Week 2

Saturday, August 23, 2025 6:57 PM

Space Complexity →

The amount of space needed in memory for an algorithm to fully execute. 2 main components

→ fixed size (size of code) (C) → Variable part (depends on input) (Sx)

→ Space complexity => $T_{space} = C + S_x$

Time Complexity → Amount of time the computer will need for completing the execution of a function/ algorithm.

This has 3 cases → Best Case
→ Avg Case
→ Worst Case

Notations for representing complexity →

1) Big-O (upper bound) Time Complexity won't go higher than this.

2) Omega (Ω) (lower bound) Time complexity won't go lower than this.

3) Theta (Θ) (tight bound) Somewhere in between Big-O and Ω always.

→ Generally we follow the Big-O notation in our course.

- $O(1)$ - constant time
- $O(logn)$ - logarithmic time

Big-O notation for some common scenarios,

- $O(1)$ - constant time
- $O(logn)$ - logarithmic time
- $O(n)$ - linear time
- $O(nlogn)$ - linearithmic time
- $O(n^2)$ - quadratic time
- $O(n^3)$ - cubic time
- $O(2^n)$ - exponential time
- $O(n!)$ - factorial time

Big-O notation for some common scenarios, from slowest-growing to fastest-growing!

Calculating Complexity →

May 2022

```
def fun(arr):
    n = len(arr)          → O(1)
    if n%2 == 0:          → O(1)
        while n>1:        } O(log n)
            n = n//4      }
            print(arr[n])
    else:
        for i in range(0,n):
            print(arr[i])
```

4·1·2 = 0
16·1·2 = 0
TC = ?
32·1·2 = 0, n = 5
5·1·2 ≠ 0
n = 4 → n = 1
→ O(5)

① Take n = 16 → O(log n)

② Take n = 32 → O(n)

↓ because of else statement!

⇒ Ans → O(n)

May 2023

```
def fun(n):
    count = 0
    for i in range(n):        O(n)
        j = 1                 → O(1)
        while j<n:            → O(n)   → O(n²)
            count = count+1
            j = j*2           → O(log n) → O(n log n)
    return count
```

TC = ?

O(n) ↓ outer loop runs n times

inner computation happens log n times.

① n = 8

⇒ Ans → O(n log n)

⑤ $f1(n) = 3n^2 + 2n = n^2$

③ $f2(n) = 3n + (\log n)^2 = n$

① $f3(n) = \log(\log n) = \log(\log n)$

② $f4(n) = 10\log n = \log n$

④ $f5(n) = 3n\log n = n\log n$

Arrange all functions in ↑ increasing order of asymptotic complexity

Lets try for n = 16

→ $f_1(n) = 3(16)^2 + 2(16)$

→ $f_2(n) = 3(16) + (4)^2 = 4 \times 16$

→ $f_3(n) = \log(\log 16) = \log(4) = 2$

→ $f_4(n) = 10 \times 4 = 40$

④ $f5(n) = 3n\log n = n\log n$ order of asymptotic complexity?

$\Rightarrow f_4(n) = 10 \times 4 = 40$

$\Rightarrow f_5(n) = 3(16) \times 4 = 12 \times 16$

Ans → $f_3 < f_4 < f_2 < f_5 < f_1$

Linear Search :

→ O(n) Complexity

```
def naivesearch(L,v):
  for i in range(len(L)):
    if v == L[i]:
      return i
  return(False)
```

→ search each element of the list.

→ If matches; return Index else FALSE

Binary Search :

Need a sorted list. Define a start & end.

⇒ Mid = (start + end) // 2

```
def binarysearch(L, v):  #v = target element
    low = 0
    high = len(L) - 1
    while low <= high:
        mid = (low + high) // 2
        if L[mid ] < v:
            low = mid  + 1
        elif L[mid ] > v:
            high = mid  - 1
        else:
            return mid
    return False
```

→ Termination Condition

→ Edit LOW

→ Edit HIGH

→ Return index if found, else FALSE.

