Problem solving 2

- There are many problems that use sorting as like a pre-step.
- This is where you would need to pick a sorting algorithm based on various trade offs.

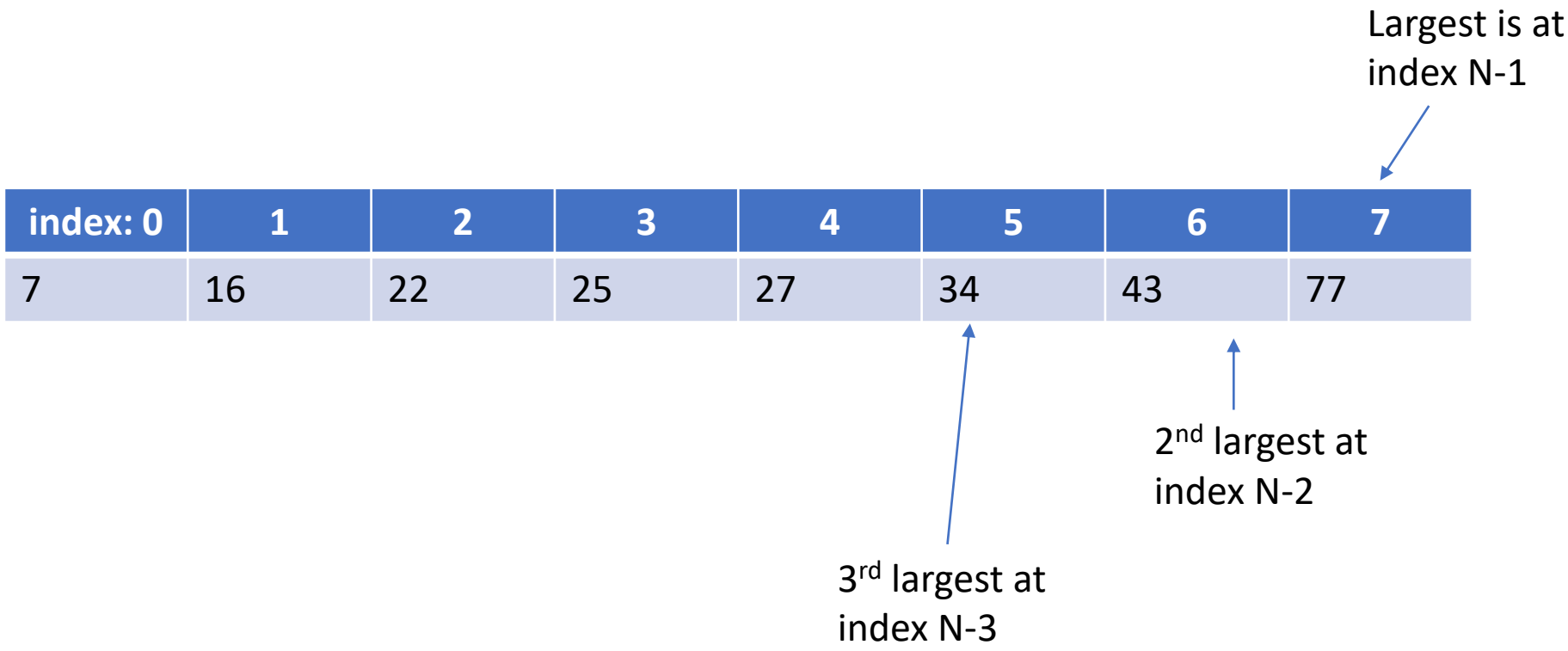
- Problems we will look at:

    - Finding Kth largest (smallest) element in a collection
    - Finding Kth largest (smallest) element in streaming data

    - Finding median or Kth percentile.

