Sorting

1. Why Sorting?
2. Algorithm design strategies
    a. Brute force (Monkey Sort, Selection Sort, Bubble Sort)
    b. Decrease-and-conquer (Insertion Sort)
    c. Divide-and-conquer (Merge Sort, Quick Sort)
    d. Transform-and-conquer (Tree Sort, Heap Sort)

3. Linear-time sorting (Radix Sort, Counting Sort, Bucket Sort)

Why Sorting

Searching for a value is faster in a sorted array than an unsorted array.

Can find duplicates easily using sorting (all items with equal value appear in consecutive positions).

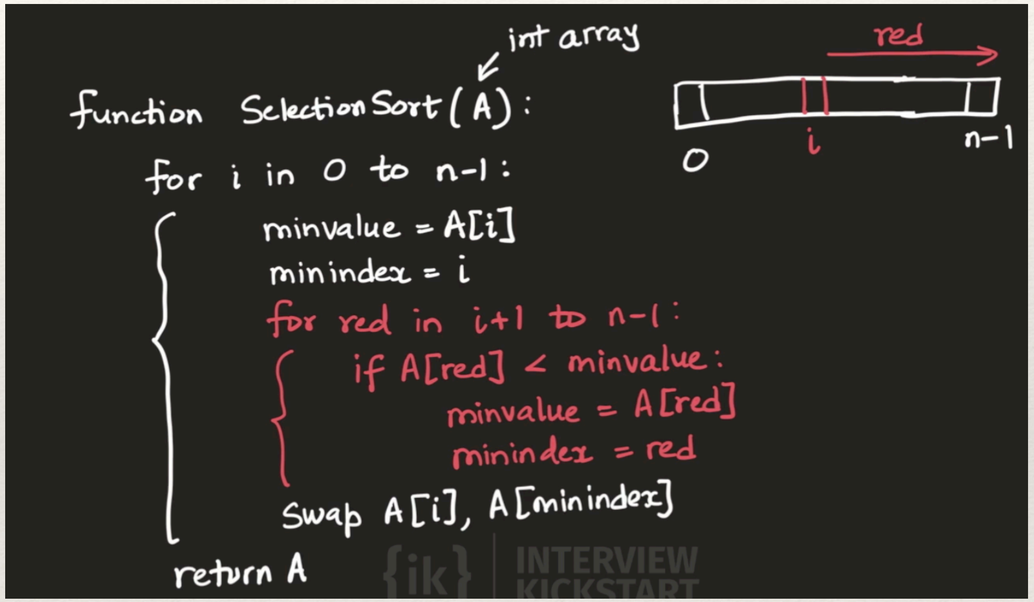
Matching items in two or more files.

Can find median and top k values quickly.

The truncated top of an immense sorted list is the universal UI.

Selection Sort

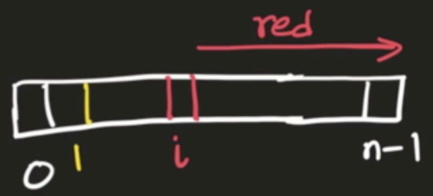
```
function selectionsort(A):  //array A[1..n]
    for i in 1 to n:
        #Find the i th smallest element and swap it
with A[i]
        min = i
        During a linear scan of A[i..n]:
            if you find a smaller element than
A[min]:
                Update the index of the min
element
        Swap the min element with A[i]
```



Analysis of Selection sort

int array

```
function SelectionSort(A):
    for i in 0 to n-1:                          → c8 n
        minvalue = A[i]                         ← c1      c1 n
        minindex = i                            ← c2      c2 n
        for red in i+1 to n-1:
            if A[red] < minvalue:               ← c3
                minvalue = A[red]               ← c4
                minindex = red                  ← c5      n(n-1)/2 * c9
        swap A[i], A[minindex]                  ← c6      c6 n
    return A                                    ← c7      c7
```



As $n \to \infty$

$$\frac{bn+c}{an^2} = \frac{b}{an} + \frac{c}{an^2}$$

$\downarrow$ 0 $\quad$ $\downarrow$ 0

$$T(n) = c_8 n + c_1 n + c_2 n + \frac{n(n-1)}{2} c_9 + c_6 n + c_7$$

$$T(n) = \underbrace{an^2}_{\text{Dominant}} + \underbrace{bn + c}_{\text{Lower-order terms}}$$

$n(n-1)/2$

$$S = 1 + 2 + \dots + n-1$$
$$S = n-1 + n-2 + \dots 1$$
$$2S = \underbrace{n + n + \dots + n}_{n-1 \text{ times}}$$
$$= n(n-1)$$

When n tends to infinity, lower order terms are a very small factor of higher order terms.

- Drop the lower order terms
- To make the analysis system-independent, we also ignore the constant factors.

$$T(n) = \boxed{\Theta(n^2)} \qquad T(n) = \Theta(n)$$
$$\sim cn^2 \qquad \sim cn$$

Asymptotic Time Complexity

As $n \to \infty$, $T(n)$ grows in a quadratic fashion

INTERVIEW KICKSTART

Bubble Sort

```
                                  ← int array
function BubbleSort(A):
                                                  BUBBLE
    for i in 0 to n-1 :  ← C3                     SORT
    {
        for red in n-1 down to i+1 :  ← C2
            if A[red-1] > A[red] :
                swap A[red-1], A[red]
    }
    return A  ← C4
```

$C_3 * n$

$\frac{n(n-1)}{2} * C_2$

$\frac{n(n-1)}{2} * C_1$

C_1

#comparison operations

$= (n-1) + (n-2) + \dots 0$

$= n(n-1)/2$

{ik} INTERVIEW KICKSTART

Notes

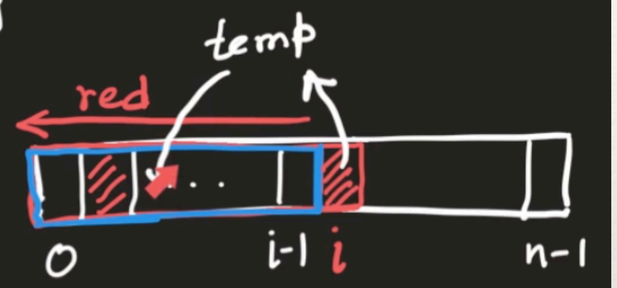
Algorithm Design Strategy

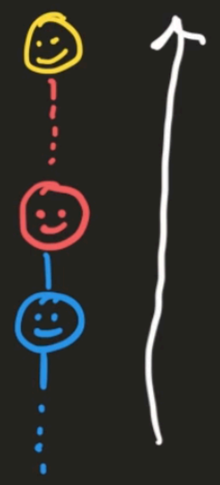
- **Brute Force** - one that comes to your mind directly based off problem statement. It is usually an exhaustive approach
- **Decrease and Conquer** - Decrease the problem of size n to n-1,
  - **Top-down -** Upfront work to nibble away one element, n -> n-1 (selection sort)
  - **Bottom up -** Work at the end to extend the (n-1)th solution, n-1 -> n, () (insertion sort)
- **Divide and Conquer**: Divide the problem into multiple smaller instances of same problem (most often 2). Combine the solutions to make solution to large problem
  - **Top-down -** Upfront work and then pass down to next level (quick sort)
  - **Bottom up -** Work at the end (merge sort merged solved subproblems at the end) (Although in case of merge sort we call bottom-up to a completely different non-recursive solution and recursive solution is called top down)
- **Transform and Conquer**: Transfer the representation and solve (heap sort - build heap and then remove one by one)
- **Enhance and conquer**: Omkar will cover during later part of the course

