DSML : Python Intermediate

Recursion - 2.

Recap: (a) Recursion: Repeating computation.

(b) Steps: Assumption, subproblem, Base cases.

(c) Dry run: function stack, trees

(d) Problems: print_n, summation, factorial, fibonacci, sorting.

Today: (a) Problems: sum_of_digits, power, recursive merge sort.

(b) Time complexity analysis:

(i) Recurrence relation.

(ii) Tree method.

(c) (Optional) Print all permutations of a string

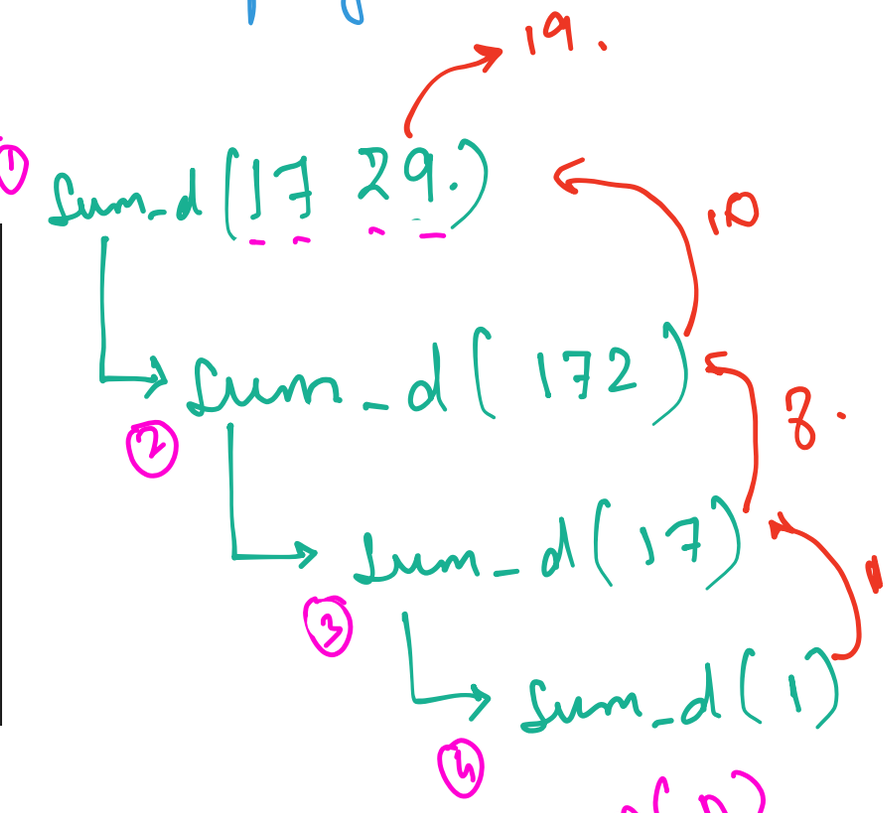
(a) [Sum of digits] Given a number n, write a recursive function to print the sum of the digits of n.

Let the name of the function which does this be called sum_d(n).

- **Assumption**: The sum_d(n) function will return the correct sum of digits for a k-digit number n.
- **Subproblem**: Sum_d(n) = sum_d(n // 10) + n % 10
- **Base Case**: if n == 0, return 0.

Code / Dry Run: Sum of digits.

```python
def sum_d(n):
  if n == 0:
    return 0
  return n%10 + sum_d(n//10)

print(sum_d(1729))
print(sum_d(101010))
print(sum_d(11235))
```



1729 // 10

= 172.

Case 1: k = 4, T(k=4) = 5

O(n)

Case 2: k = 4, T(k=4) = 4.

O(n)

(b) [Power] Given the base a and exponent n, write a recursive function that returns a^n. Let the function which does this be called pow(a, n). → $O(\log_2 n)$

Assumption pow(a, k) will return the correct answer.

Subproblem :

(i) pow(a, k) = pow(a, k-1) * a

(ii) If k is even : t = pow(a, k//2), return t*t

If k is odd : t = pow(a, k//2). return t*t*a.