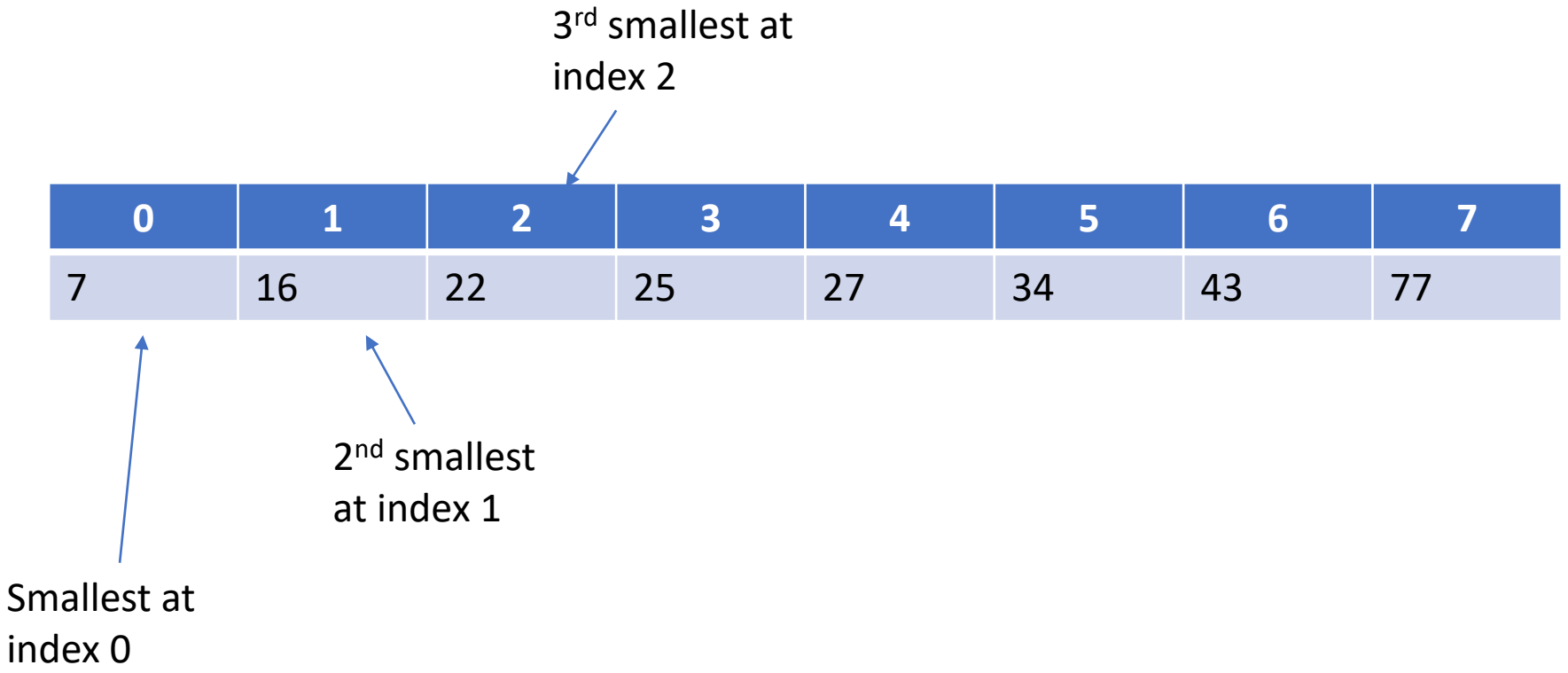
K^{th} largest

- Lets say we need to find the Kth largest element in a given array.
- This means that if we were to sort the array, then we could pick the element at index N – K



