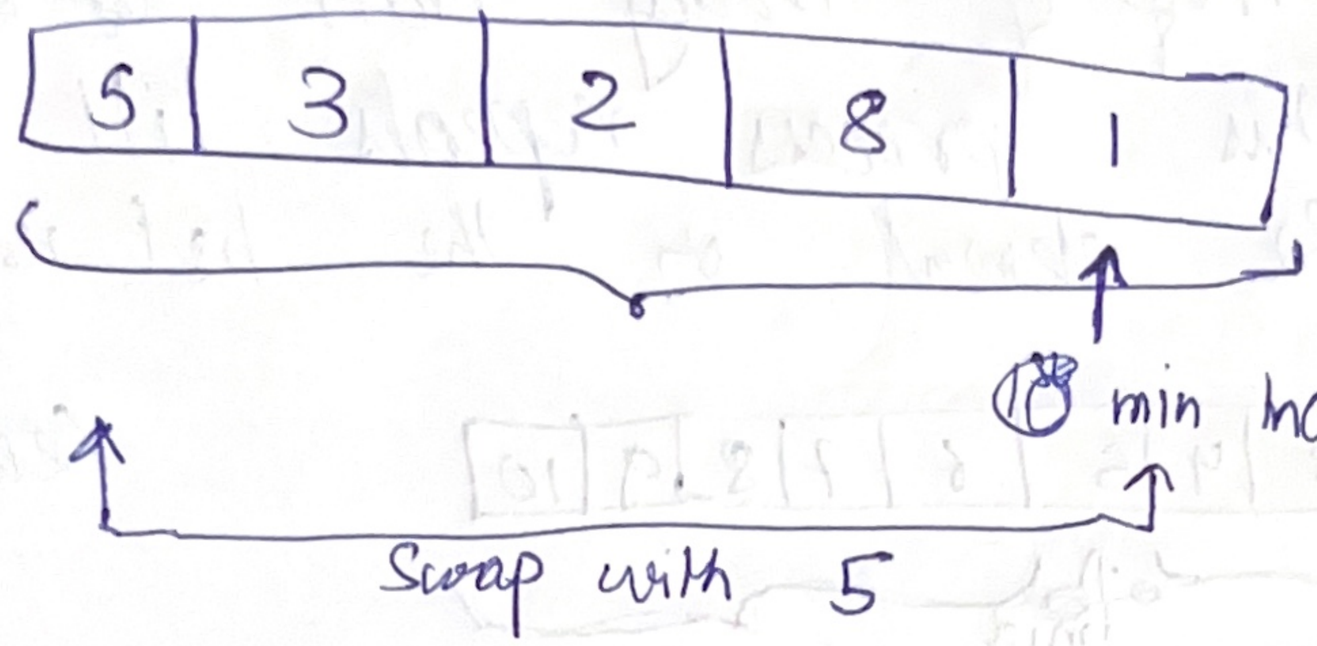
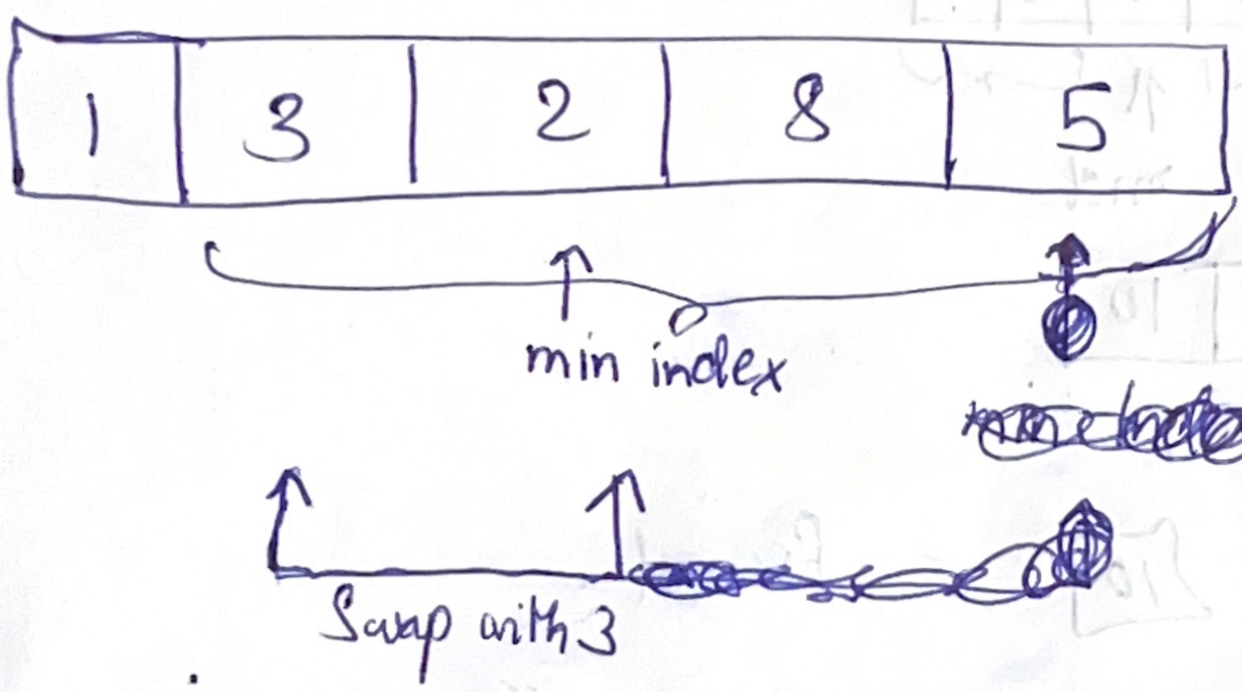
⇒ An algorithm which has $O(n^2)$ time complexity is 'SELECTION SORT'. It picks a minimum index in each pass over the array & ~~picks~~ swaps it towards the back.

