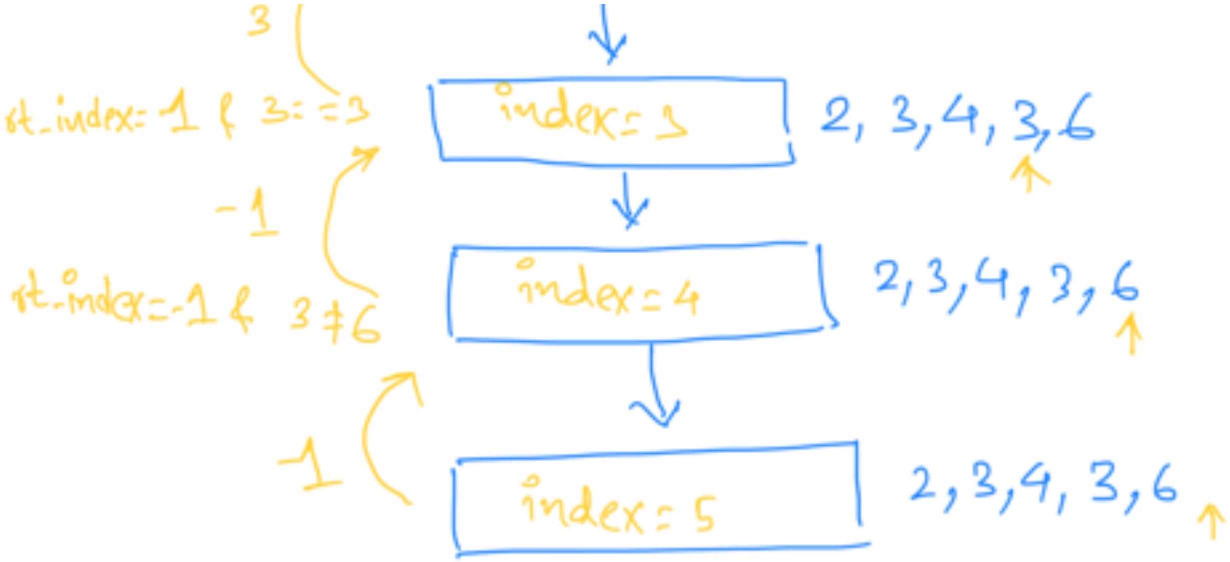
My work:
```
if st_index == -1 and Arr[index] == elem:
    return index
return st_index
```

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

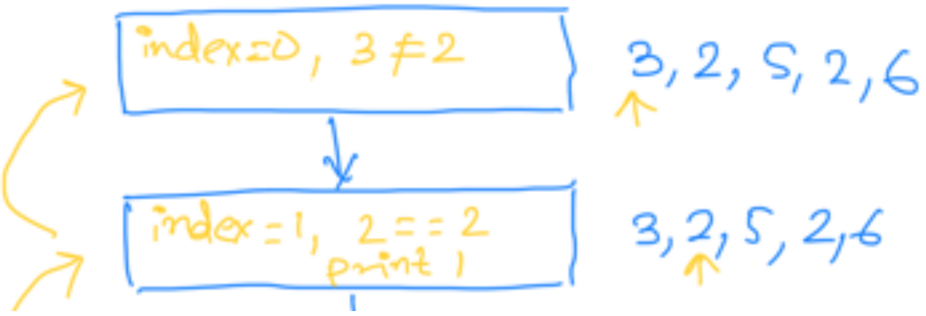
indices = 1 3

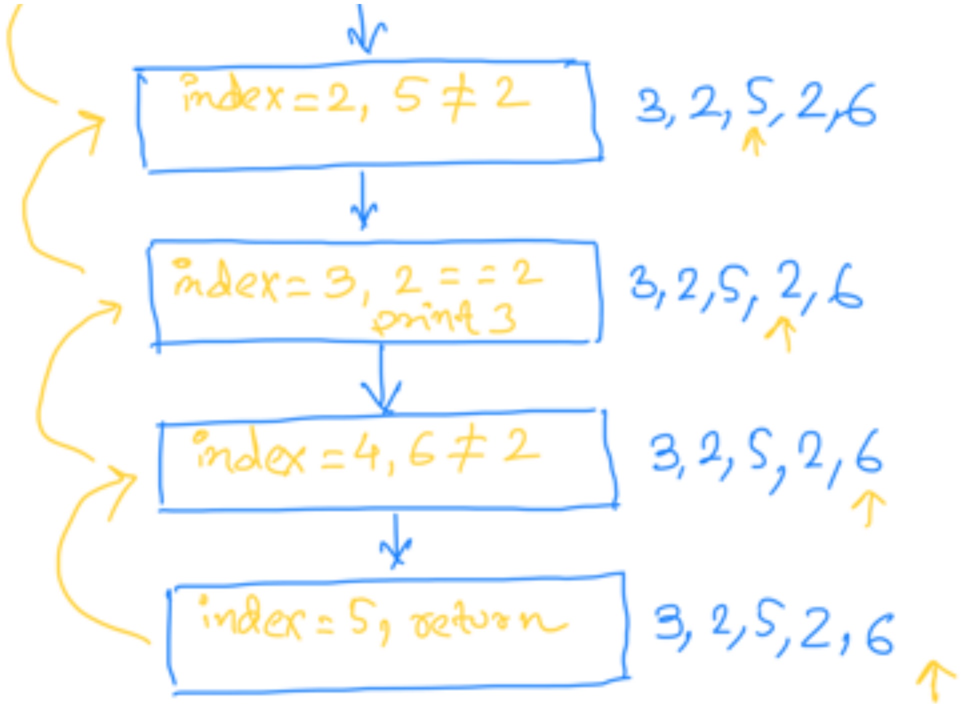
Base case: if len(Arr) == index:
return

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

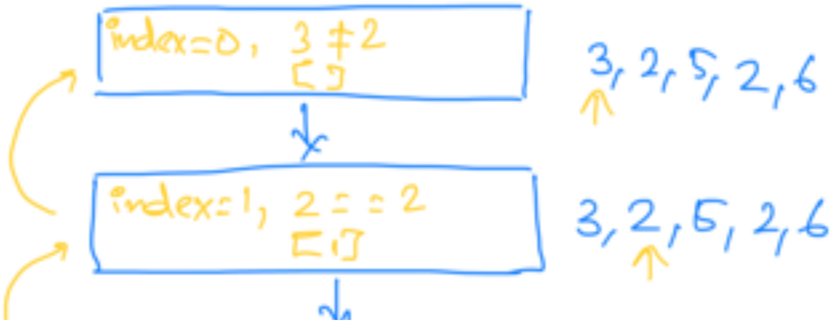
Arr = [3, 2, 5, 2, 6]
        0 1 2 3 4
elem = 2
ans = [1, 3]

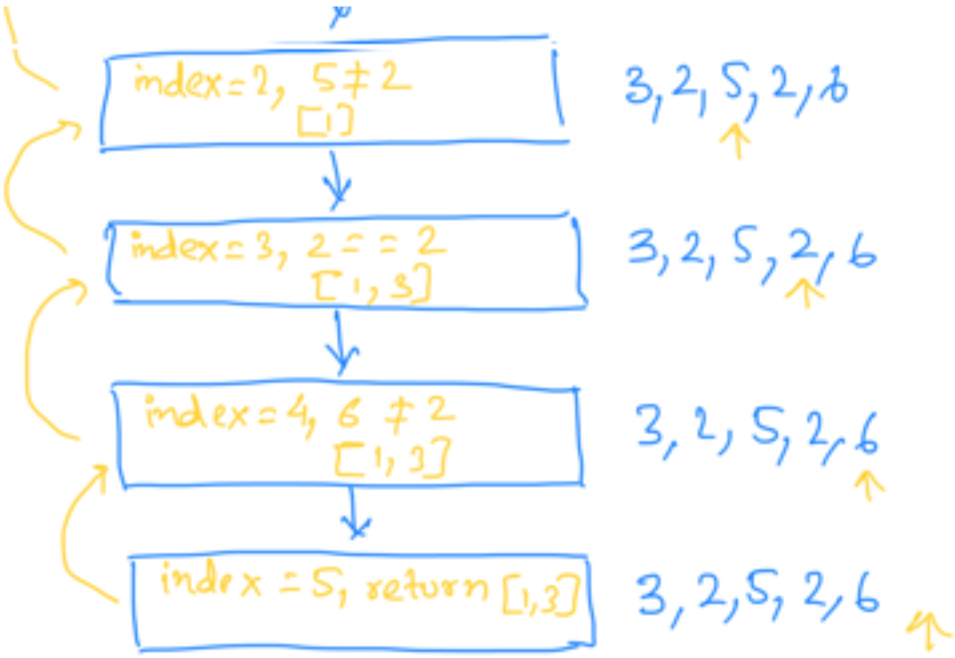
Base Case : if len(Arr) == index :
    return lst