- So, K^{th} largest at index N-K

K^{th} largest

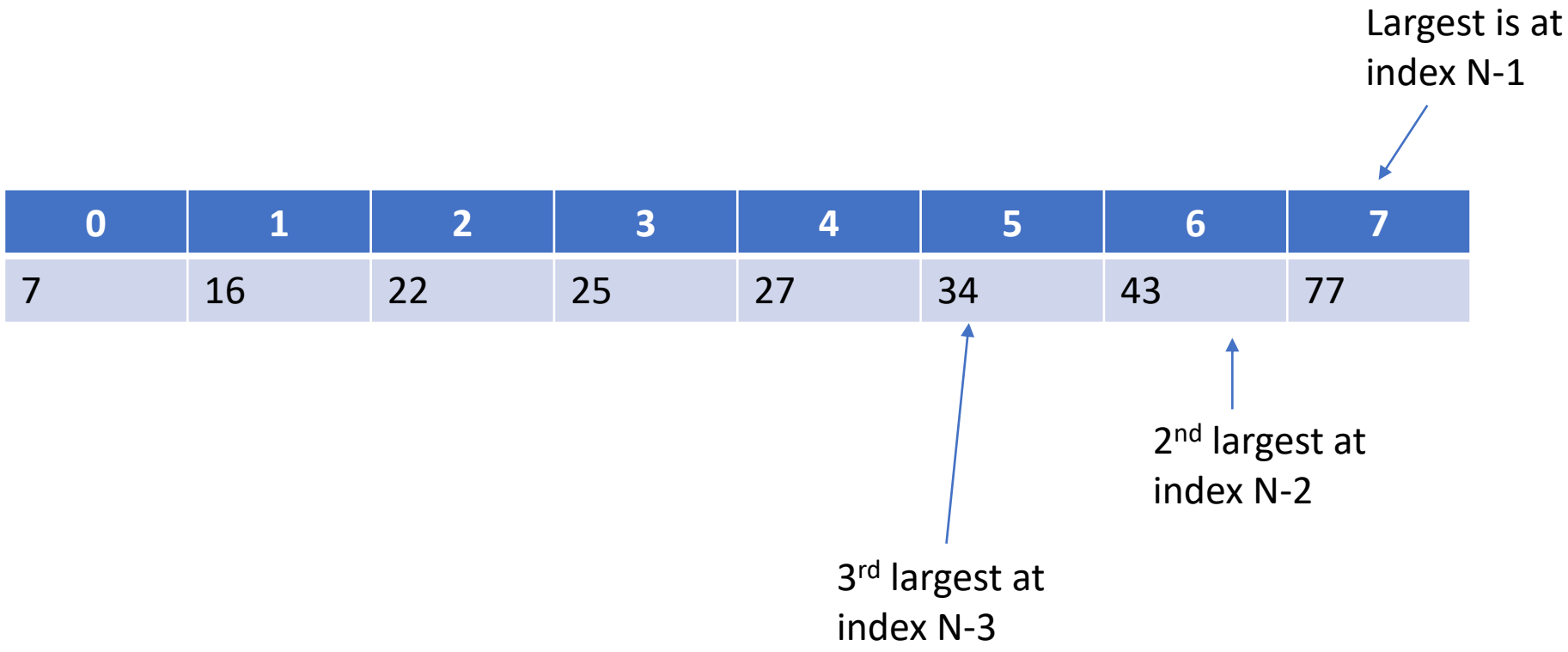


- K^{th} smallest would be at index K – 1 (if the array were sorted)

K^{th} largest

- Now, what are some options to find the K^{th} largest element:

