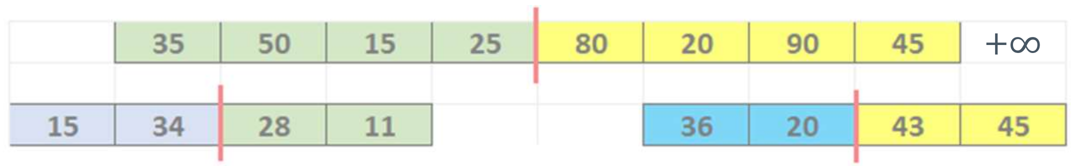
Lecture 13

Quick Sort: Overview, and Worst, Best & Average Case analysis

Randomized Quicksort; Heap Sort: Overview, and Worst, Best & Average Case Analysis

Best case Complexity: Quick Sort

- $T(n)=T(n/2) + T(n/2) +cn$
- $=2T(n/2) + cn$
- $=2[2T(n/2^2)\ cn/2] +cn$
- $=2^2\ T(n/2^2) + 2cn$
- $=2^3\ T(n/2^3) + cn + 2cn$
- $=2^3\ T(n/2^3) + 3cn$
- .
- .
- .
- $=2^k\ T(n/2^k) + kcn$
- $=n.T(1) + \log_2 n\ *cn$
- $=n.1 + cn*\ \log_2 n=n\ \log_2 n=\Omega(n \log n)$

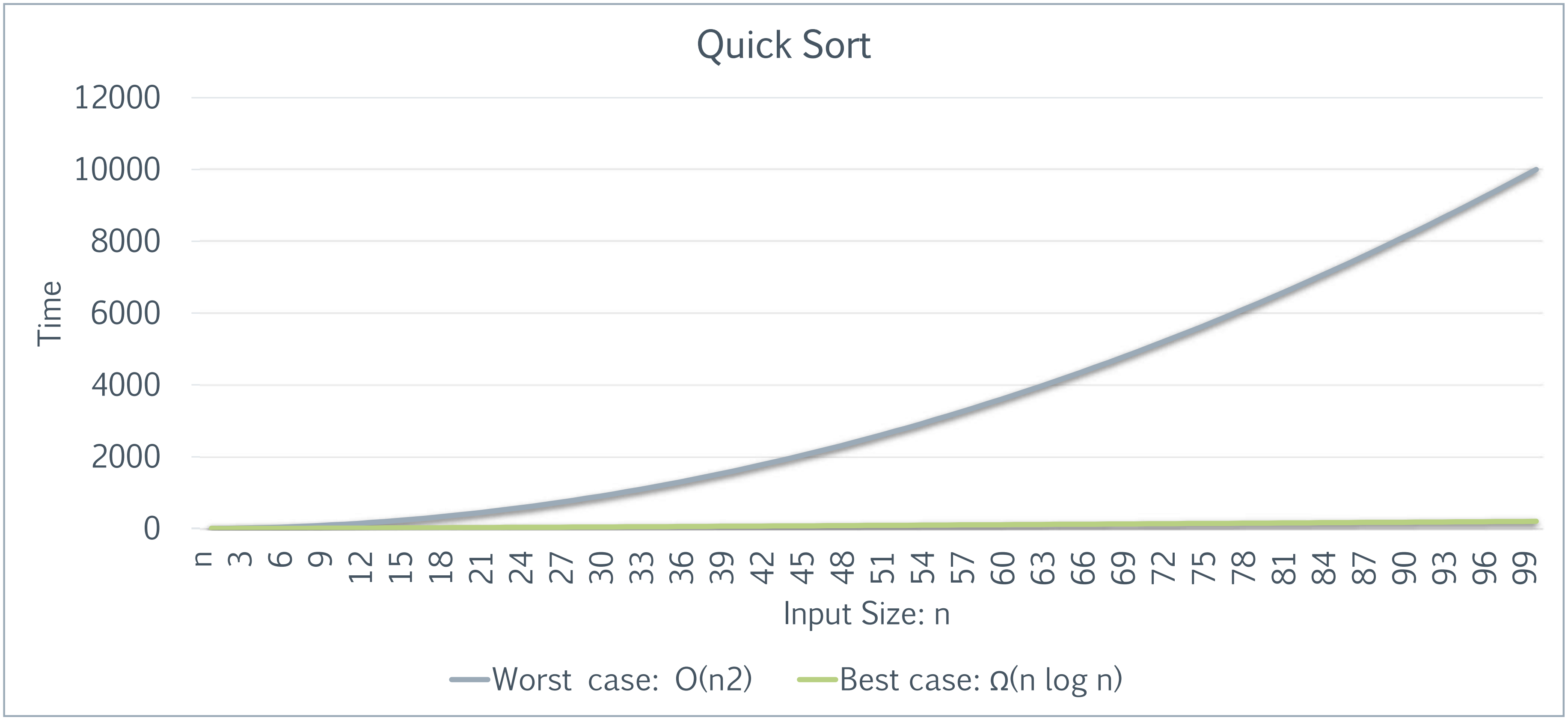


Worst-Case Complexity: Quick Sort

15	20	25	35	45	50	80	90	$+\infty$	
i=1							j=8		
i++							j++		
	15		20	25	35	45	50	80	90
No Subarray on the Left of Pivot element					j will be decremented				