My work : if Arr[index] == elem :
    lst.append(index)

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

Recursion Tree





Return indices of element in a new list

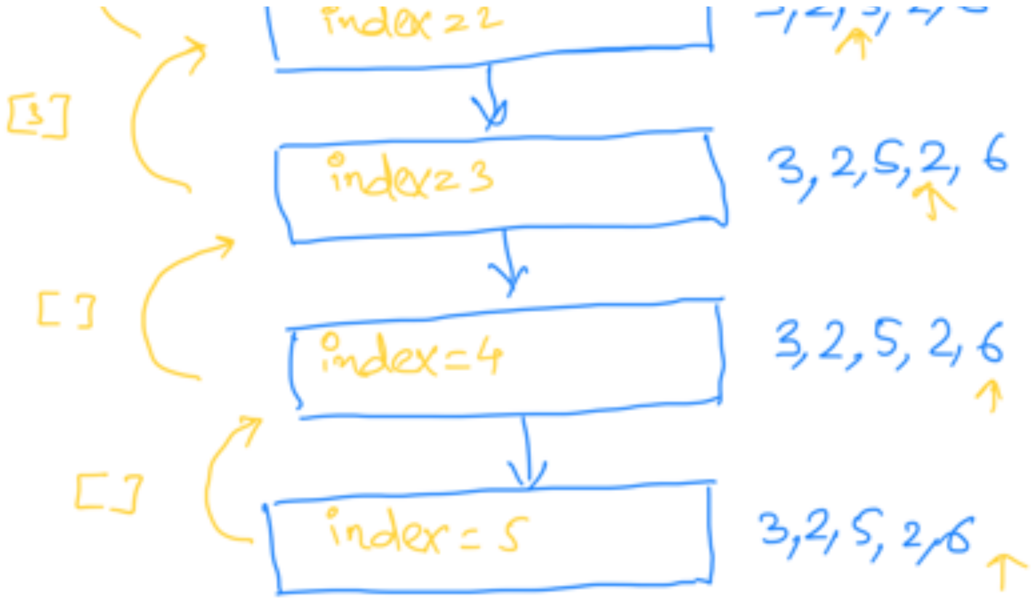
Arr = [3, 2, 5, 2, 6]
0 1 2 3 4
elem = 2
lst = [1,3]

Base case : if len(Arr) == index:
return [ ]

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

My work : if Arr[index] == 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 len(Arr) == 0 :

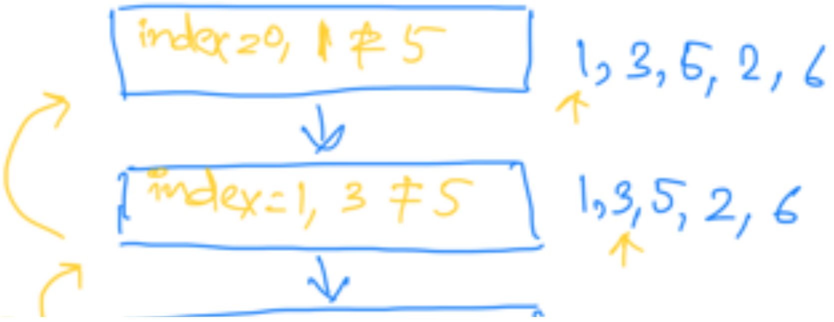
return -1

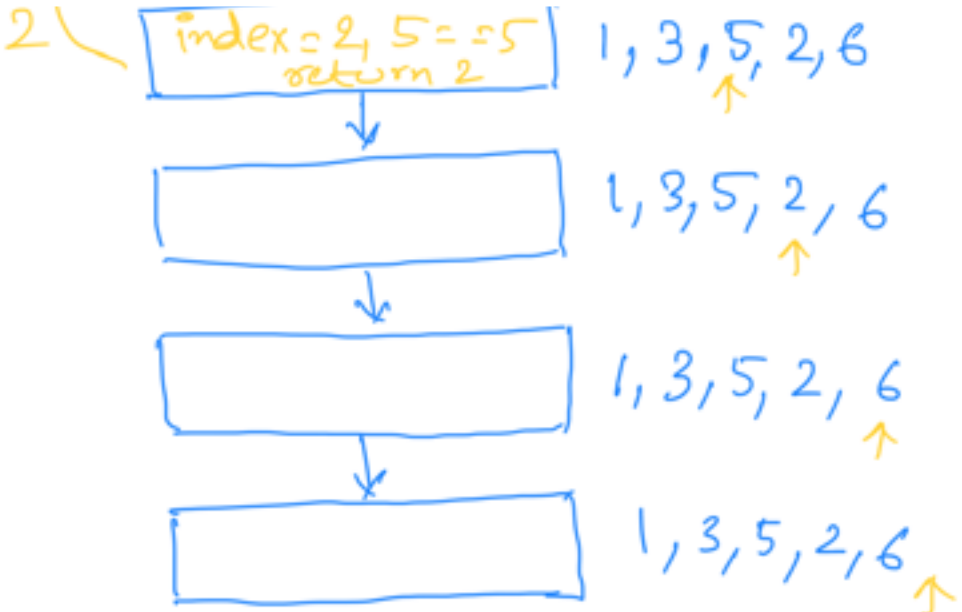
My work : if Arr[index] == index :

return index

Recursion call : Linear_search(Arr, index+1)

Recursion Tree





Binary Search.

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

target = 5

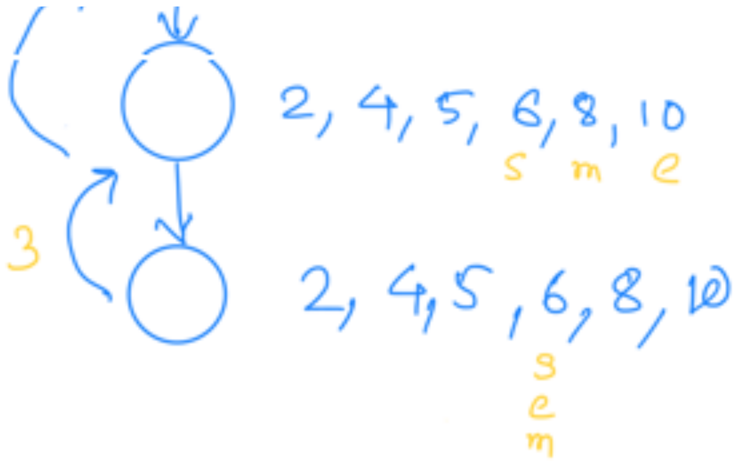
Base Case: if start > end:
return -1

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)