Insertion Sort

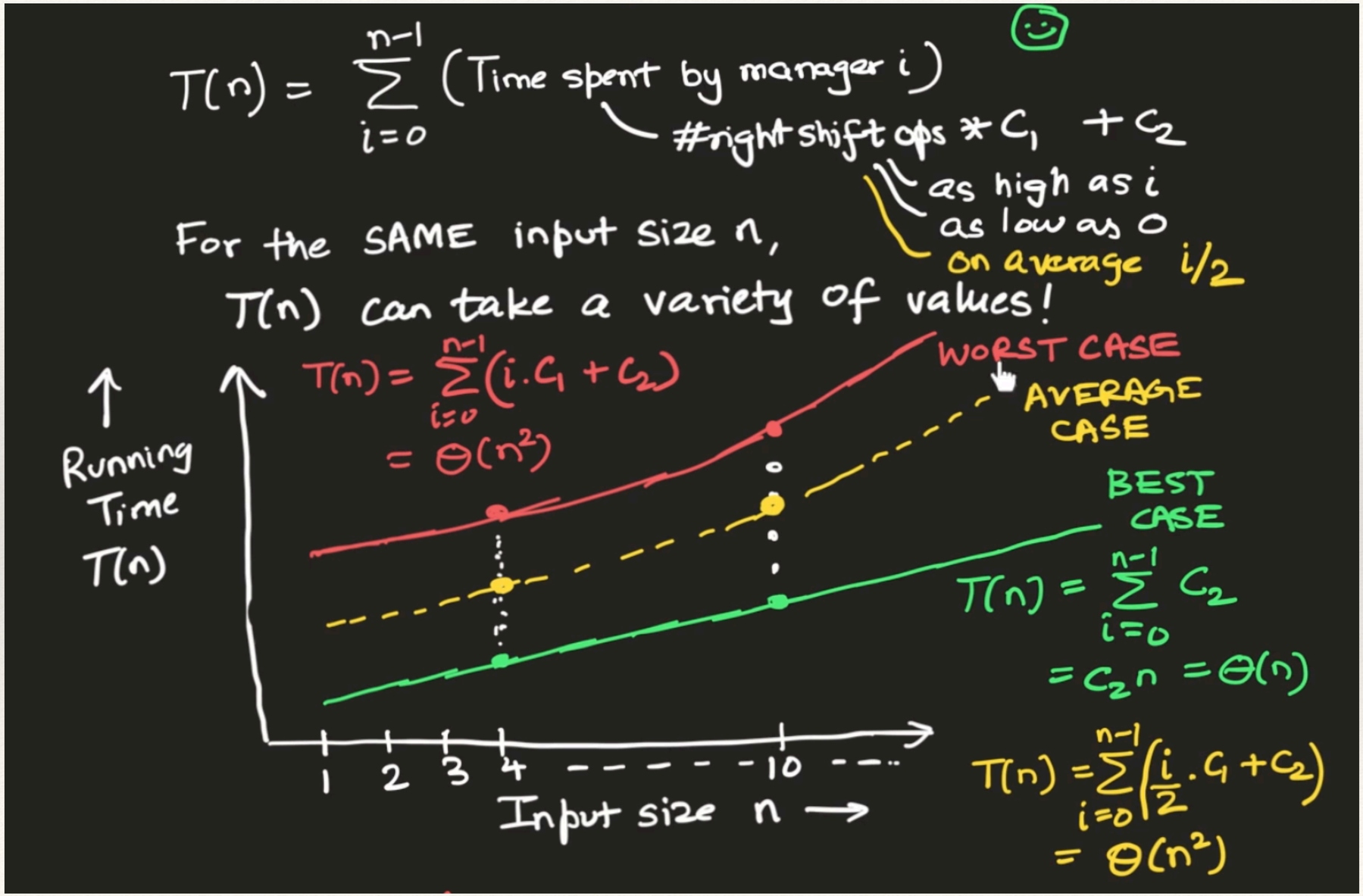
int array

```
function InsertionSort(A):
    for i in 0 to n-1:
        temp = A[i]
        red = i-1
        while red ≥ 0 and A[red] > temp:
            A[red+1] = A[red]
            red --
        A[red+1] = temp
    return A
```





Insertion sort analysis



Quiz

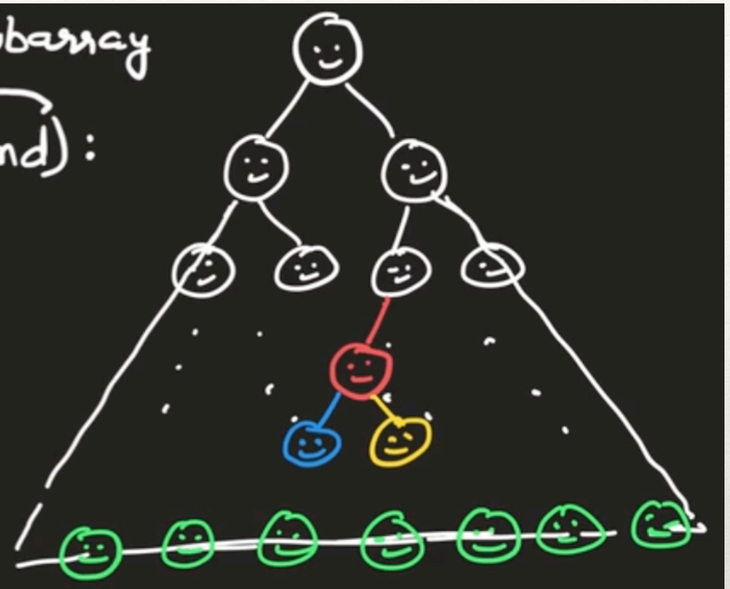
- **BogoSort**, also known as **Stupid sort**, is a sorting algorithm which just randomly shuffles the elements repeatedly (testing the array each time to check if it is now sorted) until they happen to land in the right order. The hope is that by doing this repeatedly, we will exhaustively try out all possible orderings, eventually hit the sorted order.

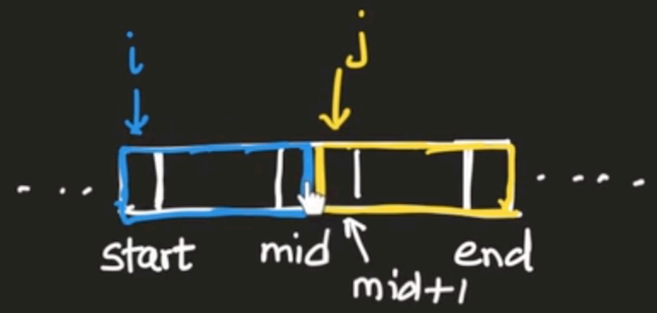
```
while not Sorted(list) do
  shuffle (list)
done
```

- In the best case, the very first permutation we start with could accidentally turn out to be the sorted permutation. Checking that the array given is already sorted takes O(n) time.
- In the worst case, this process could go on for an indefinite amount of time without hitting the sorted order.

