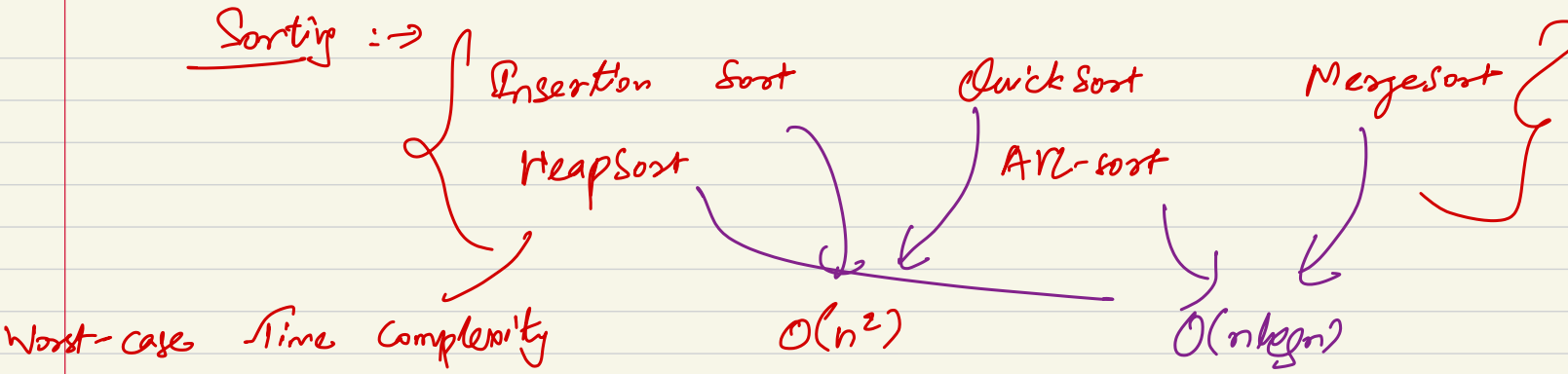


Can we do better than O(nlogn) ?

If Pine some addⁿ info, possibly

But what about the general case?

n elements → arbitrary

Can't do better than O(nlogn)

Why? Let's look at our sorting algs.

Comparison based Sorting

They depend on Comparison oper^n for sorting

n elements → ADTs (black boxes)

Suppose the only oper^n you can do is comparison ($<, \leq, =, \geq, >$)

↓

Let's assume that we only charge for comparisons

yes no

⇒ No. of comparisons can help establish a Lower bound.

↓

$\Omega(n \log n)$

To prove

Searching

given an array A [n]
n elements.

You have a key 'x'

Search x in A.

→ Sorted

Binary Search : Main operation

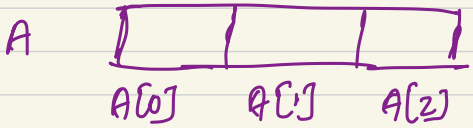
Comparison ($x < A[i]$)?