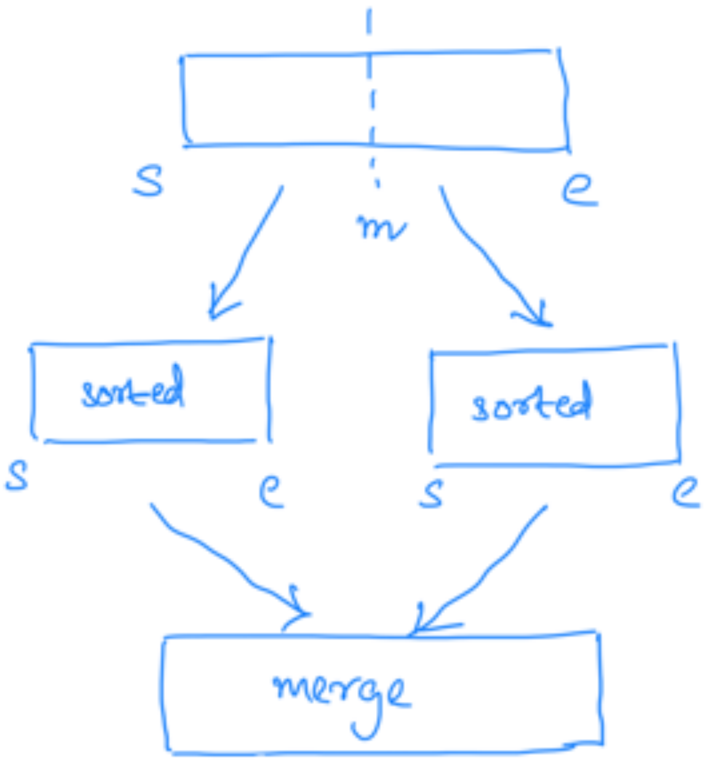
Recursion Tree

[2, 4, 5, 6, 8, 10] → 6



Merge Sort

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



Base Case : if start ≥ end :
return

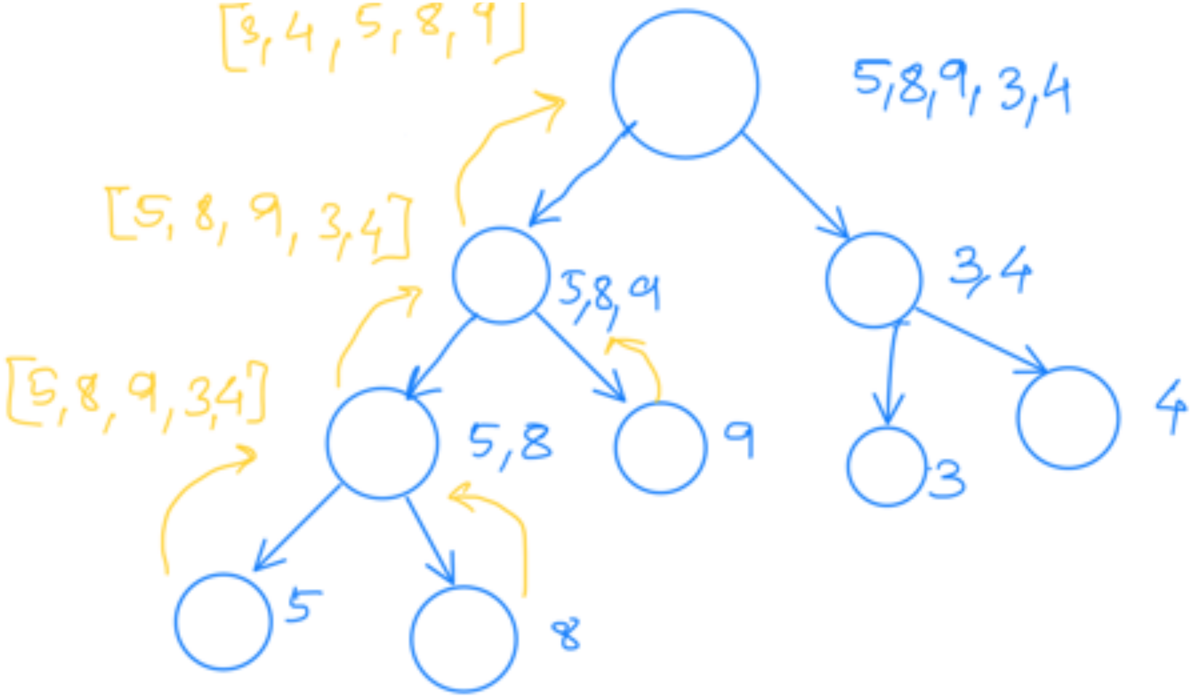
Recursion Call : mid = start + (end - start)//2
merge_sort (Arr, start, mid)
merge_ sort (Arr, mid+1, end)

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

```
    i= start         new_lst=[ ]
    j= mid+1
while  i<=mid and j<=end :
      if  Arr[i] < Arr[j] :
            new_lst.append(Arr[i])
            i=i+1
      elif  Arr[i] > Arr[j] :
            new_lst, append (Arr[j])
            j=j+1
      elif  Arr[i] == Arr[j] :
            new_lst.append (Arr[i])
            new_lst. append (Arr [j])
              i= i+1
              j= j+1
 while  i<=mid :
      new_lst. append( Arr [i])
      i=i+1
 while  j <=end :
      new_lst. append(Arr[j])
      j =j+1
```

Remaining elements from list

```
existingListIndex = start
newListIndex = 0
while  existingListIndex <= end :
        Arr [existingListIndex] = new_lst[newListIndex]
        existingListIndex+= 1
        newListIndex += 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 >= end :
return

My work : partition(Arr, start, end)
pivotElem = Arr[end]
pivotIndex = start

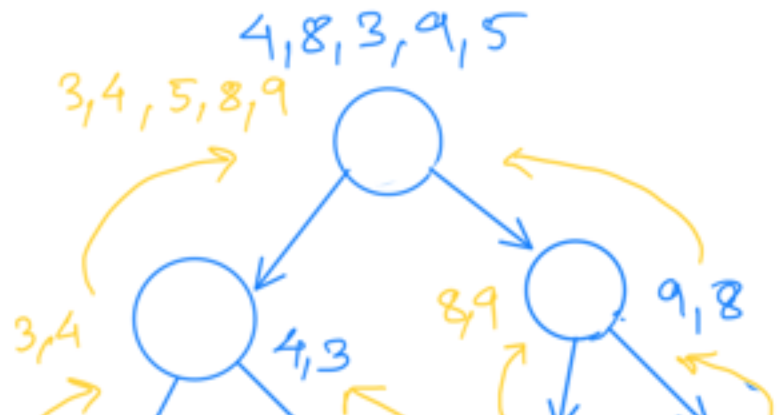
```
  i = start
while i < end:
    if Arr[i] < pivotElem:
        pivotIndex+ = 1
    i+ = 1

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



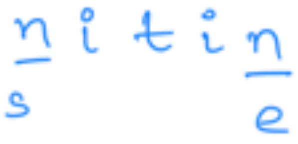
Recursion and Strings

Palindrome Check with Recursion

nitin → palindrome

abc → not palindrome

malayalam → palindrome

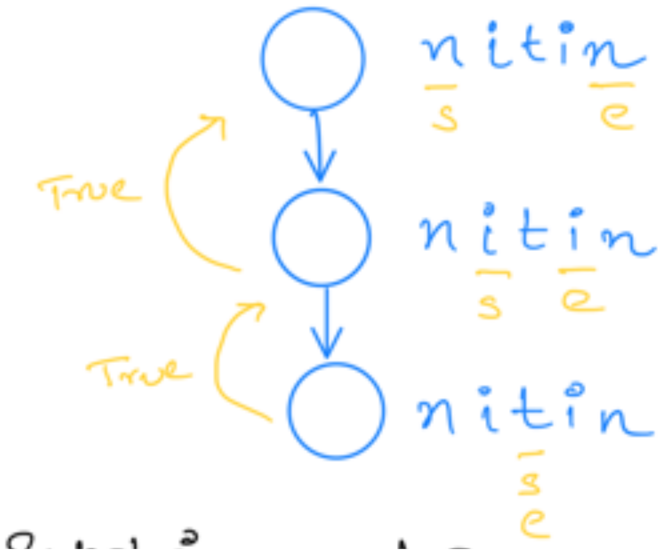


Base Case : if start $\geq$ end:
return True

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

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

Recursion Tree



Strings, Substrings and Subsequences

string = 'abc'