Merge Sort

```
                    int subarray
function mergesort (A, start, end):
  # Leaf worker
  if start == end:
      return
  # Internal node worker
  mid = (start + end)/2
  mergesort(A, start, mid)
  mergesort(A, mid+1, end)
  //Merge the two sorted halves
  i = start, j = mid+1
  aux = an empty array of
        size end-start+1
```

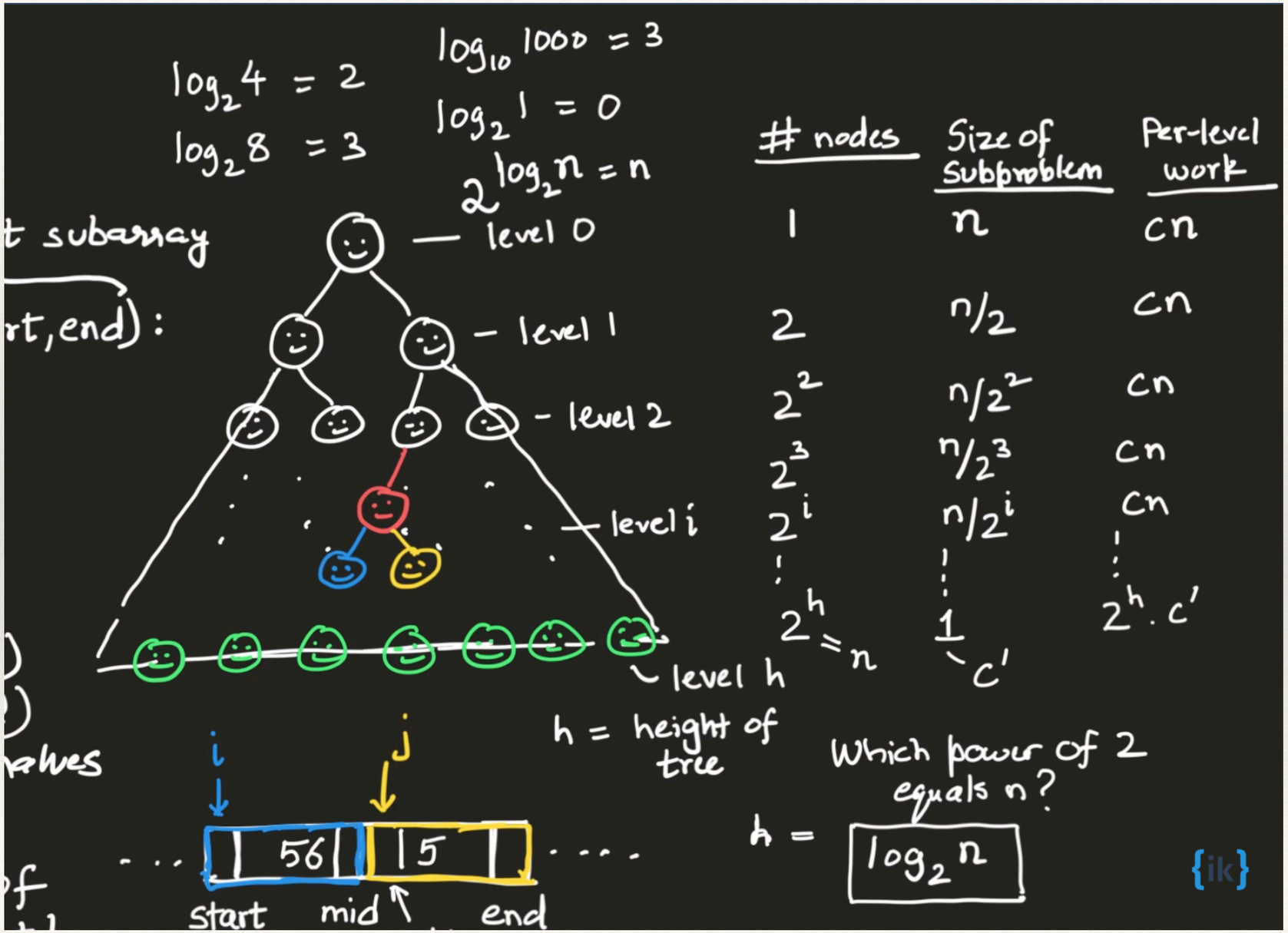





```
while i ≤ mid and j ≤ end:
    if A[i] ≤ A[j]:
        aux.append(A[i])
        i++
    else: // A[i] > A[j]
        aux.append(A[j])
        j++

// Gather phase
while i ≤ mid:
    aux.append(A[i])
    i++
while j ≤ end:
    aux.append(A[j])
    j++
A[start...end] ← aux
```

Merge Sort Analysis



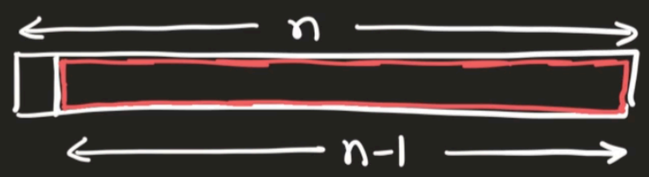
- amount of work done at each level multiplied by number of levels

$$T(n) = cn * \#\text{levels in tree}$$
$$= cn(\log_2 n + 1)$$
$$= cn\log_2 n + \cancel{cn}$$
$$= \Theta(n\log_2 n)$$
$$= \Theta(n\log n)$$

Big-Oh and Big Omega notations

- Best case analysis of merge sort, rather any sorting method could be improved to be O(n) using a hack to check whether array is already sorted.
- So best case analysis is not of much use as it could be hacked
- Similarly, average case has to be found out or analyzed
- So usually worst case complexity is considered to be complexity of an algorithm as this is the stronger guarantee whatever input is passed to the algorithm
- Bog-oh notation represents upper bound of running time or the worst case running time
- θ representation of running time represents actual asymptomatic representation of running time and sometimes that can be complicated and then Bog-oh can represent a simplified upper bound of θ equation
- For O(n^2) even O(n^3) is another upper bound but we choose O(n^2) over O(n^3) as it is both mathematically simple and most tightly bound upper bound to the θ(running time)
- Similarly a simplified lower bound of θ(running time) is called Ω(running time) is used which is called big-omega notation
- If the running time of an algorithm is Θ(n), that means it roughly runs in time cn for some constant c. i.e, it runs in linear time. If the running time of an algorithm is O(n), that means it runs in time at most cn for some constant c. i.e, it runs in "at most" linear time. This means if T(n) = Θ(n^2), then T(n) is also O(n^2).