yes no

Comparison based model

→ lower bound

$\Omega(\log n)$

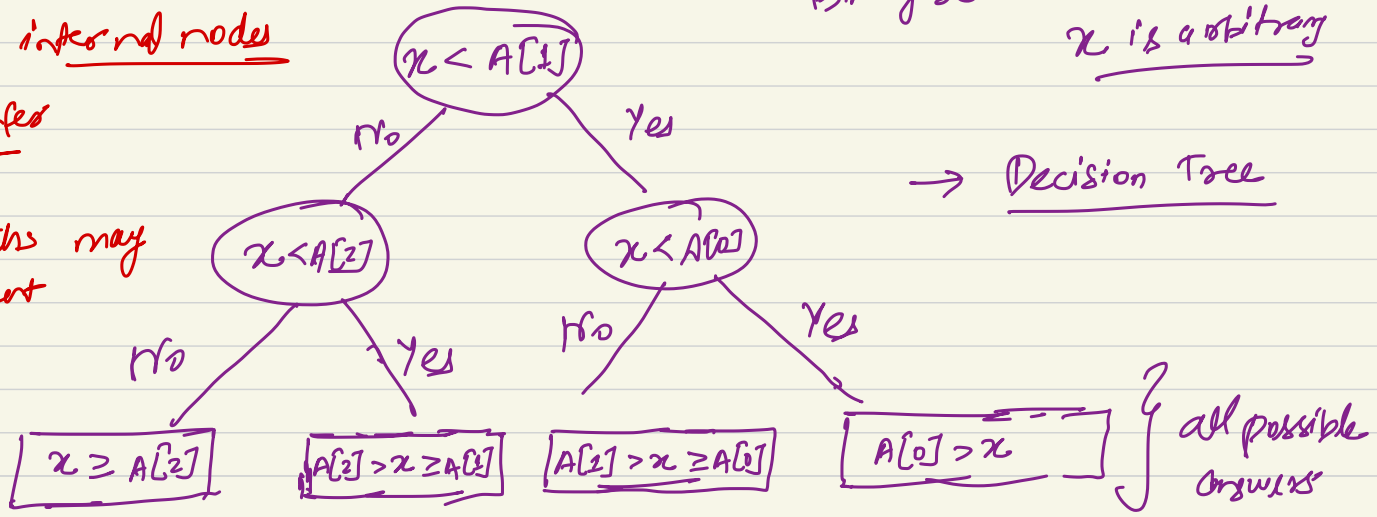


Binary Search

x

Binary Search

x is arbitrary



→ Decision Tree

* In general, internal nodes may differ
* Path lengths may be different
* What'll not change?
  → # of possible answers.
  ↓
  # of leaf nodes = n+1

Decision Tree

internal node

leaf node

root-to-leaf path

length of root-to-leaf path

longest path from root to leaf ⇒ height of the tree

Algorithm

One Comparison operation

Answer found

algorithm execution for a part^r i/p

running-time (for a part^r i/p)

Worst-case running time

Binary Tree with $(n+1)$ leaf nodes

$\Rightarrow$ Height $= \Omega(\lg n)$

Worst case running time for search in a comparison based model is $\Omega(\lg n)$

What about sorting

$A[1 \ldots n]$

Searching

Internal node

$x < A[i]$

Sorting

$A[i] < A[j]$

Yes

No

leaf node

$A[i] < x \leq A[i-1]$

$n+1$

$A[i] > x \geq A[i-1]$

$A[5] \leq A[3] \leq A[7] \; - \; - \leq A[2]$

final sorted order

leaf nodes $= n!$

Worst-Case running time for a sorting algo in comparison based model?

$\Omega(\log n!)$

$= \Omega(n \log n)$

Sorting $O(n \log n)$ is the best you can achieve.

To do better than $O(n \log n)$

→ Youve to use more information → ??

[in some restricted cases]

Integer Sorting

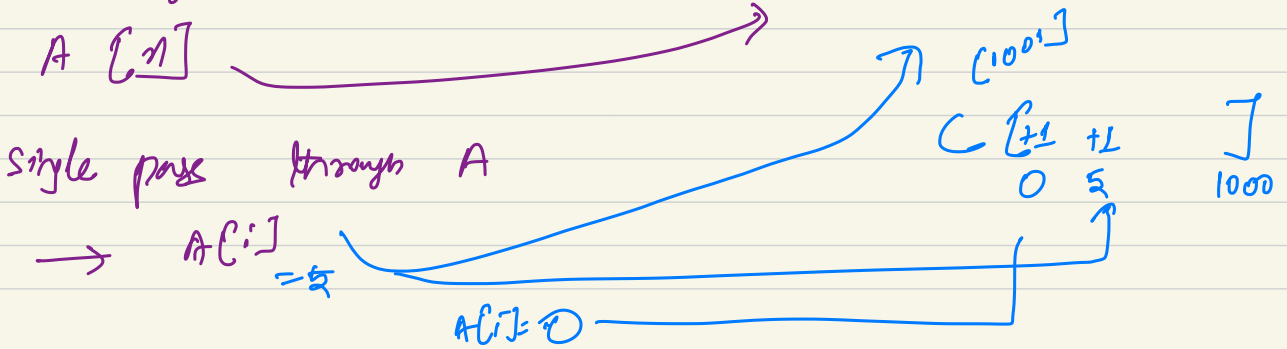
A[1] — . A[n] unsorted

→ range of values ~

min max

n integers in range [0 1000]