substring = ordering to be maintained and it should be continuous

a^1 b^2 c^3

ab^4 bc^5 ~~ac~~

abc^6

$$\frac{n(n+1)}{2} = \frac{3(3+1)}{2} = 6$$

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

a b c

ab bc ac

abc ϕ

2^n

either take it or not take it

Return Subsequences of a given string

abc → a b c

ab bc ac

abc ϕ

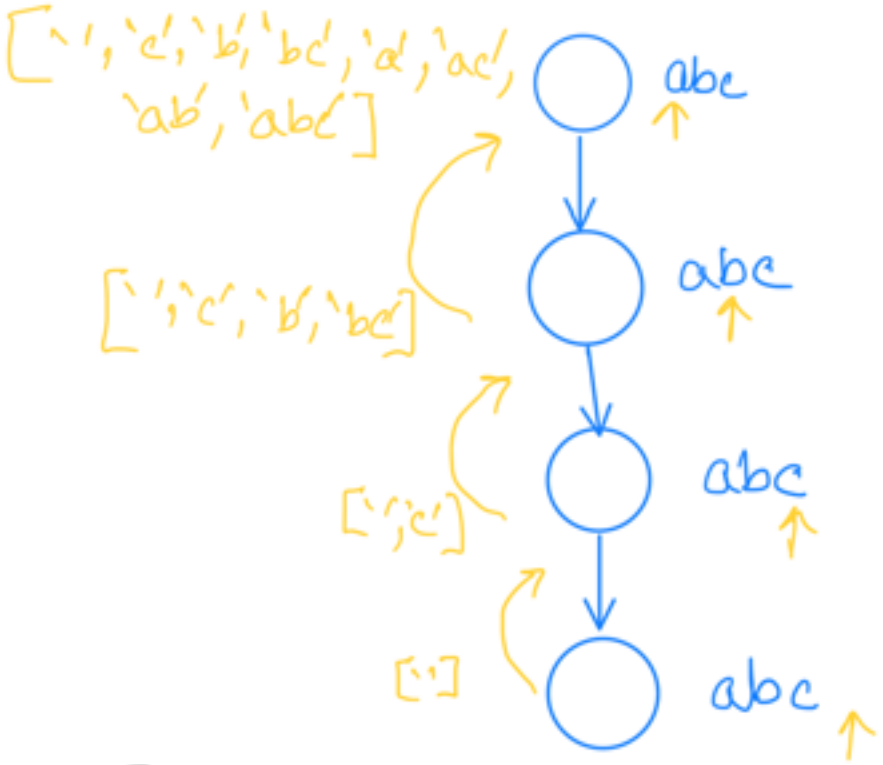
Base Case : if len(str) == 0 or if len(str)==index:
return ['']

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.

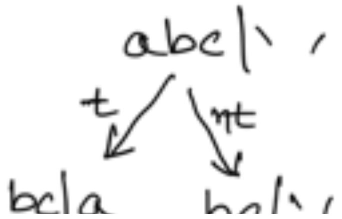
Recursion Call : return_subsequences(str[1:])
or
return_subsequences(str, index+1)

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

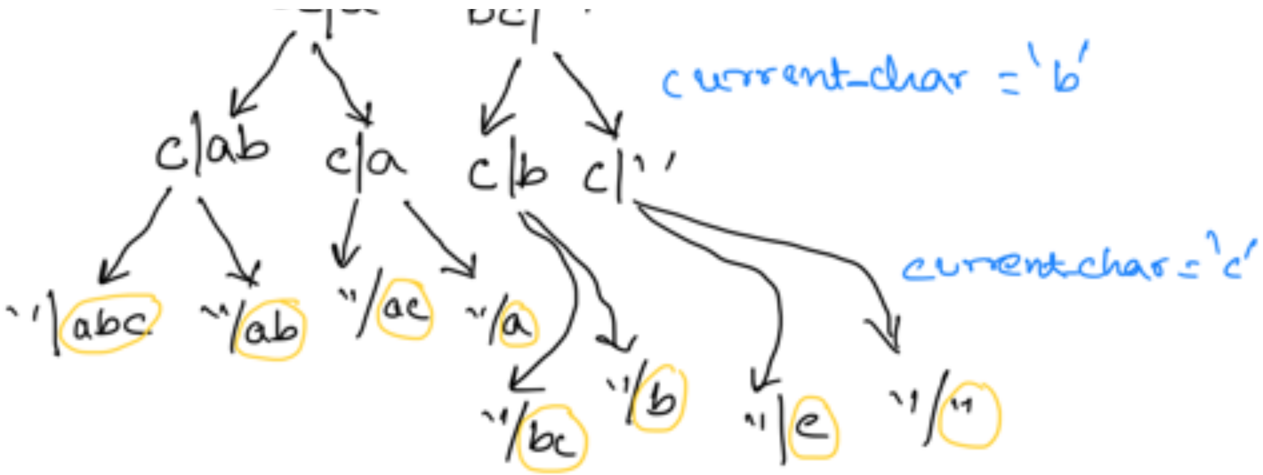
Recursion Tree

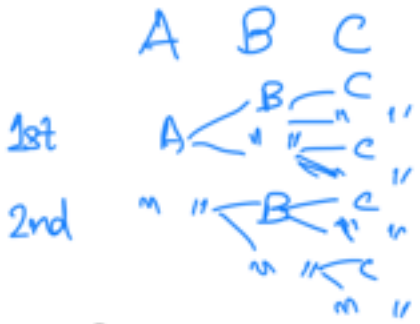


Print Subsequences of a given String



current_char = 'a'



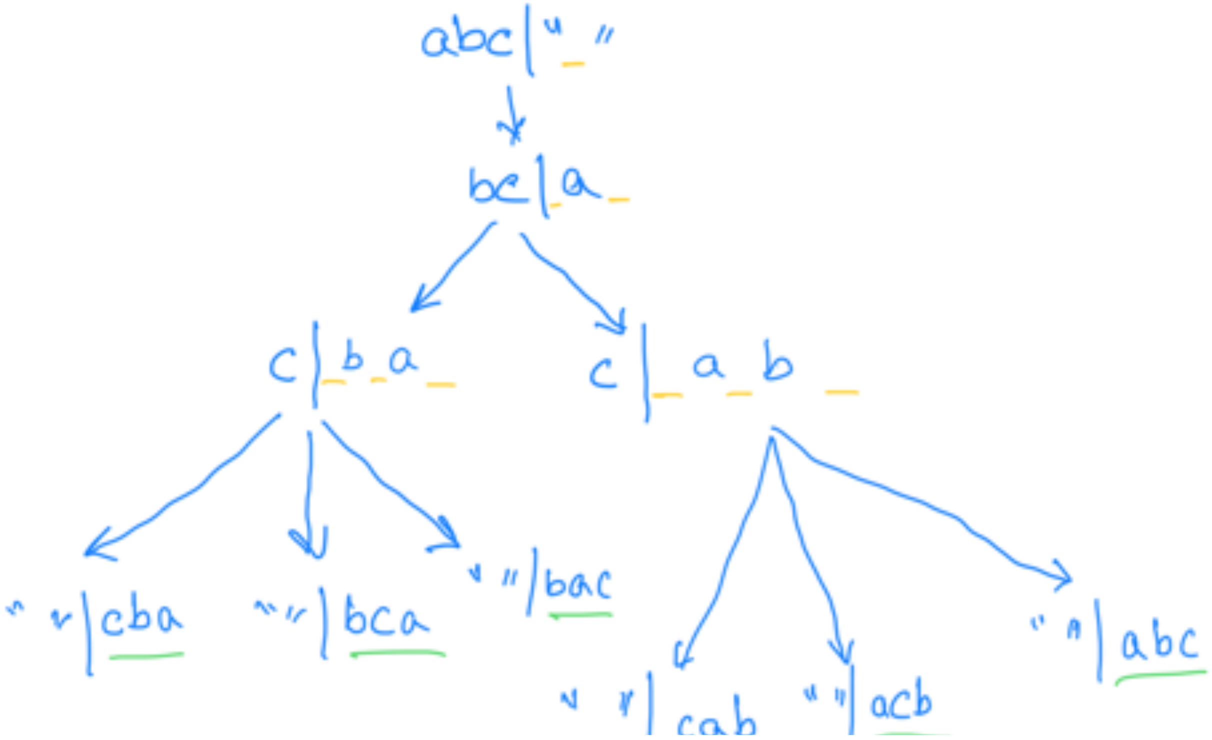


Print Permutations of a given string

string → 'abc'

permutations → abc acb
bac bca
cab cba

n! ways



Return Permutations of a String

String → 'abc'

