Abhishek Shah
Kushal Kale

Homework 1

P1 - i] If $f(n) = O(g(n))$ then $2^{f(n)} = O(3^{g(n)})$

$\Rightarrow$

False

Assume,

$f(n) = O(g(n))$ then $2^{f(n)} = O(3^{g(n)})$

Now, If $f(n) = O(g(n))$

then, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Consider the example,

$f(n) = 2n$

$g(n) = n$

Here,

$f(n) = O(g(n))$ (similarly for $\Omega(g(n))$ $f(n) \geq c \cdot g(n)$)

i.e $f(n) \leq c \cdot g(n)$

$2n \leq 2 \cdot n$ (considering $c = 2$) is true

But, for $2^{f(n)} = O(3^{g(n)})$

$2^{2n} \leq c \cdot 2^{n}$ (for any constant c)

$\Rightarrow$ is not true.

Contradiction.

Therefore, False

ii] If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then

$$f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$$

$\Rightarrow$ **True**

There exists +ve constants c_1, c_2, n_1, n_2 such that,

$f_1(n) = O(g_1(n))$

i.e $\forall\ c_1 > 0,\ f_1(n) \leq c_1 \cdot g_1(n) \quad \forall\ n \geq n_1$

$f_2(n) = O(g_2(n))$

i.e $\forall\ c_2 > 0,\ f_2(n) \leq c_2 \cdot g_2(n) \quad \forall\ n \geq n_2$

To prove this, we need a constant c_3 that causes

$f_1(n) + f_2(n) \leq c_3 [g_1(n) + g_2(n)] \quad \forall\ n > 0$ integers

$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$

$f_1(n) + f_2(n) \leq \max\{c_1, c_2\} (g_1(n) + g_2(n))$

$f_1(n) + f_2(n) \leq c_3 [g_1(n) + g_2(n)]$

i.e $c_3 = \max(c_1, c_2)$ satisfies the defn of Big-O

$\rightarrow f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

Hence, proved.

iii] If $f_1(n) = O(g_1(n))$ and $f_2(n) = o(g_2(n))$ then

$f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

⟹ **False**

Consider,

$f_1(n) = O(g_1(n))$ and $f_2(n) = o(g_2(n))$

$f_1(n) \leq c \cdot g_1(n)$ $\quad \lim_{n \to \infty} \frac{f_2(n)}{g_2(n)} = 0$

Consider,

$f_1(n) = n$ $\quad f_2(n) = \log n$

$g_1(n) = 2n$ $\quad g_2(n) = n$

$n \leq c \cdot 2n$ $\quad \lim_{n \to \infty} \frac{\log n}{n} = 0$

(L'Hopital)

Here,

$f_1(n) = n$ $\quad f_2(n) = \log n$

$g_1(n) = 2n$ $\quad g_2(n) = n$.

$\therefore f_1(n) + f_2(n)$ $\quad g_1(n) + g_2(n)$

$n + \log n$ $\quad 2n + n$

$\approx n$ $\quad \approx n$.

$f_1(n) + f_2(n) \notin o(g_1(n) + g_2(n))$

$f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$

Therefore, False

iv] $f(n) = \Theta(f(\frac{n}{2}))$

⟹ **False**

Assume,

$$f(n) = \Theta(f(\tfrac{n}{2}))$$

then,

$f(n) = O(f(\frac{n}{2}))$ and $f(n) = \Omega(f(\frac{n}{2}))$

Consider,

$f(n) = 2^n$ then,

$f(n) \leq c \cdot 2^{n/2}$ and $f(n) \geq c \cdot 2^{n/2}$

$2^n \leq c \cdot 2^{n/2}$ and $2^n \geq c \cdot 2^{n/2}$

$2^n \leq c \cdot \sqrt{2}$ and $2^n \geq c \cdot \sqrt{2}$

False and True

⇓

False.

contradiction.

Therefore False.

P-2]

$\Rightarrow$ (least complex)

$\frac{1}{n}$

$\frac{1}{1000}$, 10^{100}

$\log \log n$

$\sqrt{\log n}$

$\log n^2$, $\log^2 n$

$\log n$, $\log_{10} n$

$\sqrt{n}$

$n^{2/3}$

n, $n + \log n$, $\log 2^n$, $2^{\log n}$

$n \log n$

$n^{3/2}$

n^2, $n + \frac{n^2}{10^{20}}$

2^n, 2^{n+1}

$n \cdot 2^n$

3^n

$n!$, $(n+1)!$

2^{2n}, 4^n

n^n

(Most complex)

P3]

⇒ We needed to design a $o(n)$ algorithm that determines which number is present twice in the array.

Now,

for $o(n)$,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

$$\lim_{n \to \infty} \frac{f(n)}{n} = 0 \quad \text{---- (1)}$$

∴ We need $f(n)$ such that $c_q n^1$, will satisfy the necessary condition.

We can take $f(n)$ as $\log n$

$$\left[\lim_{n \to \infty} \frac{\log n}{n} = 0 \quad \text{(L'Hopital)} \right]$$

Remember,

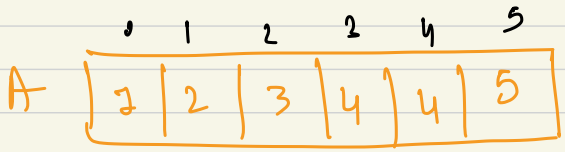
we need to find the duplicate number and that too in sorted array.