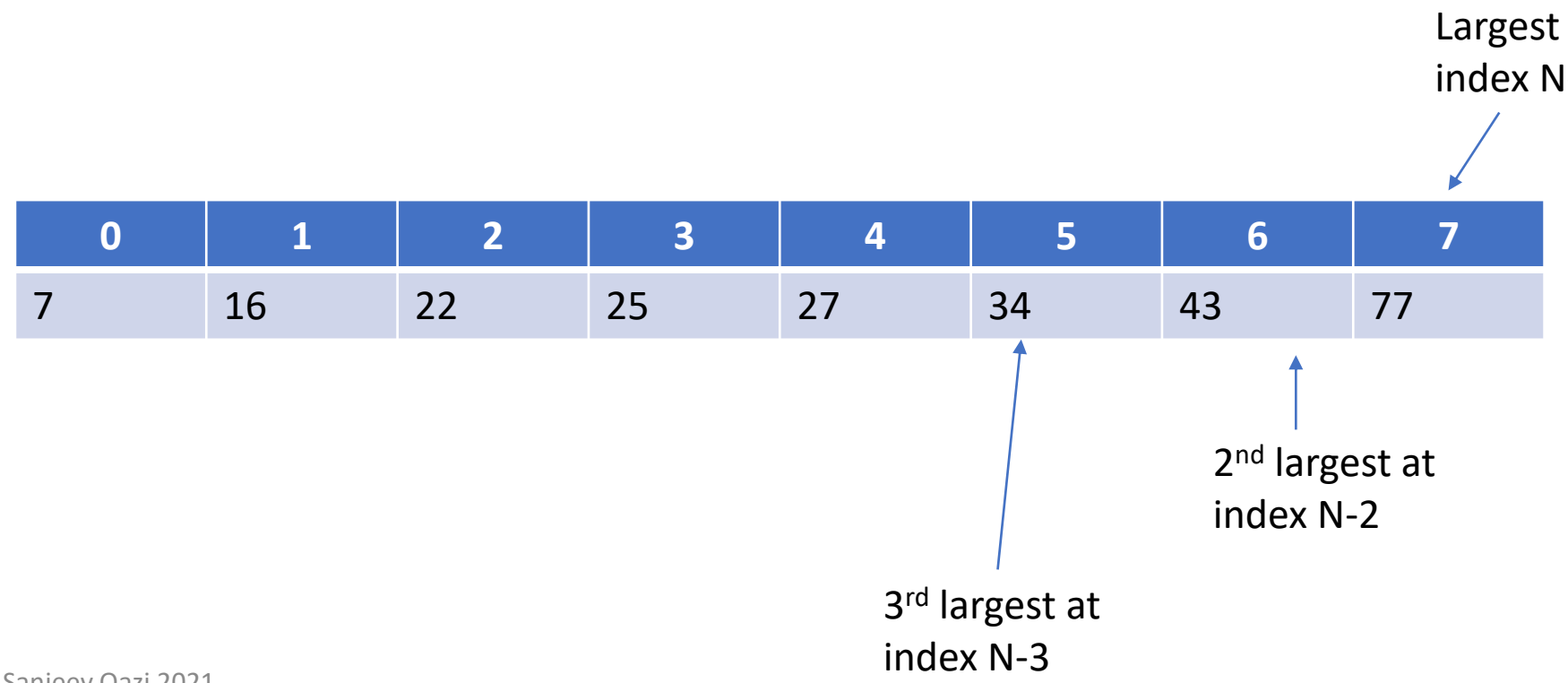
- Brute force: What would be a brute force method?



K^{th} largest

- We could sort the array and then return the element at index N-K.

- Now, to be a little smarter, we could say that maybe we don't need to sort the whole array…
  - If so, which sort algorithms are such that when an element is placed at index N – K , we know that's the element's final place in the sorted array?



K^{th} largest

- Which algorithms are such that when an element is placed at index N – K , we know that's the element's final place in the sorted array?

  - Selection sort
  - Insertion sort
  - Merge sort
  - Quicksort
  - Heapsort
  - Bubble sort

K^{th} largest

- Which algorithms are such that when an element is placed at index N – K , we know that's the element's final place in the sorted array?

  - Selection sort
  - ~~Insertion sort~~
  - ~~Merge sort~~
  - Quicksort
  - ~~Heap sort~~
  - Bubble sort

K^{th} largest

- Brute force

- If we sort and return element at index N-K

- Using Selection sort
  - We could stop when index N-K is filled up.
  - Time: O ((N – K) * N), which reduces to O (N^2)

  - The fact that we can stop early doesn't change the Big O

K^{th} largest

- Could we solve this by using a min or max heap?
  - What would we need to do?

K^{th} largest

- We could use a min heap
  - Add all elements to it,
  - But limit its size to K.

- When all elements are processed, min heap root would be K^{th} largest element within that heap because the root is smaller than every other element in the heap.

```
For each element e
        if  minheap.size < K
                minheap.Insert ( e )
         else
                if e > minheap.GetMin()
                        minheap.Insert ( e )
                        minheap.ExtractMin()          // Since we want to limit size to K
                else
                        Do nothing… just ignore e
```

K^{th} largest

- We could use a min heap
  - Add all elements to it,
  - But limit its size to K.

- When all elements are processed, min heap root would be K^{th} largest element within that heap because the root is smaller than every other element in the heap.

```
For each element e
      if  minheap.size < K
            minheap.Insert ( e )              ← O ( log K )
        else
            if e > minheap.GetMin()                    ← O ( 1 )
                  minheap.Insert ( e )        ← O (log K )
                  minheap.ExtractMin()        ← O (log K )      // Since we want to limit size to K
```

processing time for **each** element:

O (1) + O (log K) + O (log K) = O (log K)

For **N** elements O (N log K)

N log K would be N log N in general, since K could be N/2

Time: **O (N log N)**

Space: O (K)

K^{th} largest

- If we could do better than O (N log N), what would that complexity be ?