Base Cases.

i) n == 0, return 1.

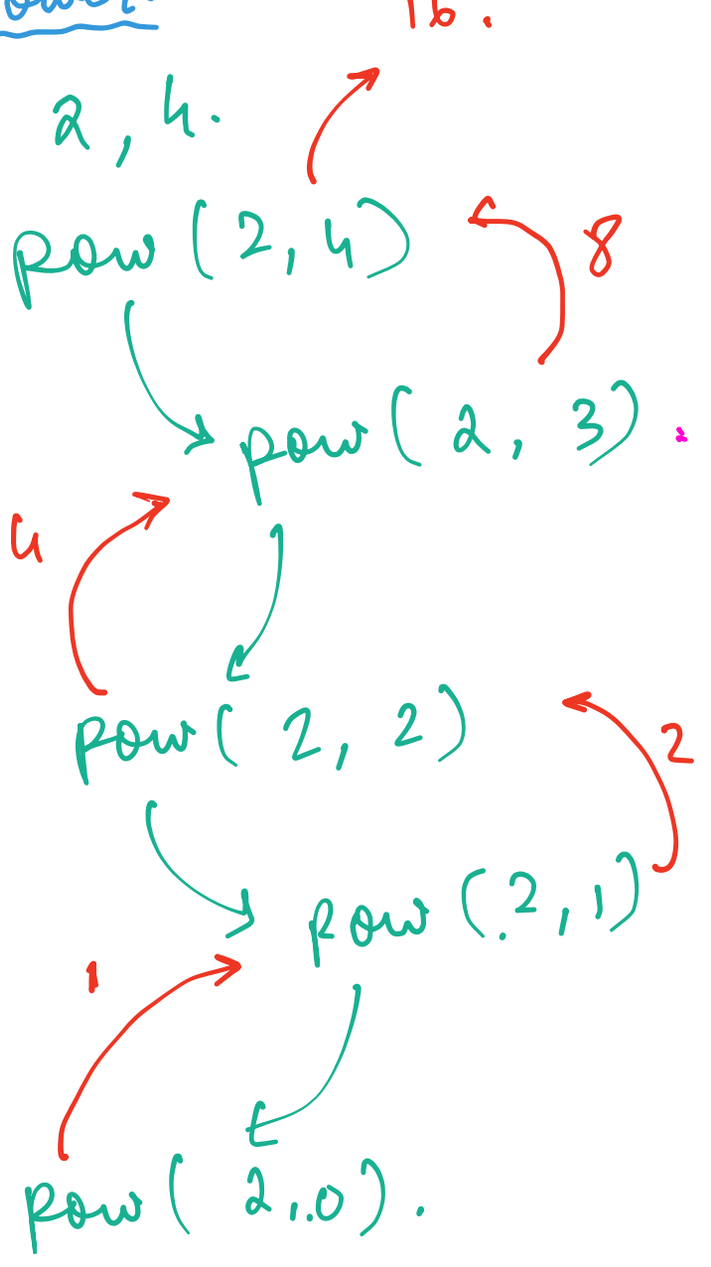
ii) n == 0, return 1

Code / Dry Run : Power.

```
def pow(a, n):
  if n == 0:
    return 1
  return pow(a, n-1) * a

print(pow(2, 4))
```

$n = 4$, $T(n) = 5$.

$O(n)$.