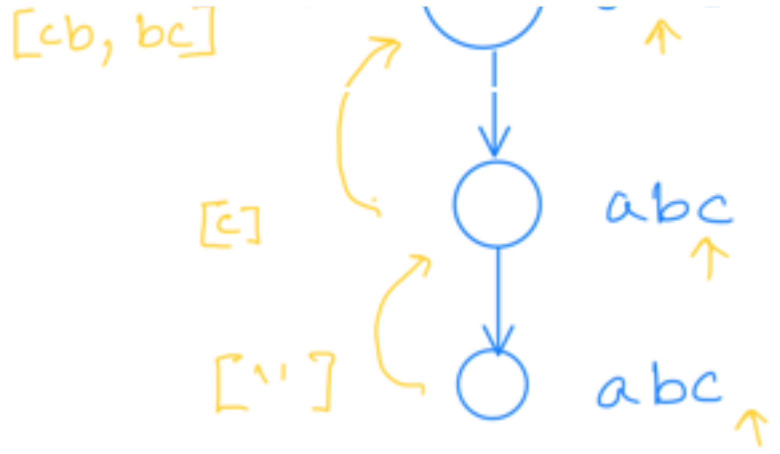
Approach : Keep current character and send smaller string to recursion call. Assume recursion call gives the permutations for smaller string. Then add current character in all possible positions to the returned permutations.

Base Case :
```
if len(s)==index or s=='':
    return ['']
```

Recursion call :
```
permutation_lst = permutations(s,index)
```

My work :
```
current_char = s[index], new_lst = [ ]
for permutation in permutation_lst:
    for position in range(len(permutation)):
        new_lst.append(s[0:position]+current_char
                                  + s[position:])
return new_lst
```

Recursion Tree



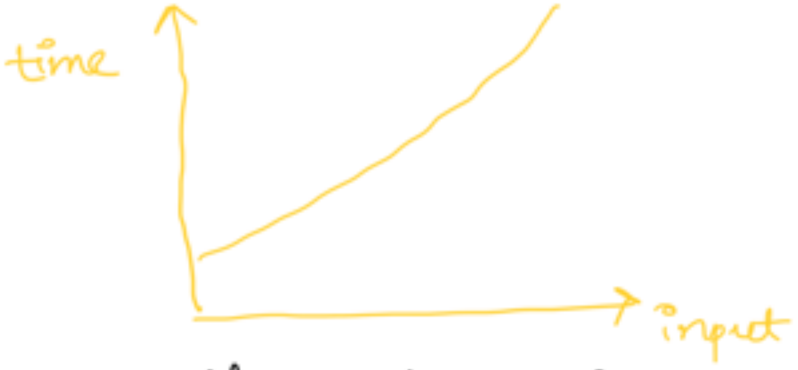
Time Complexity Analysis

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 ≠ 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

$5n^2 \lll 5n^3$

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} $\quad f(n) \longrightarrow O$ complexity

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

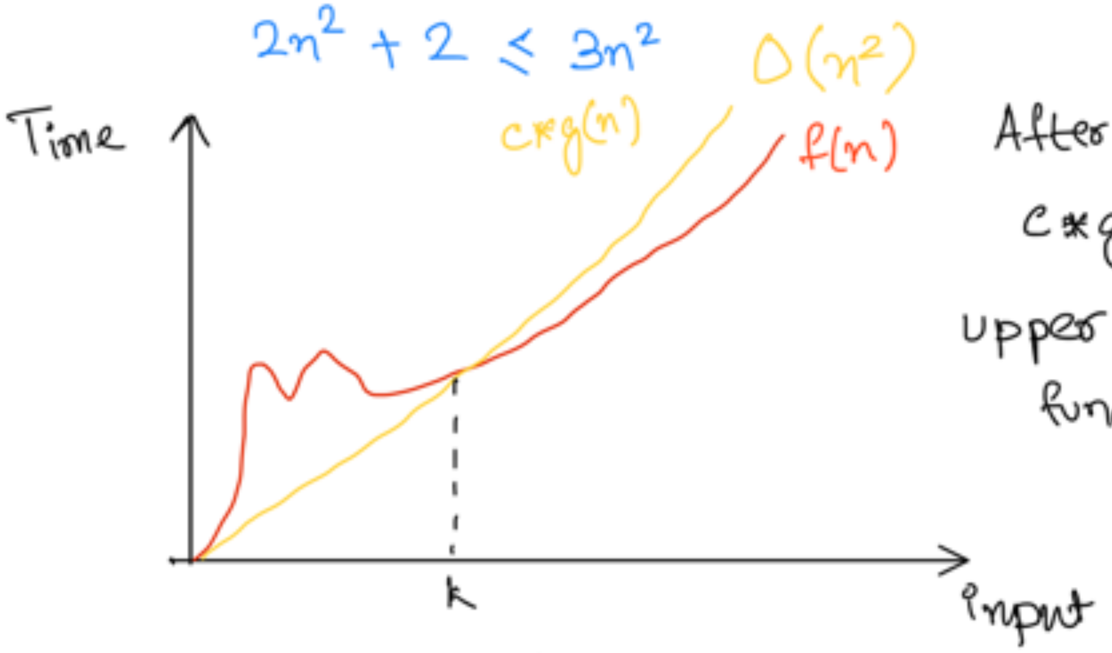
then time complexity = $O(g(n))$

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

$\downarrow$ $\downarrow$

3 n^2

$2n^2 + 2 \leq 3n^2$



After $n \geqslant k$

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

upper bound my function

Big Omega (Ω)