The efficient method will be to use Binary Search which have a complexity of $O(\log n)$.

Logic: If a number "n" is repeating, then it must be at a index "i" in the array.

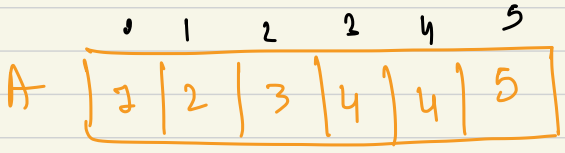
eg: for sorted array A,

A = { 1, 2, 3, 4, 4, 5 }



So technically, we just need to find a number whose value is same as its index since it is a sorted array.

for A,



Number 4 is repeating which can be found at index position 4.

i.e value = index.

We'll use Binary search for the same.

- **Algorithm**

oneDup (arr, low, high)

1. If low > high ⇒ return -1
2. mid = (low + high) / 2
3. Check if mid number the repeating one
4. If mid not in position ⇒
5. repeating number is in left.
6. If mid at proper position ⇒
7. repeating number is in right

Complexity :

Array of size n

1 comparison	About $n/2$ numbers left
2 comparison	About $n/4$ numbers left
3 comparisons	About $n/8$ numbers left
⋮	⋮
n comparisons	$\frac{n}{2^i}$ left

For comparisons till last step,

We have to find value of i when

$\frac{n}{2^i}$ value is 1

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

$$n = 2^i$$

Taking log on both sides

$$i = \log_2 n$$

$$\boxed{O(\log_2 n)}$$

P-4].

⟹ We look for stable matchings between two given disjoint sets and their preferences lists.

We first find a stable matching by keeping group 1 as the askers and group 2 as the respondents.

In this case, Gale Shapley algorithm (match) will return the stable matching which is optimal for group 1 (askers).

We then find a stable matching by putting the group 2 as askers.

In this case, we get a matching which is optimal for group 2 now.

If we get same matching for both runs, it means that we have only one matching which is optimal for both groups and we print ("No")

If we get two different matching for both runs, it means that multiple stable matching exits and we print ("Yes")

Algorithm :

1] Find inverse of list of pref of group 2

```
for i in range (len( list 2))   → O(n)
    for j in list 2 [i]         → O(n)
                                  ---------
                                  O(n^2)
```

2] Since, we need the original list further in the algo, we copy it (list 1).

```
for i in range ( len( list 1 )   → O(n)
    for j in list 1 [i]          → O(n)
                                   ---------
                                   O(n2)
```

3] Calling gale Shapley (match) $\longrightarrow O(n^2)$

4] Find inverse of list of pref. of group 1 $\longrightarrow O(n^2)$

5] Call gale Shapley algo $\longrightarrow O(n^2)$

6] Invert the match list given instead of gale Shapley to find group 1 & group 2 pairs

for i in range(len(match_2)) $\longrightarrow O(n)$

7] Check if same match and o/p required result $\longrightarrow O(1)$

Total complexity = $O(n^2)$

P5]

⟹

Random floating point numbers with uniform distribution in range $[0, 1)$

Algorithm	Input size (n)			
	100	1000	10000	100000
	Time to run (seconds)			
Merge sort	0.0	0.008	0.101	1.090
Insertion sort	0.0	0.005	12.810	Took more than 3 minutes
Bucket sort	0.0	0.005	0.014	0.200

[Time is represented in seconds (floating pt)]

Merge sort	$O(n \log n)$
Insertion sort	$O(n^2)$
Bucket sort	$O(n)$

As we can see in the table,

1] Bucket sort performs better, no matter the size of the input.
This is because, Bucket sort is mainly useful when input is uniformly distributed over a range. Its time complexity is $O(n+k)$, where k = no of buckets is always a constant.
So, $O(n+k) \approx O(n)$

2] This is followed by the Merge sort which has a overall complexity of $O(n \log n)$.
Even though it performs considerably better for large data, it still gets outperformed by the Bucket sort (due to uniformly distributed data)

3] Insertion sort has the worst time complexity out of all three due to the fact that it takes $O(n^2)$ to run any algorithm.

So, it would take forever to run for a input as large as 1000000.

" Also, we can see that as input size increases, the time to run the algorithm also increases based on their time complexity."

Random floating point numbers with a gaussian distribution with

$\mu = 0.5$ and $\sigma = \frac{1}{10000}$

Algorithm	Input size (n)			
	100	1000	10000	100000
	Time to run			
Merge Sort	0.00	0.002	0.024	0.288
Insertion Sort	0.00	0.030	2.25	Took more than 3 minutes
Bucket Sort	0.00	0.020	0.330	2.841

[Time is represented in seconds (floating point)]

Merge sort : $O(n \log n)$

Insertion sort : $O(n^2)$

Bucket sort : $O(n)$

In gaussian distribution, data is not uniformly distributed. It depends on factors such as mean and standard deviation.

In such cases i.e when data is not uniformly distributed, Merge sort completely outperforms the Bucket sort as we can see from the table above.

Bucket sort does better but is still far behind from the Merge sort in terms of time taken to run the algorithm.

Insertion sort is still the worst performing algorithm, great due to the fact that its overall time complexity is $O(n^2)$.

" Also, we can see that as input size increases, the time to run the algorithm also increases based on their time complexity. "