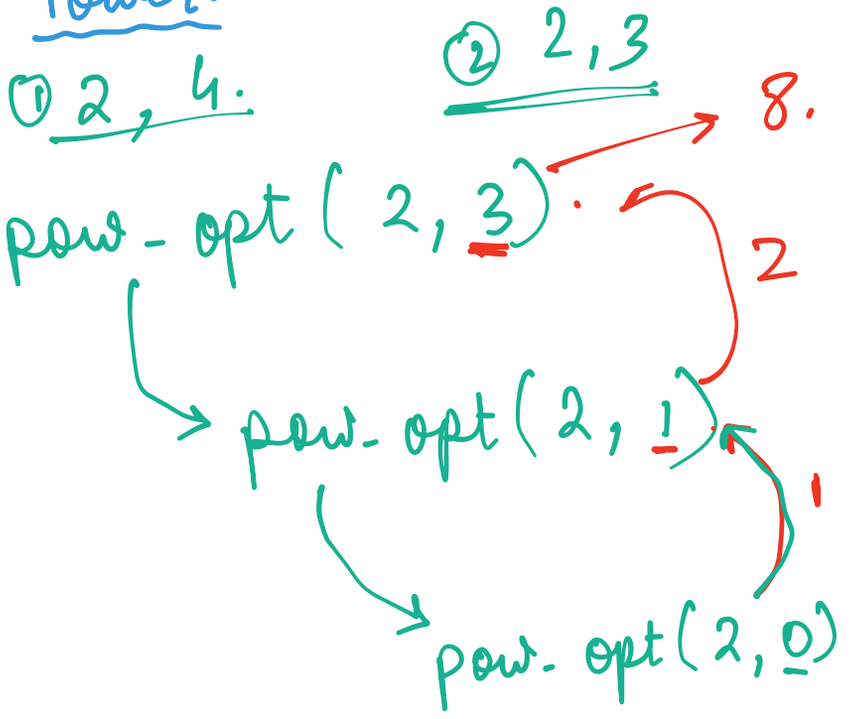


Code / Run : Power.

```
def pow_opt(a, n):
  if n == 0:
    return 1
  t = pow_opt(a, n // 2)
  if n%2 == 0:
    return t*t
  else:
    return t*t*a
```

① 2, 4.

② 2, 3



$T(n) = \log_2(n)$.

3 // 2 = 1

1 // 2 = 0

(c) Recursive Merge Sort: Sort an array a using the merge function and a recursive function. Let the function which does this be merge_sort(a).

```
def merge(A, B):
  idx_A = 0
  idx_B = 0
  C = []
  while idx_A < len(A) and idx_B < len(B):
      if A[idx_A] < B[idx_B]:
        C.append(A[idx_A])
        idx_A += 1
      else:
        C.append(B[idx_B])
        idx_B += 1
  return C + A[idx_A:] + B[idx_B:]
```

Assumption: merge_sort(a) will correctly sort the array a of length k.

Subproblem
↳ break the array into 2 parts, sort, and apply merge!

Base Case: [], [3]

a = [1, 3], [2, 4]
A B.

n = len(a)

k = len(a) // 2.

merge_sort(a)
k.

Recursive Merge sort.

```
1 def merge_sort(arr):
2   if len(arr) <= 1:
3     return arr
4
5   mid = len(arr) // 2
6   A = merge_sort(arr[:mid])
7   B = merge_sort(arr[mid:])
8
9   return merge(A, B)
```

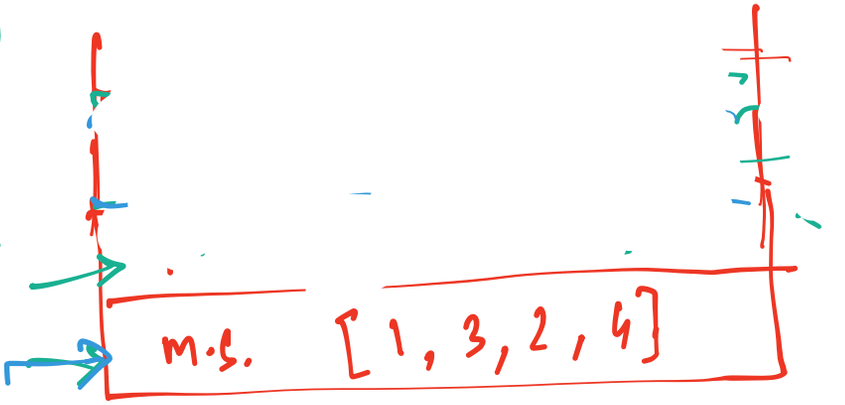
arr = [1, 3, 2, 4]

merge_sort(arr).