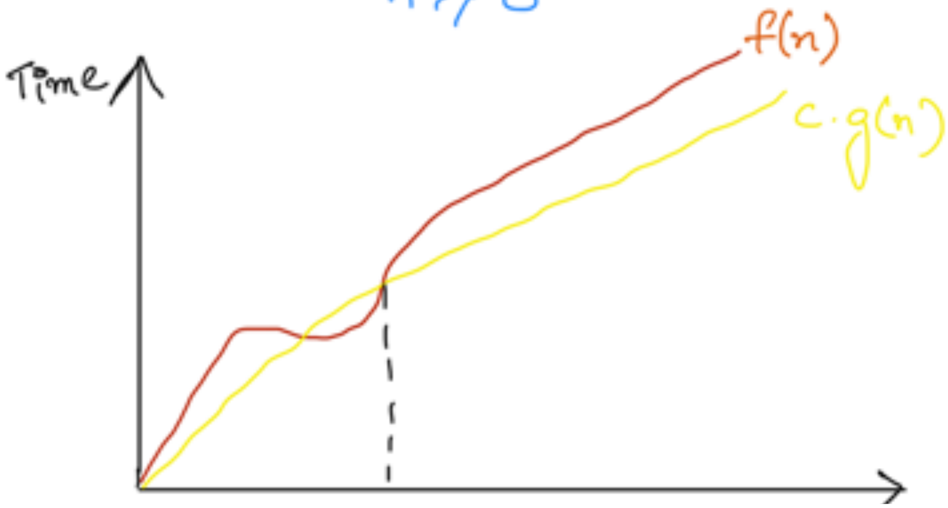
→ Best case scenario

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

when $f(n) \geqslant c \cdot g(n)$

$2n^2 + 2 \geqslant 2n^2$

$n \geqslant 0$

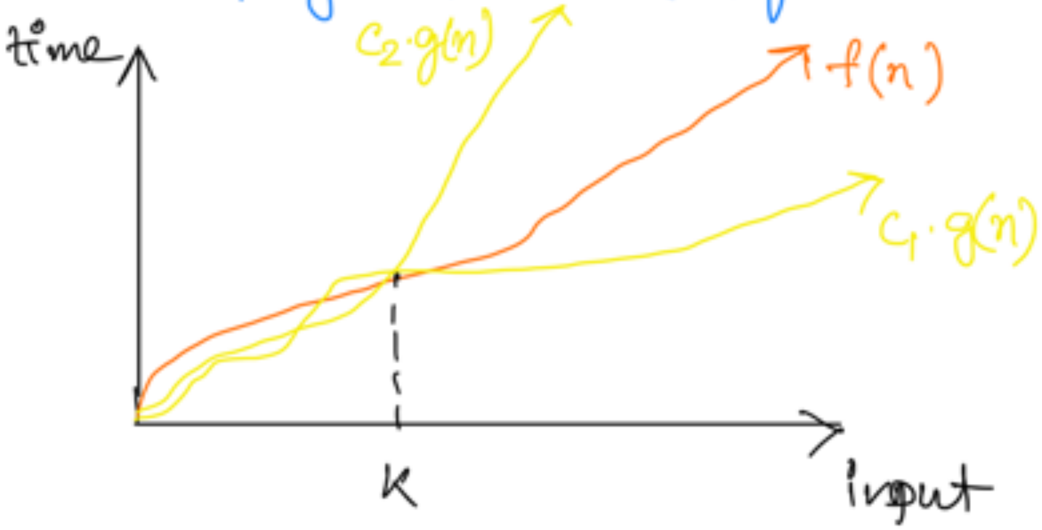


K n

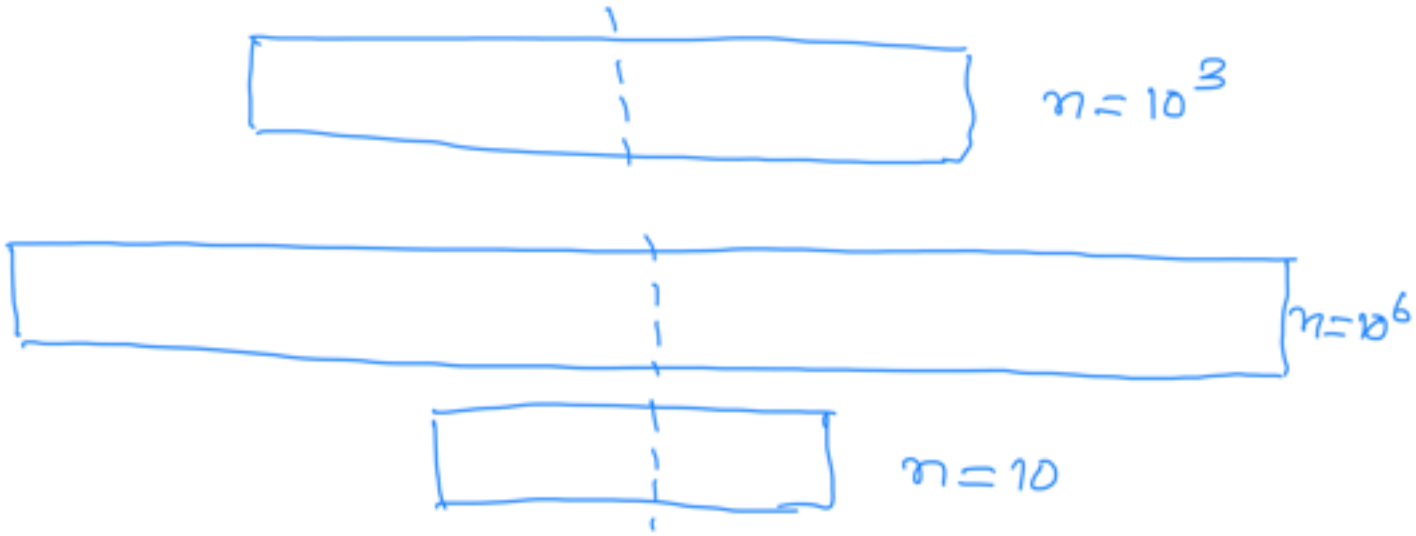
Theta (Θ)

→ Averge case complexity

$c_1 * g(n) \le f(n) \le c_2 * g(n)$



Middle of a list



mid = n//2

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

Complexity is O(1) → constant time

Largest element in an array

worst case → largest elem at the end

```
max = a[0]
for i → n
    if a[i] > max ]  } k operations
        max = a[i]   }
```

1	2	3	....	n
k	k	k		k

$$\Sigma = kn \le \underbrace{c}_{(k+1)} \cdot \underbrace{g(n)}_{n}$$

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

Linear complexity

Bubble Sort Time Complexity

```
n = len(arr)
for i in range(n)
    for j in range(n-i-1)
  k  { if arr[j] > arr[j+1]:
     {     arr[j], arr[j+1] = arr[j+1], arr[j]
```

i	j	
0	$(n-1)k$	→ 1st elem
1	$(n-2)k$	→ 2nd elem
2	$(n-3)k$	→ 3rd elem
⋮		
$n-1$	k	→ last elem

$$\Sigma = k(n-1) + k(n-2) + k(n-3) + \cdots\cdots + k$$

$$= k\left[(n-1) + (n-2) + (n-3) + \cdots\cdots + 1\right]$$

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

$1+2+3+\ldots+n$

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

$$= \frac{kn(n-1)}{2} = k\frac{n^2}{2} - \cancel{\frac{kn}{2}} \rightarrow 0$$

$$= k\frac{n^2}{2} \leq \underset{\frac{(k+1)}{2}}{c} \cdot \underset{n^2}{g(n)}$$

$g(n) = n^2$

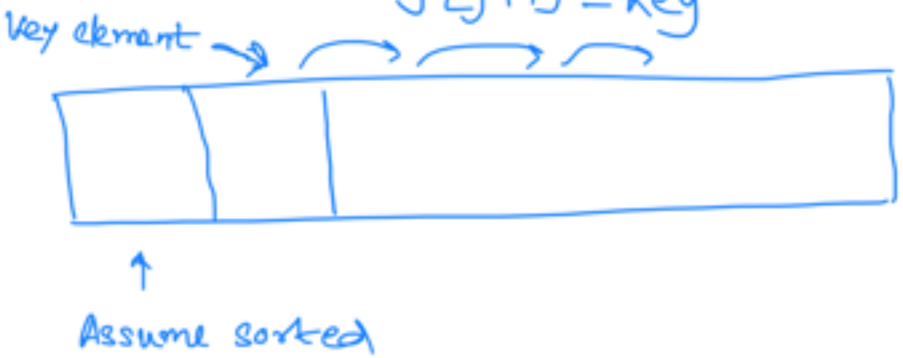
$O(n^2)$

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
```



step	j
1	$2 + 1.k$
2	$2 + 2.k$
3	$2 + 3.k$
⋮	
$n-1$	$2 + (n-1)k$

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

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

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

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

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

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

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

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

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

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

Selection Sort Time Complexity

for 1st elem $(n-1)k$ operations

for 2nd elem $(n-2)k$ operations

⋮

for $(n-1)$ elem $1 \cdot k$ operations

$$\Sigma = (n-1)k + (n-2)k + \dots + k$$

$$= \frac{k(n-1)(n-\cancel{1}+\cancel{1})}{2} = \frac{kn^2}{2} - \cancel{\frac{kn}{2}}^{\,0}$$

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

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 Algorith

[illegible] : 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 * fact(n-1)
```

$$\underbrace{n *}_{k_2}$$

fact(n) = n * fact(n-1)

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

$$T(n) = k + \cancel{T(n-1)}$$

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

$$\cancel{T(n-2)} = k + \cancel{T(n-3)}$$

$$\vdots$$

$$\cancel{T(0)} = k$$

$$a = \cancel{b} + 5$$

$$\cancel{b} = 5$$

$$a + \cancel{b} = \cancel{b} + 10$$

$$a = 10$$

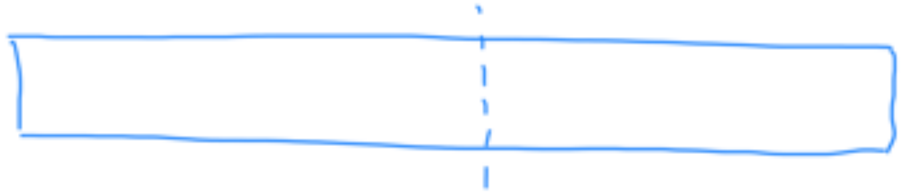
$$T(n) = k + k + k + \dots + \quad (n+1) \text{ times}$$

$$= k(n+1) = kn + \cancel{k}^{0}$$

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

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

Binary Search Time complexity



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

$\cancel{T(n/2)} = k + \cancel{T(n/4)}$

$\cancel{T(n/4)} = k + \cancel{T(n/8)}$

$\vdots$

$T(\cancel{1}) = k$ ← some constant work (base case)