A[n]



for i = 1 → n → O(n)

C[A[i]]++;

for i = 0 → 1000 → O(K)

print i C[i] times; } → O(n) for all values

O(n)

O(n+K)

* All integers
* Range is known to you
  ↳ [0, K-1] where K = O(n)

⇒ Sorting algo that takes O(n) time

0 — 1000 K=1000

n = 10^6

Linear-time sorting

→ Counting-sort ←

05/11/21

Wed

- lower bound — $\Omega(n \log n)$

for comparison-based sorting

↓

Linear-time sorting

- Integers
- Range = $O(n)$ → $O(n)$-time sorting

$n = 2$

range 10^6 ✗ $O(n+k)$

↓

range

Subtle Point :⇒ Generally, when we have to sort items, these may have multiple fields

Spreadsheet

Fields	Name	Age	Address	Marks
	Bishal			53
Sort : sort a key	Rohan			34
	Mridul			45
Ex. sort Marks	Mohan			34

What do you do with items that have the same value in that key? → Marks

✗ Pick up another field & based on that you sort. ✗

✓ Put them in the same order as the i/p array

Rohan	34
Mohan	34
Mridul	45
Bimal	53

Any sorting algo that ensures that it's a stable sorting algo

- Stable Sorting :-> A sorting that ensures that for keys with the same value, the order of the keys in the o/p array will be the same as in the i/p array, is called a stable sorting algo.

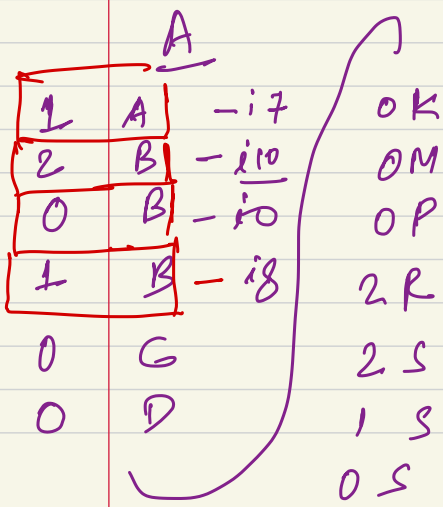
Ex. of an algo that is not stable :->

Quick Sort ✗

Counting Sort → stable sorting

I/p: An array A of n elements, with multiple fields. They need to be sorted wrt one field - "key"

Assume that the values in 'key' lie between 0 to K-1 & are all integers K = O(n)



* Arrays of linked list
* Let's reuse our algo

Initialize an array C[K]

13 elements

Key [0, 2]

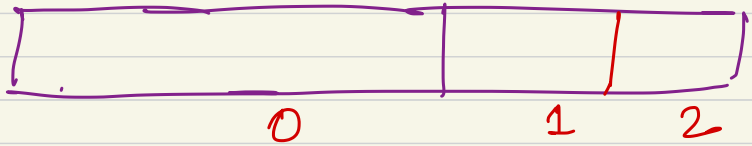
C[i] stores # times i appears in A

C

7	3	3
0	1	2

we want to write o/p in array B.