K^{th} largest

- If we could do better than O (N log N), what would that complexity be ?

- It could be O(N) or O(log N) or O (1)
  - Now, O (1) would be impossible.
  - O (log N) would also be impossible since we need to look at each element at least once, to determine the K^{th} largest.

  - So we are left with O (N) possibility.

  - Well, turns out that we can use a modification of quicksort to determine K^{th} largest in linear time !!!

K^{th} largest

- Remember in quick sort, the Partition method places the pivot element at its final sorted place (index).

```
QuickSort ( A, start, end )
        if ( start == end )
                return;

        pivotIndex = PickAPivotIndex ( start, end )

        pivotFinalIndex =  Partition ( A, start, end, pivotIndex )    // Now the element at pivotFinalIndex is at its final sorted position.

        QuickSort( A, start, pivotFinalIndex – 1 );                   // sort sub array to left of pivotFinalIndex

        QuickSort( A, pivotFinalIndex + 1,  end );                    // sort sub array to right of pivotFinalIndex
```

K^{th} largest

- So, roughly speaking, for getting the K^{th} largest, what we will do is:
  - After each call to Partition, check if pivot final index == N – K ← *Did we get lucky and pivot final index was N-K*
    - If it is equal, we are done, as we found the K^{th} largest.
    - If not, we continue to call Partition, but, only on the sub array where index N-K is.

```
pivotIndex = PickAPivotIndex ( start, end )

pivotFinalIndex =  Partition ( A, start, end, pivotIndex )

if pivotFinalIndex == N – K                          // we r done, we found Kth largest
        return A [ pivotFinalIndex ]

else if N – K < pivotFinalIndex                      // index of interest is to left of pivot index
        Look at the array to left of pivotFinalIndex
else                                                 // index of interest is to right of pivot index
        Look at the array to right of pivotFinalIndex
```

K^{th} largest

```
GetKthLargest( A, K )
        N =  A.Length;              indexOfKthLargest = N – K
        return QuickSelect ( A, 0, N - 1, indexOfKthLargest )

QuickSelect ( A, start, end, indexOfKthLargest )
        if ( end – start ) <= 0
                return A [ start ]

        pivotIndex = PickAPivotIndex ( start, end )          // same as what we did in quicksort
        pivotFinalIndex =  Partition ( A, start, end, pivotIndex )   // same as what we did in quicksort

        if pivotFinalIndex == N – K                          // we r done, we found Kth largest
                return A [ pivotFinalIndex ]
        else if  N – K < pivotFinalIndex                     // index of interest is to left of pivot index
                end = pivotFinalIndex - 1
        else                                                 // index of interest is to right of pivot index
                start = pivotFinalIndex + 1

        return  QuickSelect ( A, start, end, indexOfKthLargest )
```

K^{th} largest

Now, is this really O (N) ?

Remember the recurrence relation from Quick sort… lets look at that with a slight modification

In the best case, where we divide the array into half each time, we will have:

$$T (N) = T (N / 2) + (N - 1)$$

$$T (N) = T (N / 4) + (N/2 -1) + (N - 1)$$

$$T (N) = T (N / 8) + (N/4 -1) + (N/2 -1) + (N - 1)$$

...

$$T (N) = T (1) + (N / 2^h – 1) + ... + (N/4 -1) + (N/2 -1) + (N - 1)$$

- This becomes O (N)

- In the previous slides, we assumed the best case where the array is divided into half each time.
- That of course wont happen.
- So, lets be very conservative and assume we happen to pick the pivot such that every time only 10% of the search space is eliminated.