↓

[1, 2, 3, 4].

Line	Command
5	A = [1, 3]
6	B = [2, 4]
7	[1, 2, 3, 4].



(d) Time Complexity Analysis:

(a) The print_n function : Recurrence Relation approach.

```
1] def print_n(n):
2]     if n == 0:
3]         return
4]     print(n)
5]     print_n(n-1)
```

1
1
1
?

$T(n-1) = 3 + T(n-2)$

$T((n-1)-1) = 3 + T((n-1)-2)$

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

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

⋮

$T(n) = 3n + 2 \rightarrow O(n)$. $+ T(0)$.

"Recurrence relation"

(d) Time Complexity Analysis:

(a) The print_n function : Tree approach.

```
def print_n(n):
    if n == 0:
        return
    print(n)
    print_n(n-1)
```

O(1)

O(1).

print_n(n)

print_n(n-1)

print_n(n-2)

print_n(0)

$T(n) = n + 1$

$\downarrow$

$O(n)$

(d) Time Complexity Analysis:

(b) The pow function.

```
def pow_opt(a, n):
  if n == 0:
    return 1
  t = pow_opt(a, n // 2)
  if n%2 == 0:
    return t*t
  else:
    return t*t*a
```

Assume that the cost for pow_opt(a, n) is T(n).

pow_opt(2, 4).