B



Make a pass through A for each $j = 0 \to K-1$

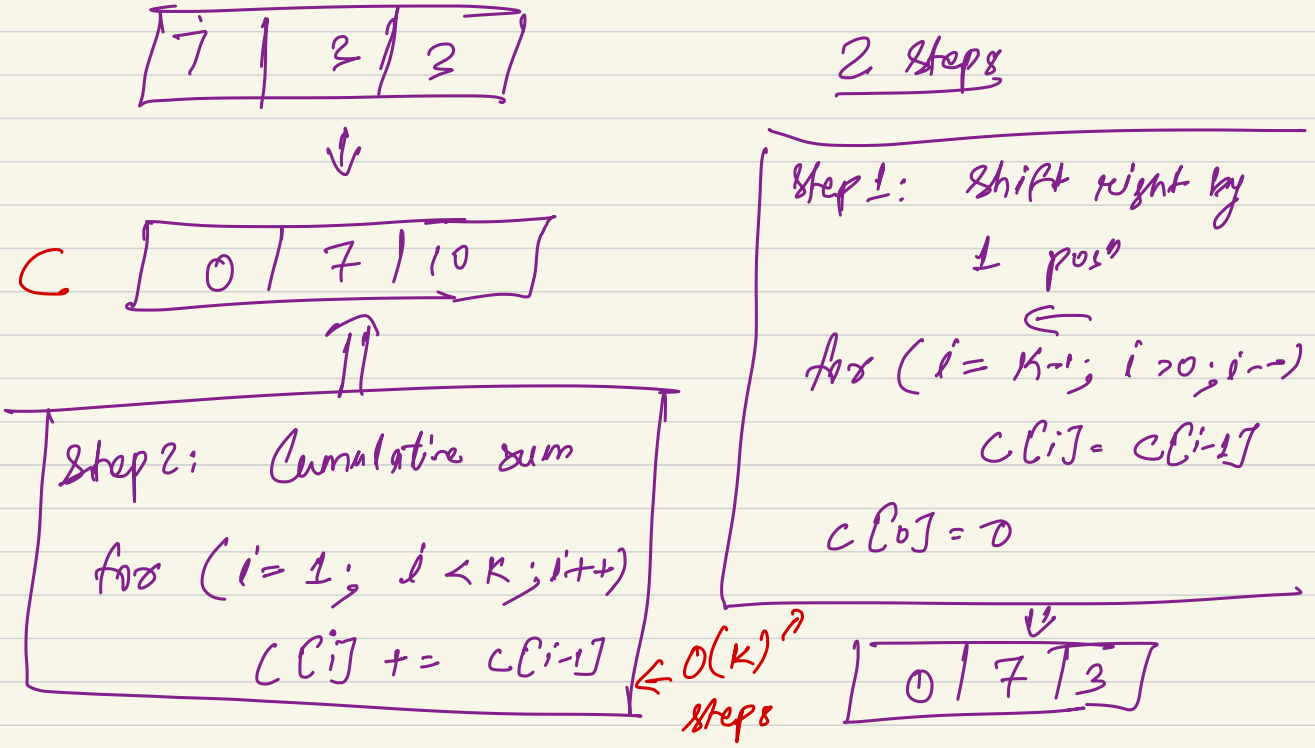
$O(Kn)$

Idea :- Modify C such that

C[i] stores what is the index in B where the next element in A with value i go to.

Ideal C in the beginning

0	7	10



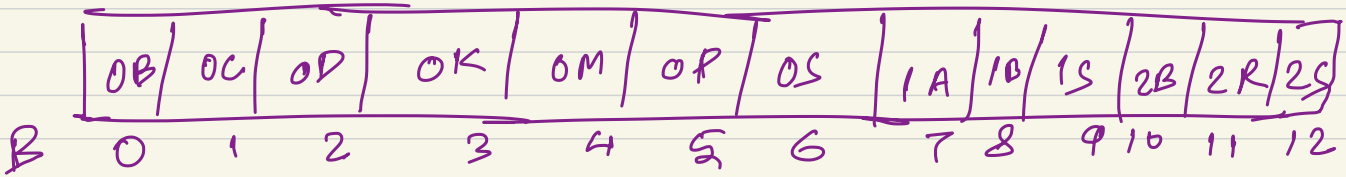
When we traverse A, if we find a value P, after this has been copied to B,