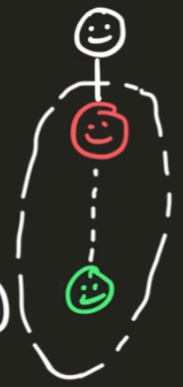
Recurrence equation



$$T(n) = cn + \underline{T(n-1)}$$

Recurrence equation

$$\underline{T(n-1)} = c(n-1) + T(n-2)$$



$$\begin{aligned}
T(n) &= cn + c(n-1) + T(n-2) \\
&= cn + c(n-1) + c(n-2) + T(n-3) \\
&= cn + c(n-1) + c(n-2) + c(n-3) + \ldots + T(0) \\
&= c[\underbrace{n + n-1 + n-2 + n-3 + \ldots 1}] + \\
&= \Theta(n^2)
\end{aligned}$$

$$1 + 2 + \ldots n-1 + n$$
$$= \frac{n(n-1)}{2} + n$$
$$= \frac{n(n+1)}{2}$$

$$T(n) = T(n-1) + cn$$

{ik}

Growth rate of different running times

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
$\Theta(1)$	any	any	any	any
$\Theta(\log N)$	any	any	any	any
$\Theta(N)$	millions	tens of millions	hundreds of millions	billions
$\Theta(N \log N)$	hundreds of thousands	millions	millions	hundreds of millions
$\Theta(N^2)$	hundreds	thousand	thousands	tens of thousands
$\Theta(N^3)$	hundred	hundreds	thousand	thousands
$\Theta(2^N)$	20	20s	20s	30

X

Merge sort running time

Insertion Sort running time

$c\, n \log n > c' n^2$ (worst-case comparison)

for small n, since $c \gg c'$

TimSort

Combines merge-sort and insertion-sort for worst-case $\Theta(n \log n)$ time and best-case $\Theta(n)$ time.

Timsort has been Python's standard sorting algorithm since version 2.3

Tim Peters

Ideal sorting algorithm

The ideal sorting algorithm would have the following properties:

- Stable: Equal keys aren't reordered.
- Operates in place, requiring O(1) extra space.
- Worst-case O(n·lg(n)) key comparisons.
- Worst-case O(n) swaps.
- Adaptive: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.

There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

Quick sort analysis

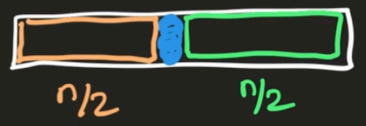
DIVIDE – cn

SOLVE – Trivial

COMBINE – Trivial

$$T(n) = cn + T(\text{left partition}) + T(\text{right partition})$$

If pivot is median, the split will be even



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

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

$$T(n) = \Theta(n \log n)$$

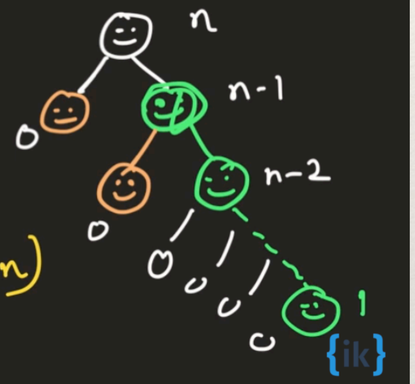
If pivot is the smallest / largest element, the split will be extremely skewed.