- We will use this mathematical formula for the sum of am infinite geometric series

  $a + ar + ar^2 + ar^3 + \ldots = a / (1 - r)$ r is < 1

- Our equation will look something like:

$K*N + 0.9K*N + 0.9^2 K*N + 0.9^3 K*N + 0.9^4 K*N + \ldots$

$= K N (1 + 0.9 + 0.9^2 + 0.9^3 + 0.9^4 + \ldots)$

$= K N (1 / (1 - 0.9))$

$= K N (1 / 0.1) = K N (10)$

$= \mathbf{O (N)}$ // drop the constants K and 10

- Even if we assume that every time only 3% of the search space is eliminated:
  - Our equation will look something like:

- We will use this mathematical formula for the sum of am infinite geometric series

  $a + ar + ar^2 + ar^3 + \ldots = a / (1 - r)$ r is < 1

$K*N + 0.97K*N + 0.97^2 K*N + 0.97^3 K*N + 0.97^4 K*N + \ldots$

$= K N (1 + 0.97 + 0.97^2 + 0.97^3 + 0.97^4 + \ldots)$

$= K N (1 / (1 - 0.97))$

$= K N (1 / 0.03) = K N (33)$

$= \mathbf{O (N)}$ // drop the constants K and 33

K^{th} largest

Now, lets look at the worst case complexity

In this scenario, partition would happen such that we happen to pick the *biggest* or *smallest* element as the pivot, which means after partitioning, all the elements will be on one side of the pivot (and nothing on the other side).

This means we are reducing the search space by only **one element** each time.

The recurrence for that will look something like:

$KN + K(N-1) + K(N-2) + \ldots + 1$
$= K (N + (N-1) + (N-2) + \ldots + 1)$ ← This is the sum of first N numbers which is $N (N + 1) /2$
$= K N (N+1) / 2$

$= O (N^2)$

However, for the worst case to happen, we would need to consistently pick the smallest or largest element as pivot.

But there are ways we discussed that we could prevent / mitigate that.

LAB - K^{th} largest

Input: 1, 5, 9, 2, 25, 16, 19, 22, 4, 7, 55

K: 4

Write a function ***FindKthLargest*** that takes the input array and K as parameters.

It should return the Kth largest element in the input.

For the lab, you can assume K passed in is valid, which means 1 <= K <= N

Median

Median of a given set of numbers is the middle element when looking at that data sorted (ascending or descending).

- Odd number of elements ➔ one middle element.
- Even number of elements ➔ two middle elements. (take mean of two middle)

- Median is usually a better representative of the set of numbers than their average.
- Median is the 50^{th} percentile.

- For example:
  - Companies that run a 24/7 software service measure their uptime and latencies as percentiles.
  - They may look at P95 or P99 or P99.9 or P99.99 , etc.

  - A service having an uptime of 5 9s means uptime of 99.999% ➔ max downtime of 0.001%
  - For a month, 0.001% is about 86400 X 30 X 0.001% = 26 seconds downtime.
  - For a year, it is about 26 X 12 = 311 seconds or about 5 mins downtime.

Median

if numbers are

10 30 80 20 50 40 70 60 90

sorted: 10 20 30 40 50 60 70 80 90

average and median are 50

if numbers are

20 10 20 20 500 30 25 60 40 700

sorted: 10 20 20 20 25 30 40 60 500 700

average: 142.5

median: (25 + 30) /2 = 27.5 ← lot more representative of the numbers than the average.

Imagine this for a much bigger input

Median

Now, if we have to find the median of some numbers:

The numbers could either be all given up front (think of that as static data)

OR

The numbers could be coming in one (or a few) at a time (think of this as streaming data).

Lets look at both cases.

Median of numbers in an array

Static data

Lets assume we are given an array containing the numbers.

Question for u all:

What are some of our options to find the median?

Median of numbers in an array

Finding a median of numbers in an array is essentially finding the K^{th} largest number, where K = N/2

1. We can sort the data and then the middle element or mean of middle two elements is the median.

   What would this time complexity be?

2. Use a heap
   - We discussed details of this option for the K^{th} largest problem.

3. Quick select
   - We discussed details of this option for the K^{th} largest problem.

Median of streaming data

Now, lets look at finding the median for **streaming** data.

For this discussion, lets assume only one new number comes in at a time.

Options are:

1. Maintain a sorted array:
    1. Every time a new number comes in, we put it in the right place such that the array is still sorted.

    2. What sorting algorithm should we use in this case?

    3. What's the time complexity per incoming number?

What other options?

Median of streaming data

Now, lets look at finding the median for **streaming** data.

For this discussion, lets assume only one new number comes in at a time.