C[i]++;

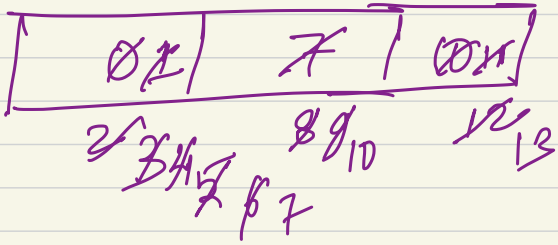
A

1A	2B	0B	1B	0C	0D
0K	0M	0P	2R	2S	1S	0S

B



C



Stable counting sort

$O(n+k)$

* In-place sorting algo?
→ No

n Integers range $[0, K-1]$ $K = O(n^2)$

Can you still get an $O(n)$- stable alg?

Yes

Reuse the same ideas

A is an array of n integers

range 0 to n^2-1

Each integer a can be written as

$$a = a_1 n + a_0$$

$$a_1 = \lfloor a/n \rfloor \qquad a_0 = a \% n$$

$a_0, a_1 \in \{0, \ldots n-1\}$

$a = \{a_1, a_0\}$

a_1 ↙ MSB

a_0 ↘ LSB

To sort all integers

→ Sort acc. to a_0 (LSB) [Counting Sort] ✓

→ Sort acc. to a_1 (MSB) [Counting Sort]

What if I reverse

a_1 ⟵

a_0 ⟶

70

37

✗

n elements $\quad 0 \rightarrow n-1$

Ex $n = 10$

$a_1\ a_0$	a_0	a_1
52	30	07
33	41	17
07	52	27
41	62	30
62	33	33
27	75	41
75	07	47
17	27	52
47	17	62
30	47	75

Sorted

Complexity: → Running Counting Sort twice

$O(n+k) = O(n)$

$k = O(n)$

Generalize this idea

$k = O(n^d)$ d is a constant integer

→ Counting Sort d times

⇒ $O(d(n+k)) = O(dn)$

Radix Sort

As long as d is comparable to $\log n$, it is fine.