$T(n) = k + k + k + \dots$? how many times

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

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

Say x times to reach 1

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

$\Rightarrow \quad n = 2^x$

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

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

$\Rightarrow \quad x = \log_2 n$

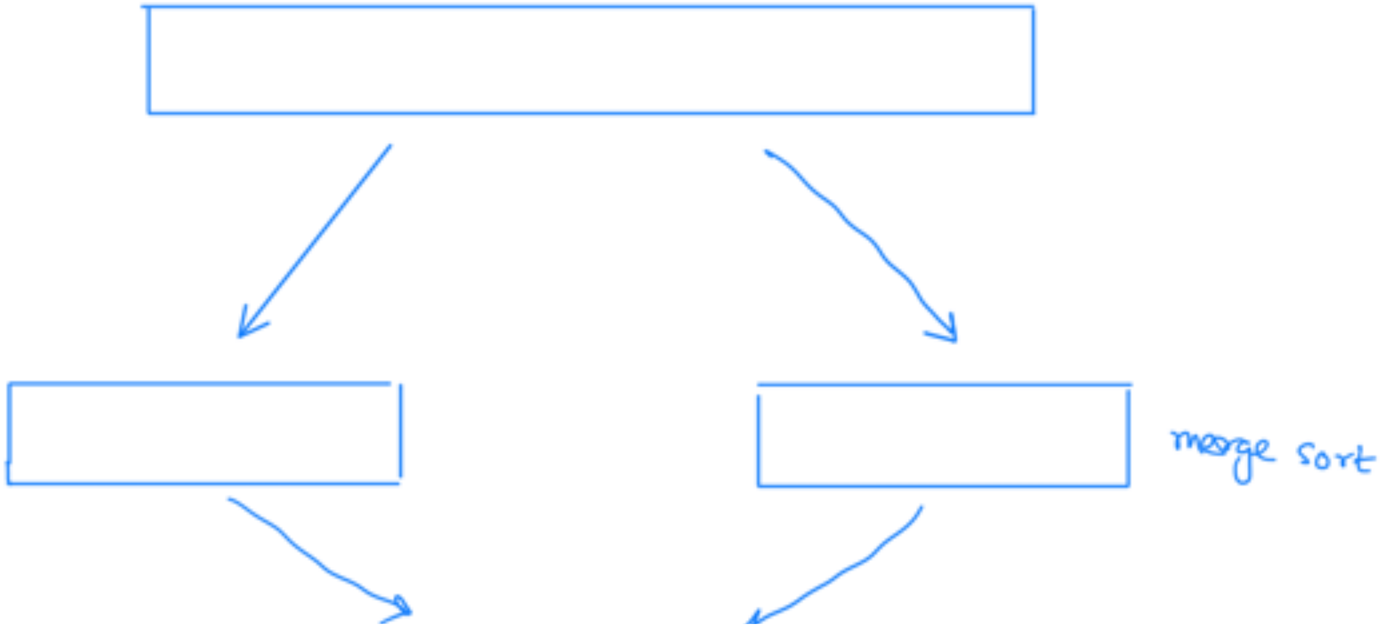
$k + k + k + \dots \log_2 n$

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

Time complexity of binary search is $\log_2 n$

10^6 elements : 20 comparisons

Merge Sort Time Complexity



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

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

$$T(n) = kn + \cancel{2T(n/2)}$$

$$\cancel{2T(n/2)} = k\frac{n}{\cancel{2}} + \textcircled{4}\,\cancel{2T\left(\frac{n}{4}\right)} \quad \times 2$$

$$4\,\cancel{T(n/4)} = k\frac{n}{\cancel{4}} + \textcircled{8}\,\cancel{2T\left(\frac{n}{8}\right)} \quad \times 4$$

$$\cancel{8} \quad \vdots \qquad\qquad \times 8$$

$$\cancel{T(1)} = kn$$

$$T(n) = kn + kn + kn + \cdots\cdots \; x \text{ times}$$

$$n \quad \frac{n}{2} \quad \frac{n}{4} \quad \frac{n}{8} \quad \cdots\cdots \quad 1$$

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

$$n = 2^x$$

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

$$x = \log_2 n$$

$\Rightarrow$

$+ \; kn.\log_2 n \quad \leq (k+1) \, n \log_2 n$

Time complexity $= O(n \log n)$
(best, avg, worst case)

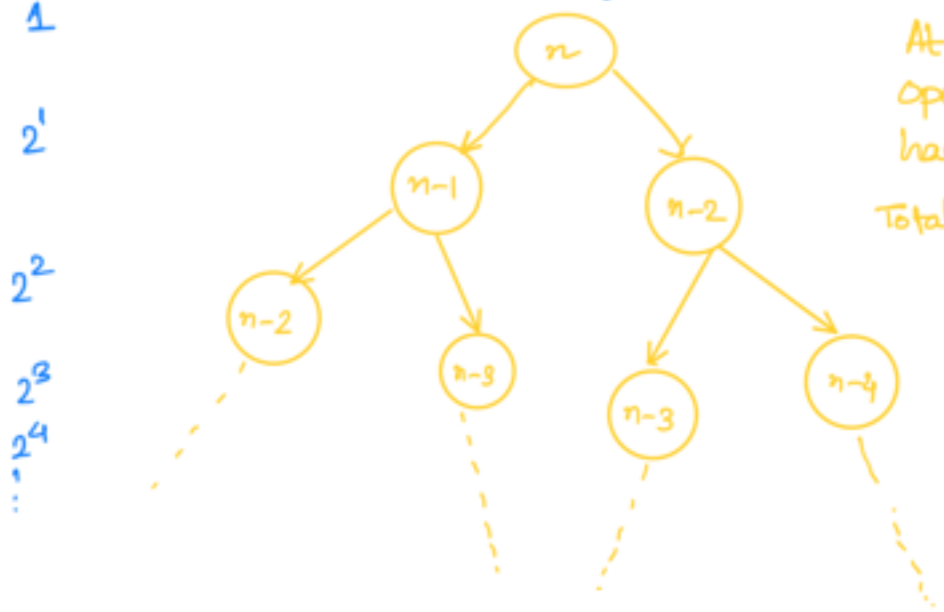
Fibonacci Number - Time complexity

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

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

} k operations

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



At each node k operations are happening
Total operations
= k * no. of nodes

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

∴ No. of Nodes

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

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

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

$$2 - 1$$

$$= 2(2^n - 1)$$

$\therefore$ No. of operations

$= k * \text{No. of nodes}$

$= k * 2(2^n - 1)$

$= 2^{n+1} k - \cancel{2k}^{\nearrow 0}$

$T(n) = k \, 2^{n+1}$

$= 2k \, 2^n \leq 3k \, 2^n$

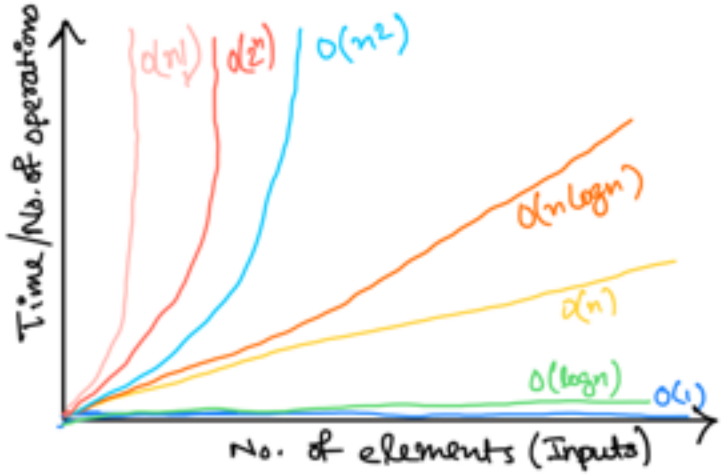
$\therefore$ Complexity is $O(2^n)$ i.e. exponential

Space Complexity

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

Space complexity = Input space
+
Auxiliary space

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 ~ 10^8 ops/sec

In problems on internet, the same is also provided in question defn

e.g. Constraints : $1 <= nums, length <= 5\times10^4$

if we apply, algorithms like bubble sort for max length array, no. of operations = $O(n^2) = (5\times10^4)^2 = 25\times10^8 > 10^8$

Hence will give time limit exceeded error.