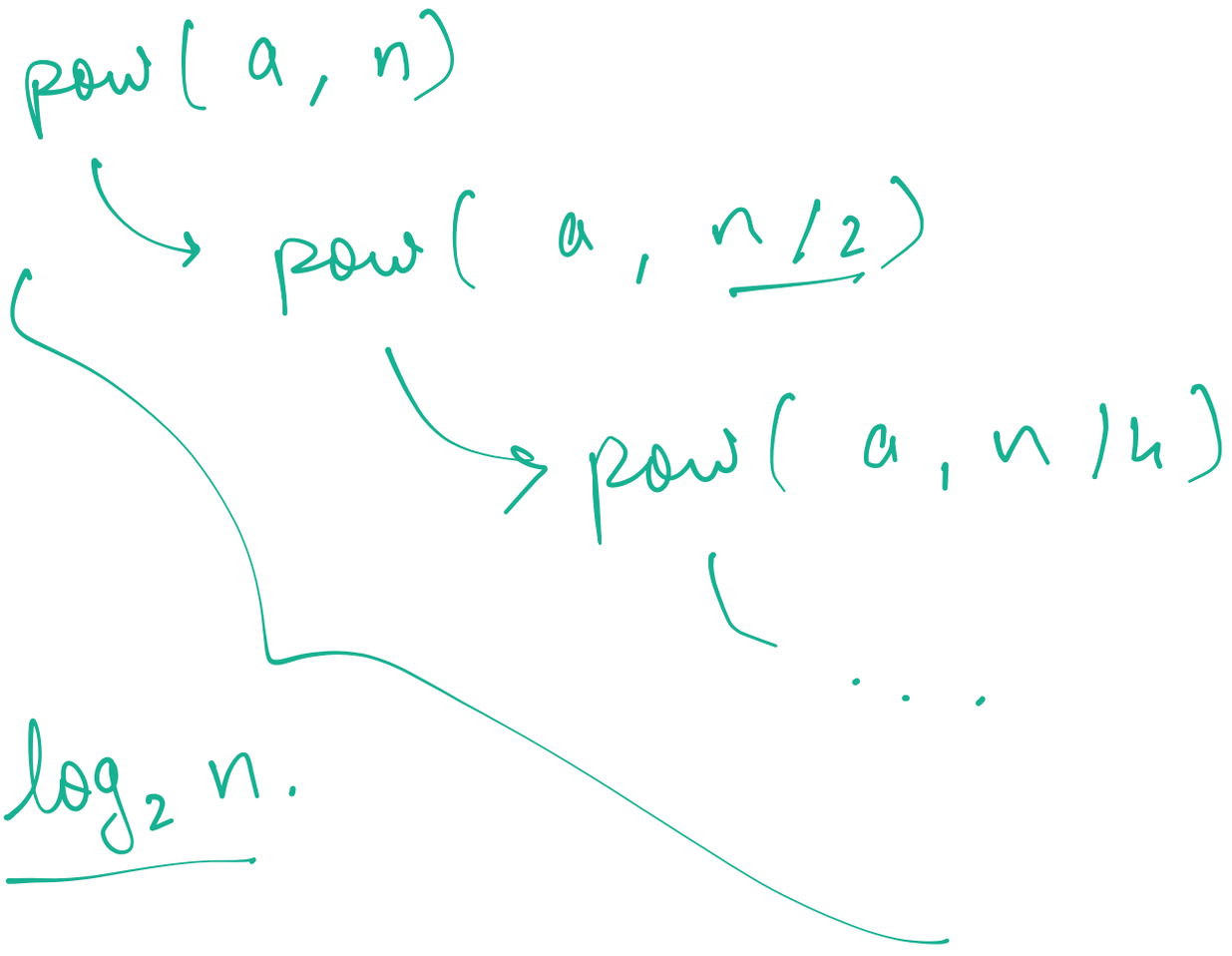
T(0) → 2

$T(n) = T(n/4) + 4 + 4$ → Recurrence relation.

$T(n) = 4 \cdot \log_2(n) + 2.$

→ $O(\log_2 n)$.

(d) Time Complexity Analysis:



$T(n) = \log_2 n$.

(e) [String Permutations] Given an input string, print out all possible permutations of the string. Assume all chars are unique.

Let the function which does this be called str_perm(s).

Eg: s = 'abc'

Expected output:

['abc', 'acb', 'bac', 'bca', 'cab', 'cba'].

s = "Hello"

len(s)

```
def len_s(s):
    if s == '':
        return 0
    return 1 + len_s(s[1:])
```

→ S = "hi ihihiihh"

Problem: count_hi(s) → gives me the total no. of 'hi' in s.

→ Assumption: count_hi(s) will give the correct count upto len k.

→ Subproblem → if s[:2] == 'hi':
return 1 + count_hi(s[2:])

→ else:
return count_hi(s[1:])

→ Base case:

→ if len(s) ≤ 2:
return 0, 1, 'h', 'hi'

```
def pow_opt(a, n):
  if n == 0:
    return 1
  t = pow_opt(a, n // 2)
  if n%2 == 0:
    return t*t
  else:
    return t*t*a
```

pow_opt(2, 3)
↳ T(n/2)