$$T(n) = cn + T(n-1)$$

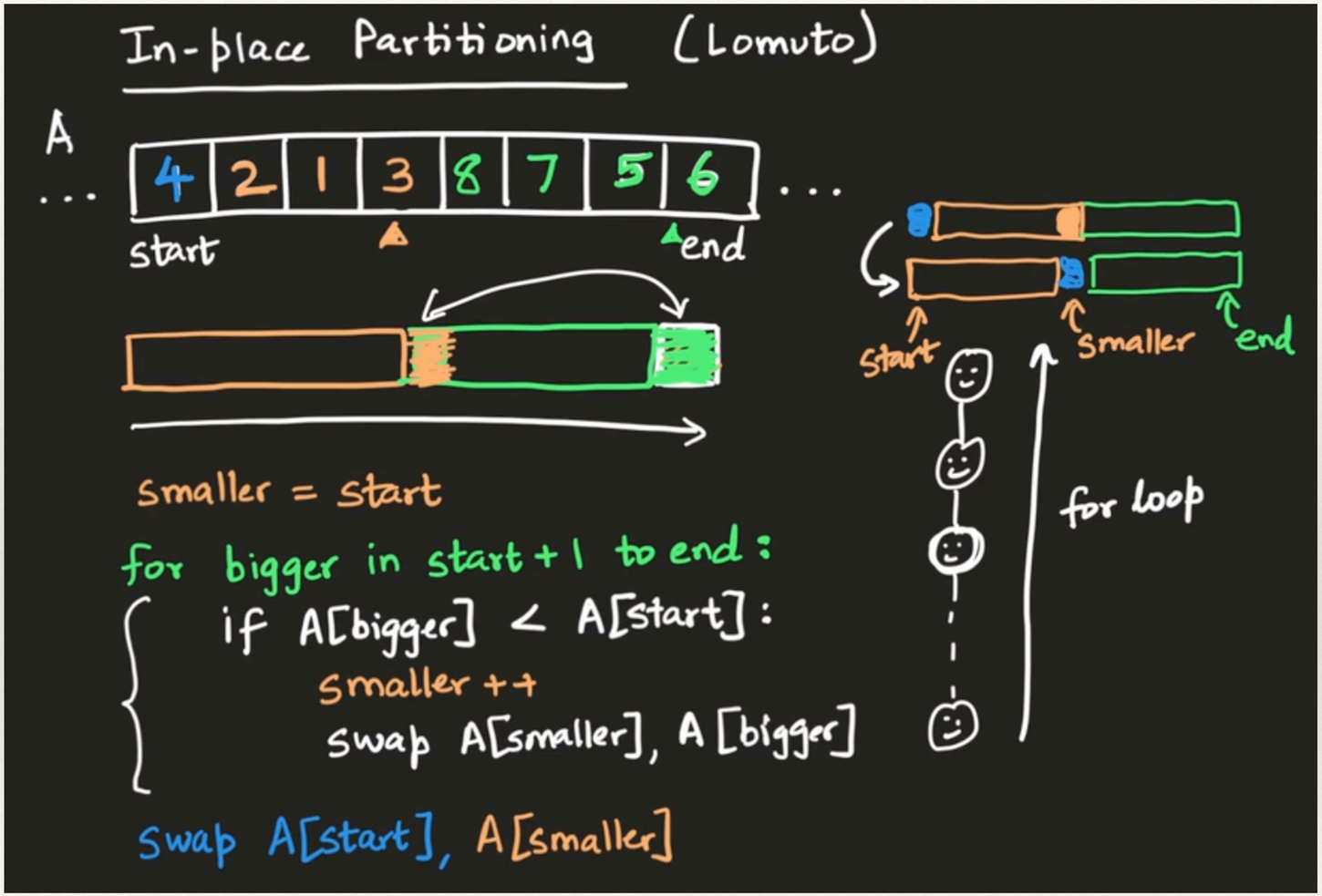
$$T(n) = \Theta(n^2)$$

WORST CASE

AVERAGE CASE = ? $\Theta(n \log n)$

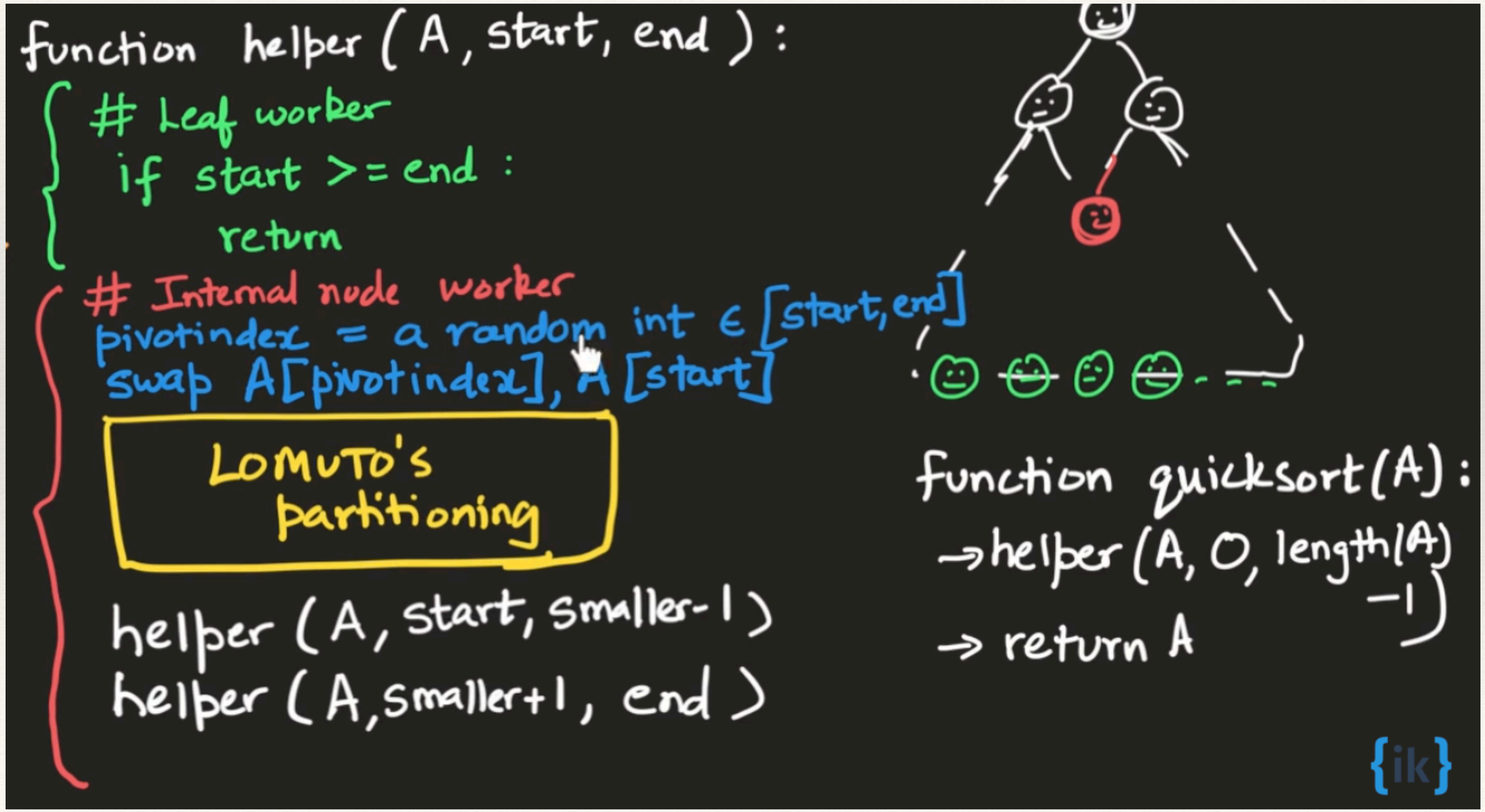


Quick sort Lomuto's partitioning

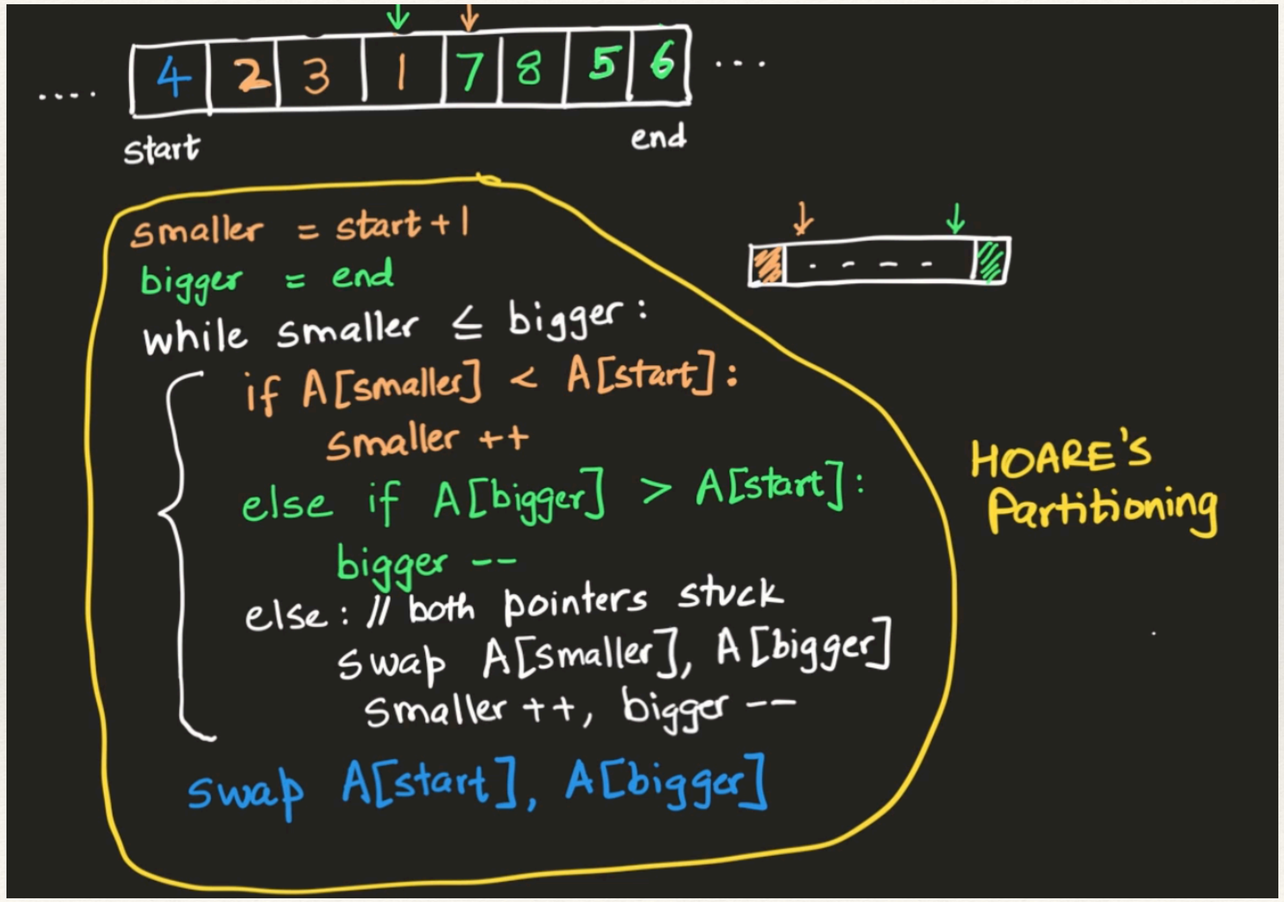


Quick sort (randomized)