Options are:

1. Maintain a sorted array:
   - Every time a new number comes in, we put it in the right place such that the array is still sorted.
   - What sorting algorithm should we use in this case?
   - What's the time complexity per incoming number?
2. Quick select
   - What's the time complexity per incoming number?

Median of streaming data

Other options:

We could use a heap.

1. Every time a number comes in, we would add that number to the heap.

2. Now, to compute the median, we would:
    1. Remove half the numbers
    2. Get the root element, which will be the median.
    3. But now we would need to add the removed elements from step#1 back into the heap, since we need them for the next number that comes in (this is streaming data).

    4. Next slide: Lets look at the complexity of this approach.

Median of streaming data

Heap option:

1. Every time a number comes in, we add it to the heap. ← O (log N)
2. Now, to compute the median, we would:
    1. Remove half the numbers. ← (N / 2) * log N = O (N log N)
    2. Get the root element, which will be the median. ← O (1)
    3. But we need to add removed elements back into heap, we need them to compute new median when the next number comes in (this is streaming data). ← (N / 2) * log N = O (N log N)

So this means processing <u>one number</u> is:

O (log N) + O (N log N) + O (1) + O (N log N)

= O (N log N)

This turns out to be worse than keeping a sorted array and inserting the new number each time (the option we looked at few slides ago).

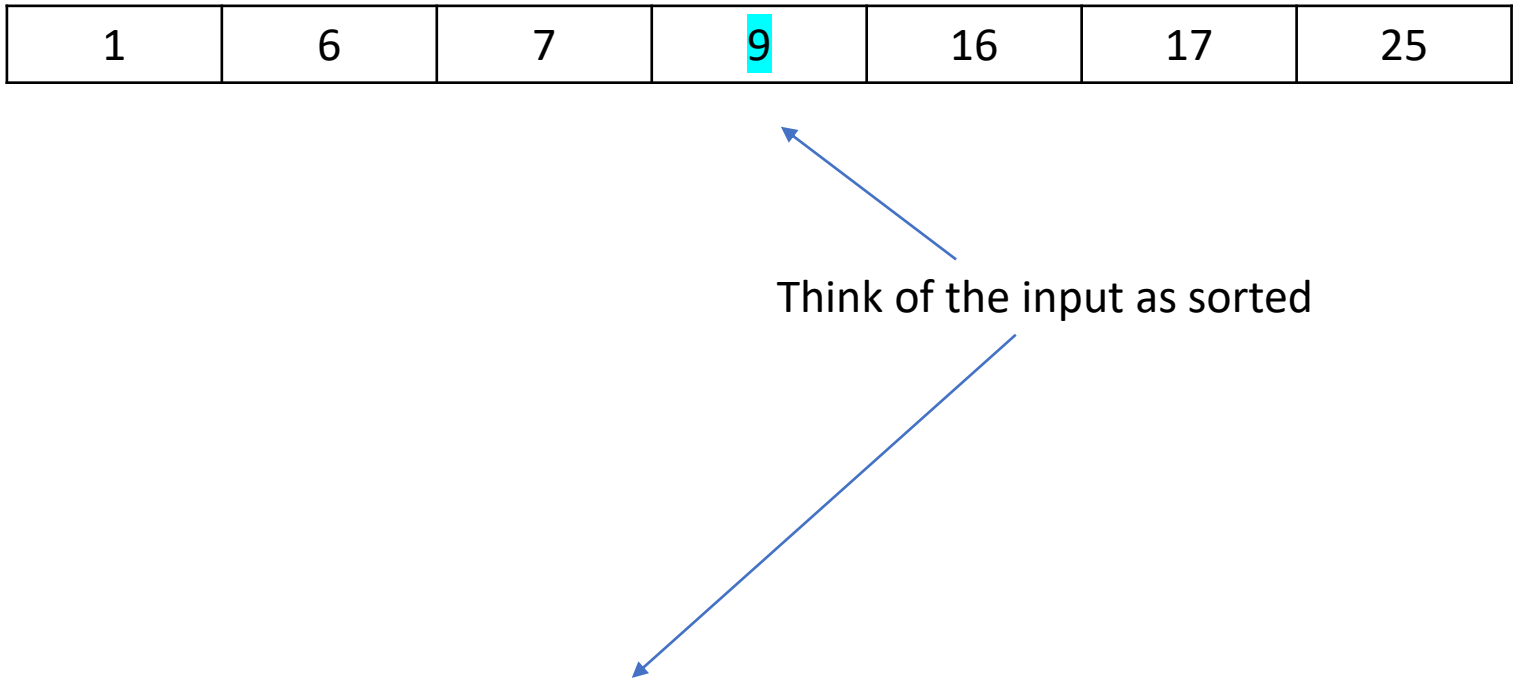
Median of streaming data

Before we talk about our next approach, lets look at the following:

N is **odd**

Input: 9, 7, 1, 6, 25, 17, 16

Median: 9



N is **even**

Input: 9, 7, 1, 6, 25, 17, 16, 27

Median: (9 + 16) / 2.0 = 12.5

1	6	7	9	16	17	25	27

What is the property of 9, compared to elements to the left of it?

And what about 16, compared to elements to the right of it?

Median of streaming data

Now, think of the array as split in the center to get these two sub arrays: left half and right half

And now, if I wanted to know that 9 is the largest element among the numbers in the left half, without sorting all numbers, what data structure would I use?

What about 16 from the other sub array?

Median of streaming data

So what we are really saying is that we would like to add the incoming numbers (one at a time) to either of the two heaps.

And then we can ask the heaps to give us the root (O (1)) and that would help us to calculate the median.

Questions to be answered:

1. How do we pick which heap to add the incoming number?
    1. Next slide

2. What about the sizes of the two heaps? How do we keep them balanced?
    1. We cannot afford to get the two heaps out of balance:
        1. by balance we mean that the difference in their sizes is either 0 or 1 … i.e.: at most 1
    2. abs (lowerNumbersHeap.size() - upperNumbersHeap.size()) <= 1

Median of streaming data

lowerNumbersHeap (maxheap)

upperNumbersHeap (minheap)

When a new number comes in, we will add to one of the heaps.

If its less than the root of the max heap, add to max heap, else to min heap.

Median of streaming data

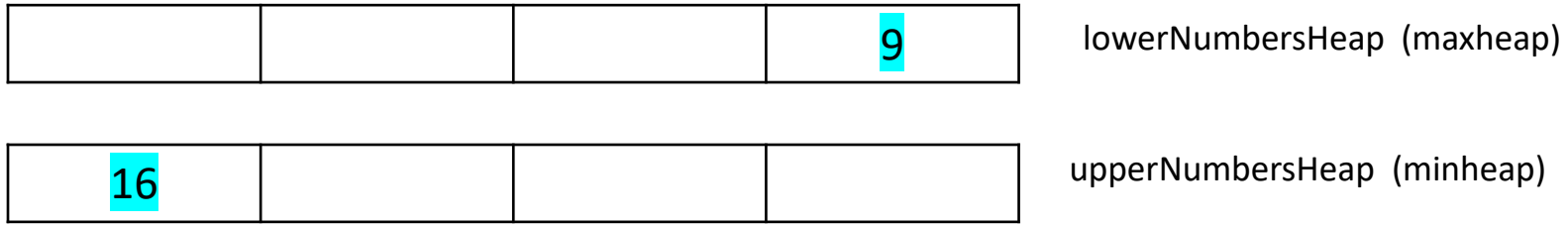
lowerNumbersHeap (maxheap)

upperNumbersHeap (minheap)

And then we will worry about the sizes of the heaps… we don’t want them to be different in size by more than 1

If the difference in size is more than 1, we need to remove from the bigger heap and insert in the smaller heap.

Median of streaming data



Finally, we can compute the median:

```
If  ( upperNumbersHeap.Size( ) == lowerNumbersHeap.Size() )
        median = (upperNumbersHeap.GetRoot() + lowerNumbersHeap.GetRoot() ) / 2.0
else ( lowerNumbersHeap.Size( ) > upperNumbersHeap.Size() )
        median = lowerNumbersHeap.GetRoot()
else
        median = upperNumbersHeap.GetRoot()

return median
```

Median of streaming data

Time complexity:

Add number to one of the heaps: O (log N) where N is the number of elements in the heap.

Rebalance the heaps: O (log N)

Compute median: O (1)

Overall complexity: O (N log N)

Space complexity: O (N) since we store all numbers.