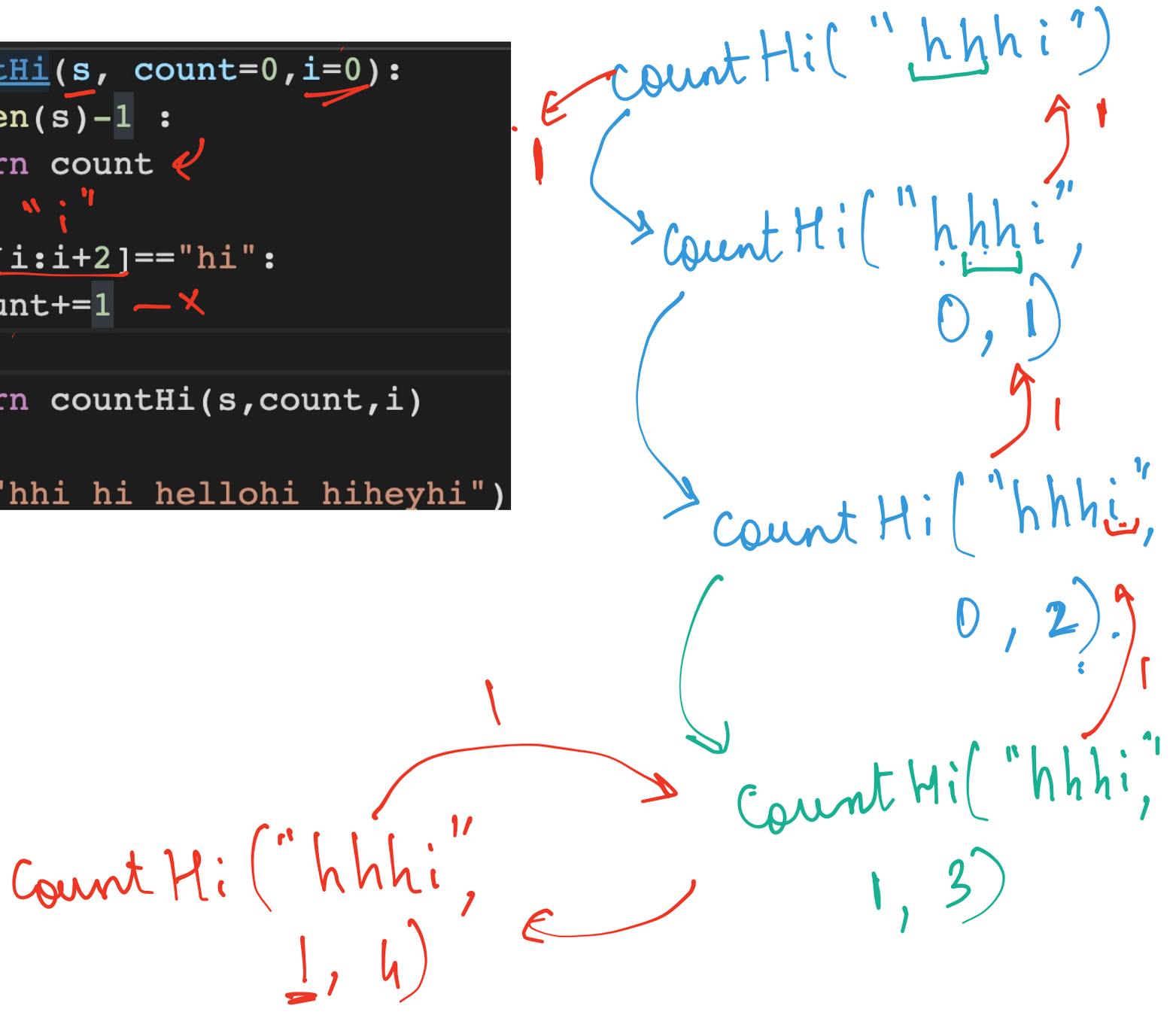
pow_opt(2, 3//2)
↳ T(n/2).