- use randomized in interview rather than deterministic



Quick sort - Hoare's partition



Quick sort

- Running time of QuickSort on an array with all elements equal or sorted or reversed sorted, assuming that the pivot element is picked randomly, is O(n^2)
- Suppose we modified Quicksort to have three partitions instead of two. The left partition has values < pivot. The middle partition has values = pivot. The right partition has values > pivot. We then recurse on the left and right partitions. How much time will this 3-way partitioning take on an array with all elements equal?
  - answer = O(n)
- Above question means 3-way partitioning is good when array can have many duplicates

Merge Sort Vs Quick sort

- Merge Sort is O(n log n) for best, average and worst case
- Quick Sort is O(n log n) for best & average case, but O(n2) worst-case
- But Quick Sort is also in-place, which Merge Sort is not. (although call stack space in Quicksort should also be noted)
- Quick Sort runs faster in empirical analysis.
- Merge Sort does have one advantage: stability

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Applications

A typical application. First, sort by name; then sort by section.

```
Selection.sort(a, Student.BY_NAME);
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Selection.sort(a, Student.BY_SECTION);
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

no longer sorted by time

still sorted by time

Stability in sorting

- Check comparision scheme of each algorithm to find out it is stable or not
- Incorrect use of relational operator could cause even stable algorithm implemented in a way to make in instable. for e.g. using >= instead of >
- watch again ik video stability in sorting if needed
- Selection sort - not stable
- Insertion sort - stable
- Bubble sort - stable
- Merge sort - stable
- Quick sort - not stable
- Java uses Quicksort for primitive types, and Merge Sort (Timsort) for Objects.
- Since Merge Sort and Insertion Sort are both stable, Timsort is stable. Timsort has been Python's standard sorting algorithm since version 2.3.
- C++: sort uses quicksort, and stable_sort uses mergesort.

Stability in sorting

Numbers

45, 12(1), 8, 12(2), 5, 6

Selection sort

step1: 5<->45

5, 12(1), 8, 12(2), 45, 6

stpe2: 12(1) <-> 6

5, 6, 8, 12(2), 45, 12(1)

Insertion: -stable

Bubble: compares contiguous and promotes, so relative sequencing stays

Stability in sorting

Numbers

45, 12(1), 8, 12(2), 5, 6

Selection sort

step 1: 5<->45

5, 12(1), 8, 12(2), 45, 6

step 2: 12(1) <-> 6

5, 6, 8, 12(2), 45, 12(1)

Insertion: -stable

Bubble: Stable, compares contiguous and promotes, so relative sequencing stays

Quick: e.g. partitioning moves large than pivot to right and one large already passed by then sequence can be changed

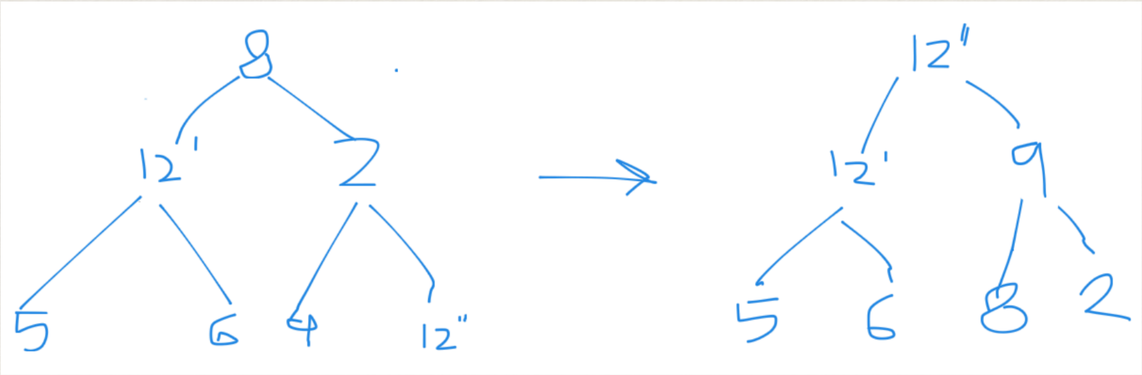
Merge: Stable - in merging phase pick from lower index sub-array when equal items found