Complexity → O(log n)

Q [16, 53, 59, 81, 94, 99, 121, 150, 162, 170]

apply BS to search 105 in given list. Then the no. of comparisons of searching element 105 with list elements done in this process is ___?

low = 0, high = 9

① mid = 4, ele = 94, No match

② low = 5 ⇒ mid = 7, ele = 150, No match

③ High = 6, mid = 5, ele = 99, No match

④ Low = 6, mid = 6, ele = 121, No match

→ 4 Comparisions.

Selection Sort → Select the first element of list as min element

start comparing with the entire list elements, make a swap at the place, such that the element at the minimum pos is at it's sorted location after the complete iteration.

→ In place sorting algorithm (No extra space needed)

→ Not stable. (similar elements may not retain their order after sorting is done)

```
def selectionsort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        minpos = i
        for j in range(i+1,n):
            if L[j] < L[minpos]:
                minpos = j
        (L[i],L[minpos]) = (L[minpos],L[i])
    return(L)
```

→ Pivot selected

} For all elements after pivot

j will be the index of the smallest element

↓ swap the smallest one with the pivot. (List starts getting sorted)

Complexity → for best case → $O(n^2)$

for avg case → $O(n^2)$

for worst case → $O(n^2)$

Insertion Sort → Iterate through the list and compare the adjacent elements. If they are not correctly placed, SWAP them.

→ In-place Sorting

→ stable sorting

```
def insertionsort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        i = i
```

→ For every element in the list, compare with the adjacent,

```
    if n < 1:
        return(L)
    for i in range(n):
        j = i
        while(j > 0 and L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
    return(L)
```

→ For every element in the list, compare with the adjacent, if wrong order do a swap. else, get a new element, until unseen elements are finished.

↓

Return sorted list

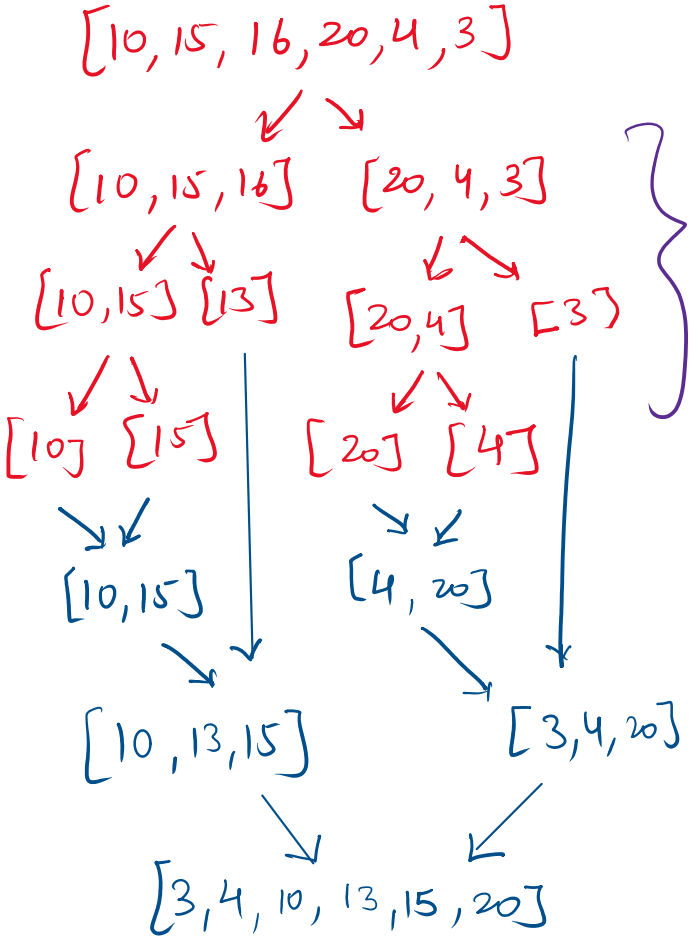
Complexity → for best case → $O(n)$ [Already sorted list]

for avg case → $O(n^2)$

for worst case → $O(n^2)$

Merge Sort → Has 2 parts → One is the merge function.