How to determine the solution by looking at constraint?

Constraint	Worst Time Complexity	Algo
$n \leq 12$	$O(n!)$	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 ~ 10^8

Above is a

help in remembering.

Space Complexity

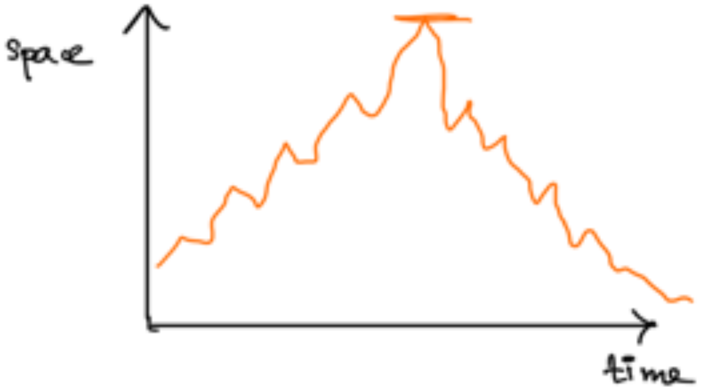
Space complexity is space required as a function (f(n)) of input size

Space complexity = input space
+
auxiliary space

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

Merge

sorted array = [ ] dependent on input size



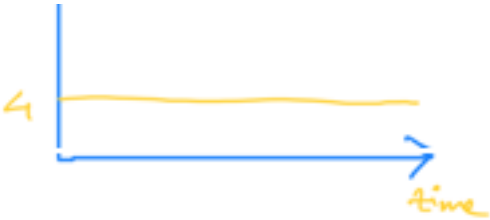
We are only concerned about the maximum space that is required during the entire algorithm

8gb

install	2gb	+
install	2gb	+
delete	4gb	−
install	6gb	+

n = 100 000
while (i <= n) : space ↑

a = 5
a+ = 1



Space complexity of algorithms

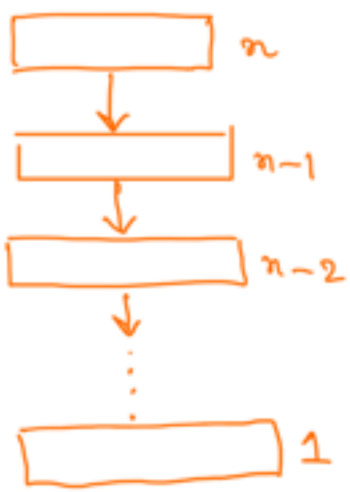
Insertion Sort : $O(n^2)$ = $O(1)$
Bubble Sort : $O(n^2)$ = $O(1)$
Selection Sort : $O(n^2)$ = $O(1)$

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

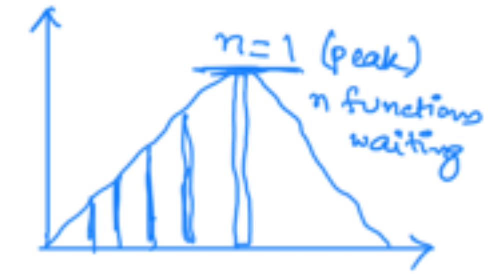
Space Complexity - Recursive Algorithm

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

space ⓚ



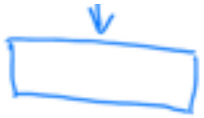
Each function takes k space



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

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

Binary Search (Recursion)



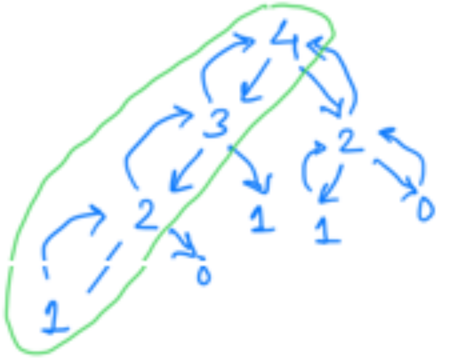
$n/4$ k

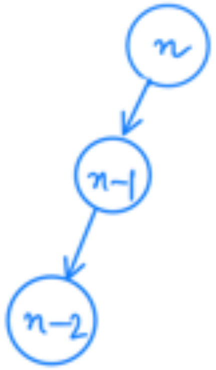
k * (no. of functions) ⇒ $k. \log_2 n$

We will have $\log_2 n$ functions waiting

Fibonacci Number : Space Complexity

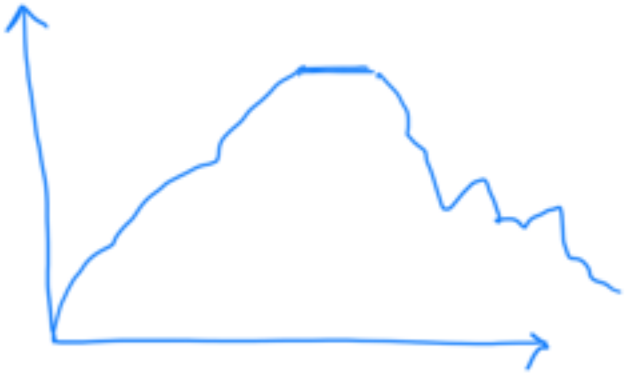
```
def fib (n):
    if n<= 1
        return 1
    else fib(n-1) + fib(n-2)
```





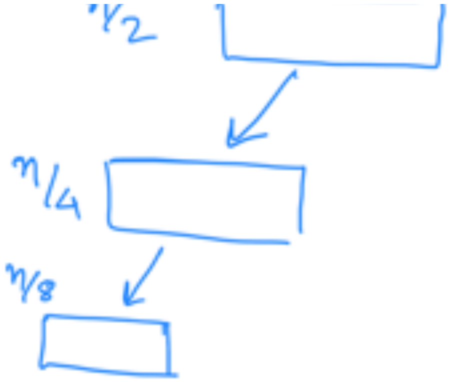
maximum number of calls in memory = n

So space complexity = kn



Merge Sort - Space Complexity

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

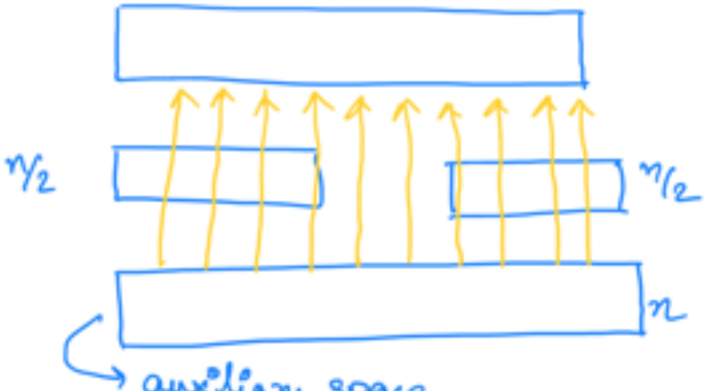


first left sub-tree is going to run

```
def merge sort(arr)

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

$n$ { merge(left, right)
```



auxiliary space

Space complexity

$= O(\log_2 n + n)$

for a very large n

$\underline{\underline{n}} >> \log_2 n$

major term

My major term is n

Space complexity → kn

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

array of n space

bigo cheatsheet.com