Stability in sorting

Heap:

8 12(1) 2 5 6 9 12(2)

Max heap using build heap - 12(2) gets removed first as max



Notes

- Binary Heap is data structure used to represent Abstract Data Type priority queue
- geometric series of powers of 2 derivation to reach at S = (2^(h+1))-1
- so a complete binary tree with height h will have (2^(h+1))-1 nodes in total
-

$$\boxed{1 + 2^1 + 2^2 + \dots + 2^h} = n$$

$$S = 1 + 2^1 + 2^2 + \dots 2^{h-1} \; 2^h$$

$$2S = 2 + 2^2 + 2^3 + 2^h \; 2^{h+1}$$

common ratio

$$2S - S = 2^{h+1} - 1 \Rightarrow S = 2^{h+1} - 1 = n$$

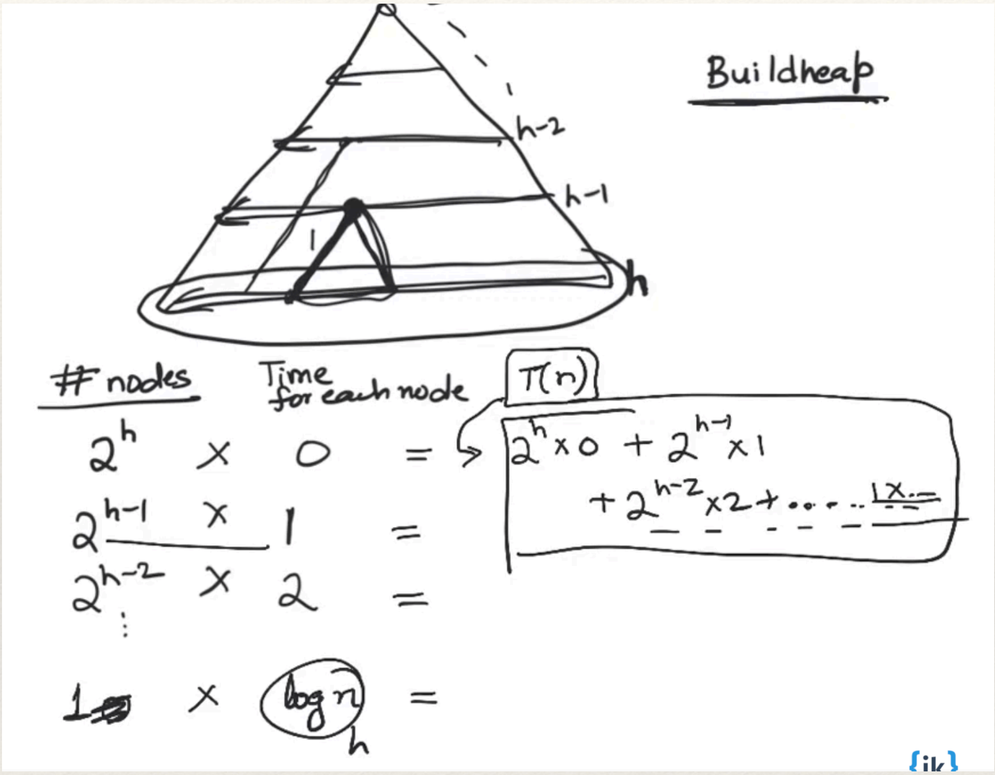
$$\Rightarrow 2^{h+1} = n+1$$

$$h+1 = \log(n+1)$$

$$h = \log(n+1) - 1$$

$$\sim \log n$$

BuildHeap complexity analysis



$$T(n) \le 2^h \cdot 0 + 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + \dots 2^{h-i} \cdot i + \dots + \dots \infty$$

$$\sum_{i=0}^{\infty} 2^{h-i} \cdot i = \sum_{i=0}^{\infty} \frac{2^h}{2^i} \cdot i$$

$$= 2^h \boxed{\sum_{i=0}^{\infty} \frac{i}{2^i}}$$

$+1$

$\times 2$

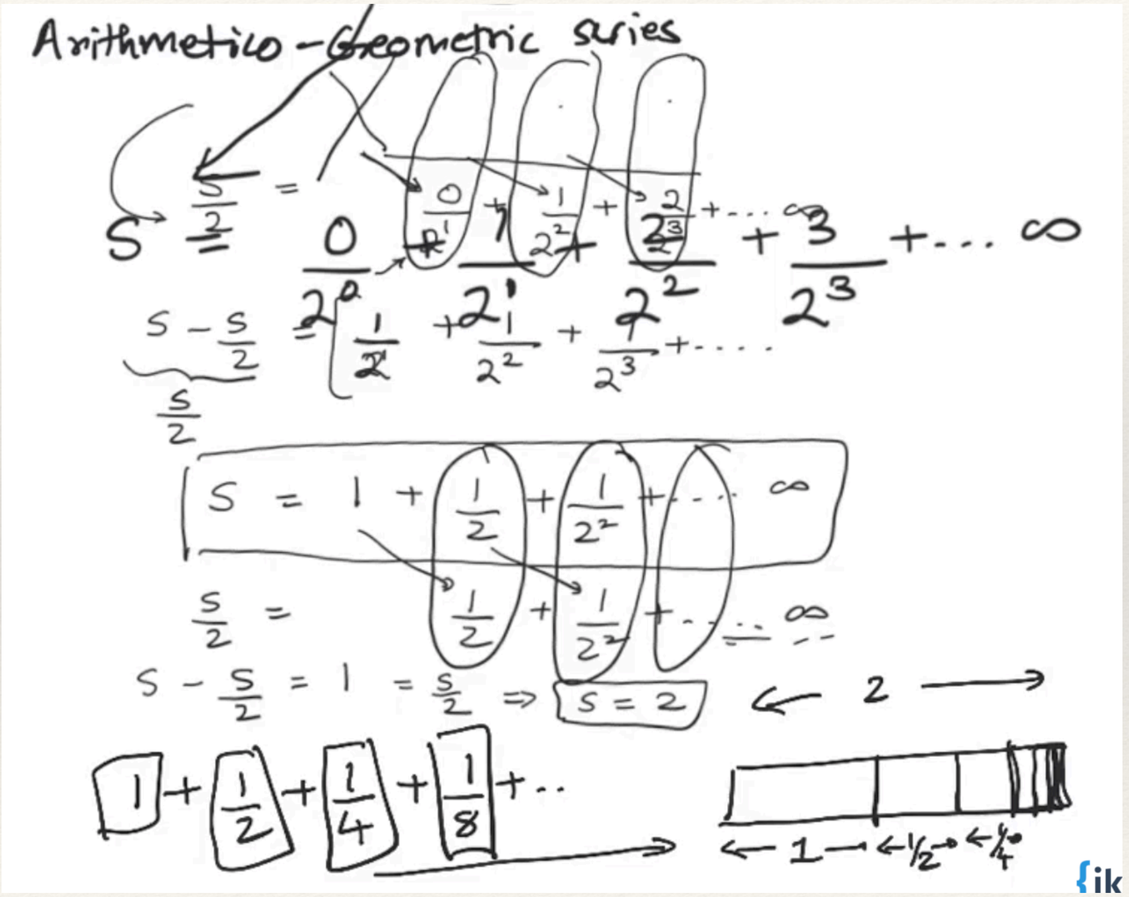
Arithmetico-Geometric series

$$S = \cancel{\frac{0}{2^0}} + \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \dots \infty$$

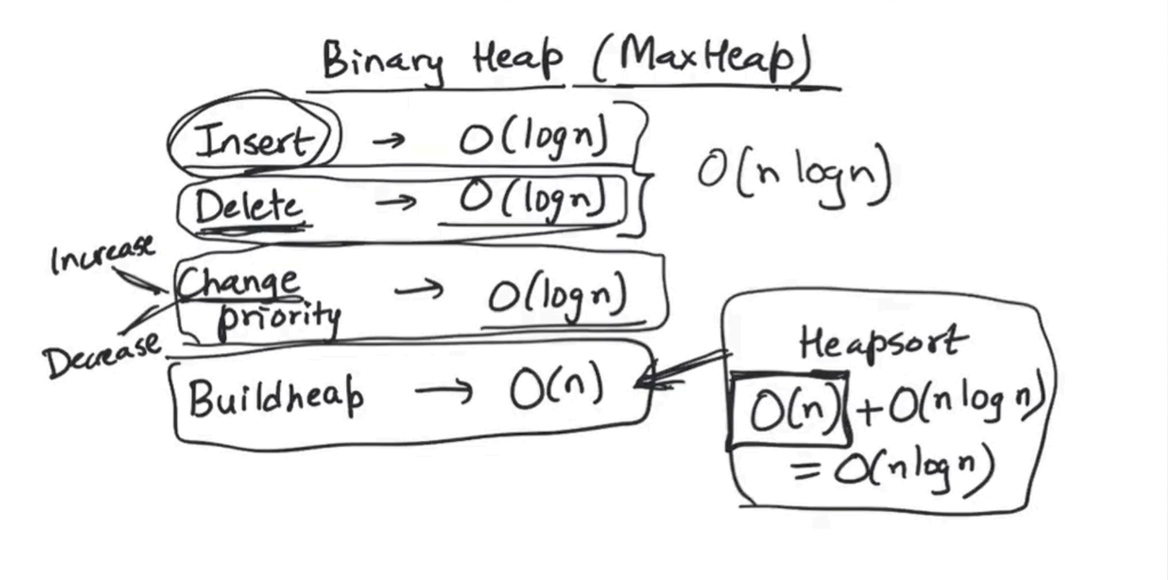
$$\frac{S}{2} = \frac{0}{2^1} + \frac{1}{2^2} + \frac{2}{2^3} + \dots \infty$$

$$\underbrace{S - \frac{S}{2}}_{\frac{S}{2}} = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \dots$$

BuildHeap complexity analysis

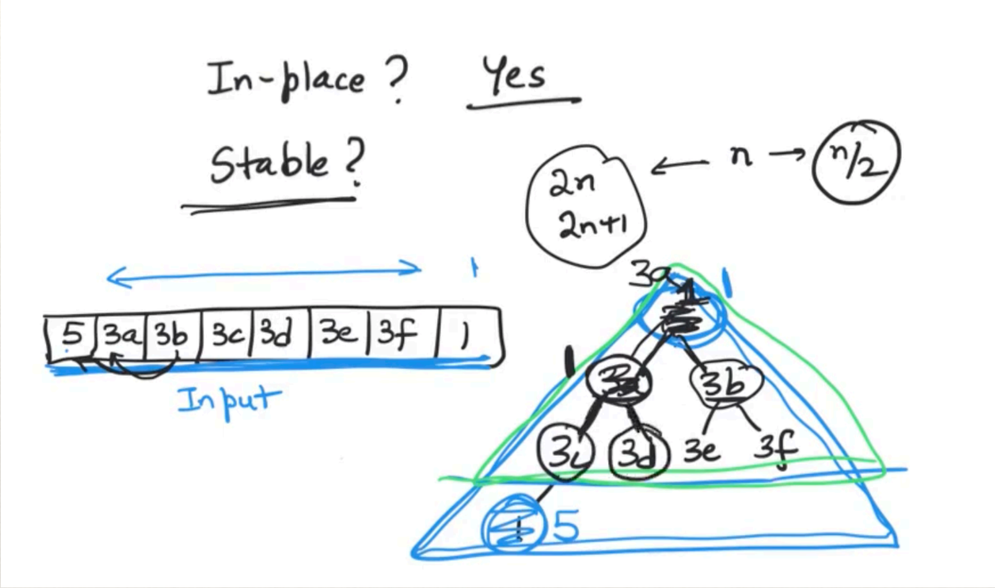


Applications



- every operation of heap can be done in (log n) time
- build heap can be done in (n) time
- and heap sort in (n log n), regardless of inserts were done repetitively (O(n log n) +O(n log n)) or using build heap (O(n) +O(n log n))

Applications



Notes

- every operation of heap can be done in (log n) time
- watch average case analysis for quick sort and make notes again
-

Bucket Sort

Divides the entire range of values into a set of buckets. Generalization of Counting Sort (where bucket size = 1)

Steps -

- Set up an array of initially empty "buckets".
- Scatter: Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Gather: Visit the buckets in order and put all elements back into the original array.

Time and Space Complexity

Sorting Algorithm	Average Case	Best Case	Worst Case
Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick Sort	O(n.log(n))	O(n.log(n))	$O(n^2)$
Merge Sort	O(n.log(n))	O(n.log(n))	O(n.log(n))
Heap Sort	O(n.log(n))	O(n.log(n))	O(n.log(n))
Counting Sort	O(n+k)	O(n+k)	O(n+k)
Radix Sort	O(n*k)	O(n*k)	O(n*k)
Bucket Sort	O(n+k)	O(n+k)	$O(n^2)$

Sorting Algorithm	Space Complexity
Bubble Sort	O(1)
Insertion Sort	O(1)
Selection Sort	O(1)
Quick Sort	O(log(n))
Merge Sort	O(n)
Heap Sort	O(1)
Counting Sort	O(k)
Radix Sort	O(n + k)
Bucket Sort	O(n)

Sorting Algorithm	Stable Sort?
Bubble Sort	Yes