→ One is for applying merge function on the splitted list, upto single element in each list.



Breaking down in halves so that we obtain lists of each element.

Merge operation.

→ Sorts the list at time of merging

→ Not inplace Sorting → Stable sorting

```
def merge(A,B): # Merge two sorted list A and B
    (m,n) = (len(A),len(B))
    (C,i,j) = ([],0,0)

    #Case 1 :- When both lists A and B have elements for comparing
    while i < m and j < n:
        if A[i] <= B[j]:
            C.append(A[i])
            i += 1
        else:
            C.append(B[j])
            j += 1

    #Case 2 :- If list B is over, shift all elements of A to C
    while i < m:
        C.append(A[i])
        i += 1

    #Case 3 :- If list A is over, shift all elements of B to C
    while j < n:
        C.append(B[j])
        j += 1

    # Return sorted merged list
    return C
```

→ initializations

} Comparing and appending to the ans list

} Fill remaining values of A in C.

} Fill remaining values of B in C.

↳ Returns the sorted list after merging the smaller parts.

```
def mergesort(L):
    n = len(L)
    if n <= 1: #If the list contains only one element or is empty return the list.
        return(L)
    Left_Half = mergesort(L[:n//2])
    Right_Half = mergesort(L[n//2:])
    Sorted_Merged_List = merge(Left_Half, Right_Half)
    return(Sorted_Merged_List)
```

} Recursive implementation of breaking the initial list into smaller parts.

And applying merge function on them.

⇒ Recurrance Relation → $T(n) = 2T(n/2) + O(n)$

Complexity → for best case → $O(n \log n)$

for avg case → $O(n \log n)$

for Worst case → $O(n \log n)$

Q What are the min. and max. no. of swapping operation in Insertion sort?

Min swaps are for a sorted list

Q What are the min. and max. no. of swapping operation in Insertion sort ?

Min swaps are for a sorted list
⇒ 0 swaps.

Max swaps are for Reverse sorted list
⇒ $\frac{n(n-1)}{2}$ swaps.

Q [38, 28, 43, 22, 112, 33, 39]
no of swap using Insertion sort

22, 28, 33, 38, 43, 112, 39 (11)

⇒ 9 swaps.

38 ↙ 28 (1)
↓
28, 38, 43 ↙ 22 (III)
↓
22, 28, 38, 43, 112 ↙ 33 (III)
↓
22, 28, 33, 38, 43, 112

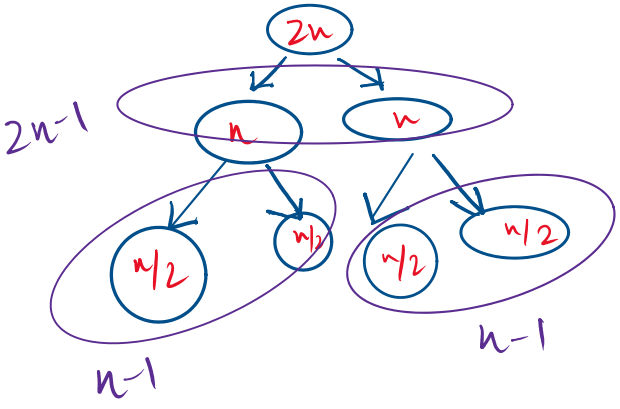
Q 4 sorted lists each of length $n/2$ are merged into a single sorted list of $2n$ using two way merging.

what will be the max no of element comparison needed for this process?

for 2 lists, of $n/2$ length each

L1 = [1, 3, 5, 7]
L2 = [2, 4, 6, 8]
→ out = [1, 2, 3, 4, 5, 6, 7, 8]
→ 7 comparisions.

⇒ $2 \times \frac{n}{2} - 1$ Comparisions



⇒ Total comparisions → 2n-1 + n-1 + n-1

Ans → 4n-3