$$T(n) = 1 + T(n/2) + 1 + 1$$

$$T(n) = T(n/2) + 3 + 3 + 3 \rightarrow \text{Recurrence relation.}$$

$$T(n) = 3 \times \log_2(n) \rightarrow O(\log_2 n).$$

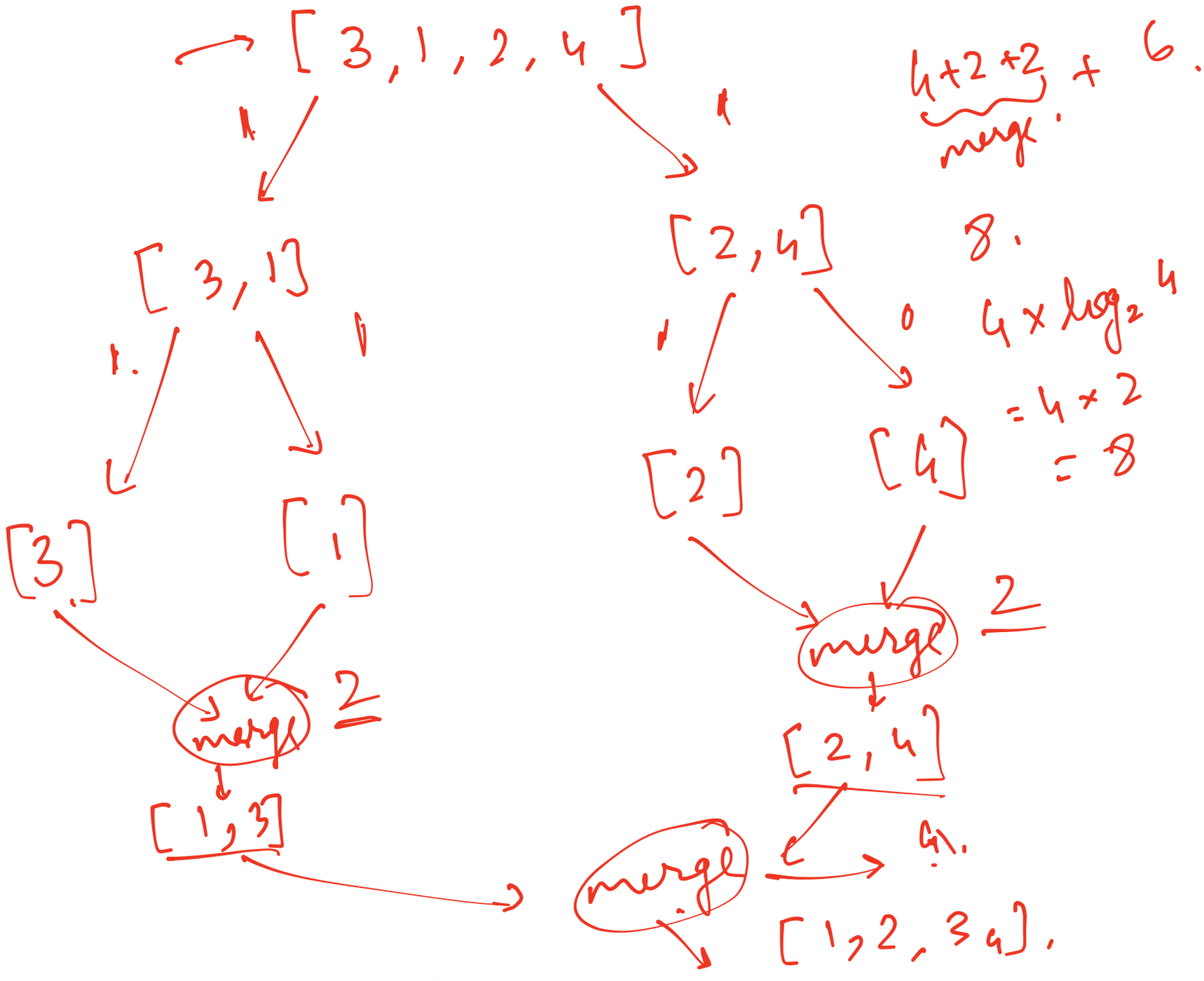
```python
def countHi(s, count=0,i=0):
  if i>len(s)-1 :
    return count
  else:
    if s[i:i+2]=="hi":
      count+=1
    i+=1
    return countHi(s,count,i)

countHi("hhi hi hellohi hiheyhi")
```



$n \rightarrow$ merge

$\log_2 n \rightarrow$ sort.

total : $O(n \log_2 n)$


[3,1,2,4]
4+2+2 + 6.
merge.
[3,1]
[2,4]
8.
4 x log2 4
= 4 x 2
= 8
[3]
[1]
[2]
[4]
merge
2
merge
2
[1,3]
[2,4]
merge
4.
[1,2,3,4].

1024 x 1024.

Number	len
1024 →	1 each
512 →	2 each
256 →	4 each
128 →	8 each
64 →	16 →
32 →	32 →
16 →	64 →
8 →	128 →
4 →	256 →
2 →	512 →
1 →	1024

Cost of merge is the same at each stage.

512 x 2. — 1024 ①
256 x 4. — 1024 ②.
128 x 8. — 1024 ③.
64 x 16. — 1024 ④.
32 x 32 — 1024 ⑤
16 x 64 — 1024 ⑥
8 x 128 — 1024 ⑦
4 x 256 — 1024 ⑧
2 x 512 — 1024 ⑨

⑨ x 1024 = n $\log_2 n$.