eg:



← 1st iteration



← 2nd iteration

:

and so on

We see here that we need 2 loops to to implement this solution:

1st - For the number of passes

2nd - To find min index in each pass

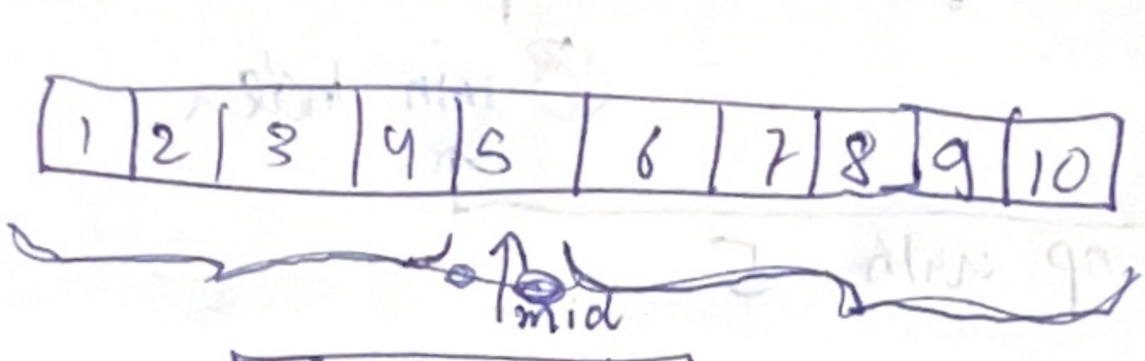
∴ The for an array of size 'N' the inner loop runs for $N-1, +N-2 \cdots +1$ times

$1+2+3 \cdots N-1 = \frac{N(N-1)}{2} = \frac{N^2-N}{2}$

∴ Time complexity: $O(N^2-N/2) = \underline{O(N^2)} \rightarrow$ Ans

2) Binary search works on a sorted array by ~~looking~~ dividing the array into 2 halves & identifying which half the element belongs to. If it is in the upper half it discards the lower half of array until or vice versa. This process repeats till we find the element or the list is exhausted

eg



Search = 10

Here we see that the list gets halved in each step. Thus, if we have a list of size 'N' then in each step list becomes $N/2$.

1st iteration $= N/2$
2nd " $= N/2^2$
3rd " $= N/2^3$
⋮
kth " $= N/2^k$

Let k be a value such that $N/2^k <= 1$. This 'k' is the maximum number of steps the algorithm will take

$N \approx$ Complexity $= O(k)$

Now,

$$\frac{N}{2^k} \approx 1$$

$$\Rightarrow N \simeq 2^k$$

$$\Rightarrow \log_2 N \simeq k$$

$$\Rightarrow \therefore O(k) = \underline{O(\log N)}$$

$\therefore$ Complexity = $\underline{O(\log N)}$ $\longrightarrow$ Ans

ipython.ipynb ● pagerank.ipynb ●

ipython.ipynb > %timeit 6*12

+ Code + Markdown | Run All Clear Outputs of All Cells Restart Interrupt | Variables Outline ...

```python
def binary_exponent(a,b):
    """
    This function performs binary exponentiation iteratively
    """
    a_pow=a                    # This variable stores the power of 'a' in a particular iteration

    #1st iteration - Here the power of a is 1
    if b%2==1:
        result = a
    else:
        result = 1
    b = b//2

    while b>0:
        rem = b%2
        b = b//2
        a_pow = a_pow*a_pow  # Power of a increases in powers of 2 in every iteration
        if rem==1:           # if remainder is 1, then multiply the result with current iteration of a_pow
            result = result*a_pow


    return result
```

[1] ✓ 0.5s

```python
binary_exponent(6,12)
```

[2] ✓ 0.1s

```
2176782336
```