Insertion Sort	Yes
Selection Sort	No
Quick Sort	No
Merge Sort	Yes
Heap Sort	No
Counting Sort	Yes
Radix Sort	Yes
Bucket Sort	Yes

Problem solving skill

- Problem solving is about finding a solution and if you already know solution that becomes an exercise
- Timed tests are important
- Coding fluency is important
- discuss approaches and tentative solution with interviewer, agree on one and then only start coding
- try to take minimum hints
- keep investigating even if a solution reaches, dead end, that will give you some clue to resolve that dead-end
- active engagement is important
- Explore, get Lost, Find a Path

Sorting algos misc

- Counting sort useful when range of numbers in known and is limited
- Radix sort is multiple passes of counting sort, last round for sign (need to explore)
- Mathematical concepts needed - AP, GP
- High school algebra courses can be done from net
- Few useful concepts not covered
  - Enhance and conquer design strategy
  - Cycle sort
  - Tree sort (make a BST and inorder traversal)
  - Prefix sum
- Recursion can **head recursion** (make recursive call work when returned) or **tail recursion** (work upfront and then make recursive calls)

Interview problems related to Sorting

- Presorting
- Extensions of Merge Sort
- Extensions of QuickSort
- Extensions of Heapsort

Pre-sorting Problems

- All pre-sorting problems are transform and conquer problems which use Input re-arrangement. Transform and conquer problems either use Input re-arranegment (sorting) or representation change (tree)
- pre-sorting problems use following after sorting
  - binary search
  - or linear scan
  - or 2 pointer pass
  - or index lookup

Pre-sorting Problems

1. Two Sum (Presort and 2-pointer) or Hashset
2. 3 sum
3. Searching (pre-sort + Binary search)
4. Closest pair ($|x-y|$ is minimum) presort + linear scan
5. Element uniqueness (finding duplicates) - (pre-sort+linear scan) or hashtable
6. Frequency distribution - (pre-sort+linear scan) or hashtable
7. Kth largest / Kth smallest / Top K / Median/ Percentile - (pre-sort + index lookup)

Extension of merge sort

1. Merge 2 sorted arrays (1 variant is merge in-place (1 array has extra space needed)
2. Union of 2 sorted arrays
3. Intersection of 2 sorted arrays
4. Intersection of un-sorted arrays
    1. Sort and then find intersection
    2. add one of the 2 arrays to a hash table and scan another array
5. Prefix sums, sliding window and 2 ptr (all linear search O(n)) were mentioned need to explore

Extension of Quick sort (quickselect)

1. Kth largest/smallest in an array
2. K smallest/largest of an array
3. Median of array
4. Sort Colors/ Dutch national flag (3-partition quickselect)

Extension of Heap sort (quickselect)

1. Kth largest/smallest in an stream
2. K smallest/largest in an stream
3. Median/percentile of stream
4. Sort Colors/ Dutch national flag (3-partition quickselect)