$n = 10$

999

9 9 9

$9 \times 10^2 + 9 \times 10^1 + 9$

$n^{d-1} + n^{d-2} \ldots$

Counting Sort
Radix Sort

$O(n)$
$O(dn) \rightarrow O(n)$ for constant d

→ Another algo - that can go beyond integers

Expected $O(n)$ complexity

Worst Case $O(n^2)$

Bucket-Sort

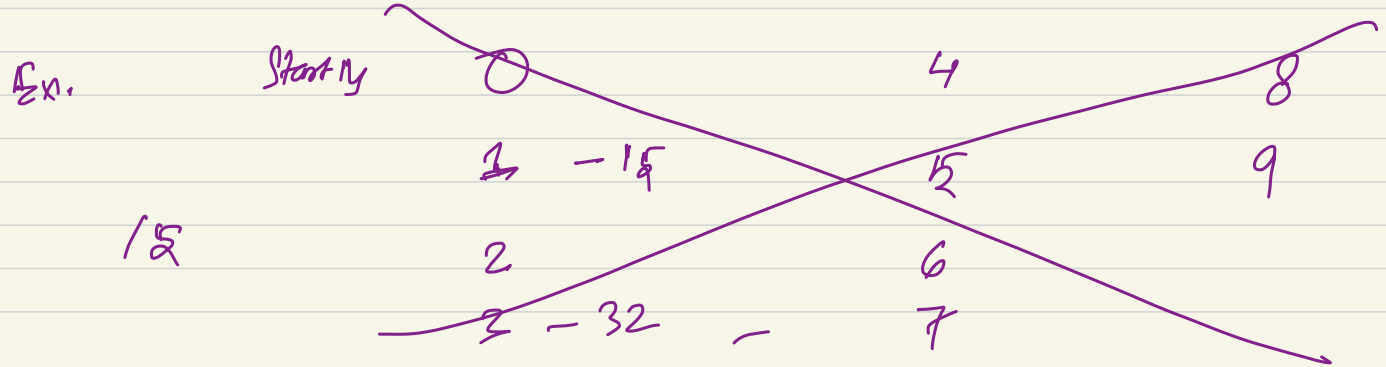
Take your i/p

n elements

Coming from some distn

If you can somehow divide your i/p into n buckets

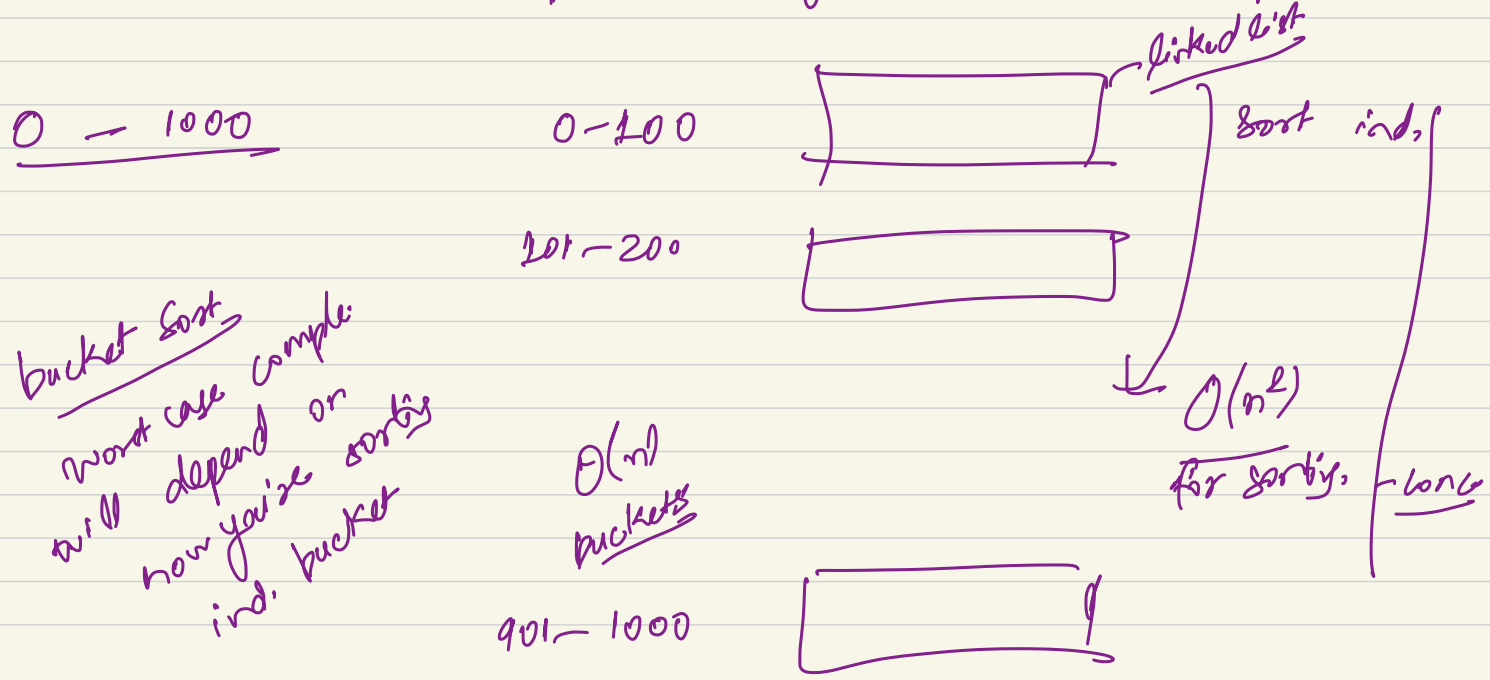
such that each bucket is expected to have O(1) elements



→ f^n that maps i/p to these buckets

Sort each bucket individually

Concatenate the o/ps to get the final o/p.



{ expected O(n) }