- $T(n)=T(n-1) + cn$
- $T(n) \quad =T(n-1) + cn$
- $=T(n-2) +c(n-1) +cn$
- $=T(n-3) + c(n-2) + c(n-1) +cn$
- .
- .
- .
- $=T(1) + c*2 +c*3 + \ldots + c(n-1) + cn$
- $=1 + c*2 + c*3 + \ldots + c(n-1) + cn$
- $=c + 2*c + 3*c + \ldots + n*c +1 - c$ adding and subtracting c
- $=c[1 + 2 + 3 + \ldots. + n] – c +1$
- $=c*n(n+1)/2 -c +1$
- $= cn^2/2 + cn/2 – c +1$
- $= O(n^2)$ – worst case complexity as $n^2 > n \log n$ (Best Case $\Omega(n \log n)$) $T(n)=T(n-1) + cn$

Graphical View: Worst Case and Best case



Average Case Complexity: Quick Sort

	Recurrence Relation:					
	T(n)=T(n-1) + T(0) +n					O(n2)
Input:	10	11	12	13	14	15
Input:	15	14	13	12	11	10
Input:	8	8	8	8	8	8

Balanced Partitioning						
Input						
8	6	7	3	4	5	pivot
3	6	7	8	4	5	pivot
3	4	7	8	6	5	pivot
3	4	2	8	6	5	pivot
3	4	2	5	6	8	
			pivot			

Worst case analysis

Worst case: **unbalanced** partitioning

Input: 10,11, 12, 13, 14, 15 →15 pivot

(10, 11, 12, 13, 14)15

14 pivot

Produces **0 and n-1** elements

10 11 12 13 14 15

n-1

0 elements

Produces **0 and n-1** elements

10 11 12 13 14 15

n-1

No element to the right

0

Recurrence Relation: T(n)= T(n-1)+ T(0)

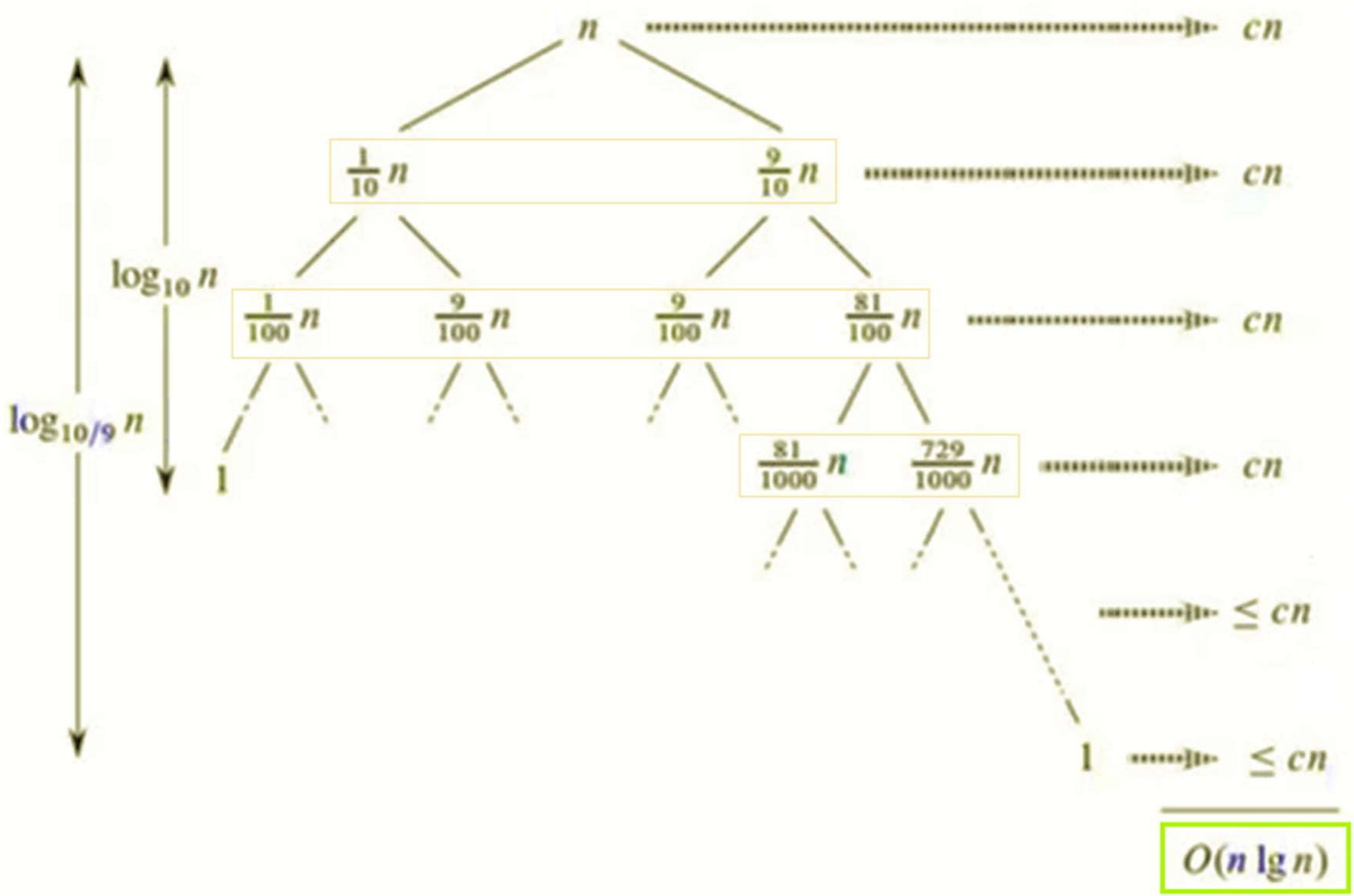
10 11 12 13 14

n-2 elements

0 elements

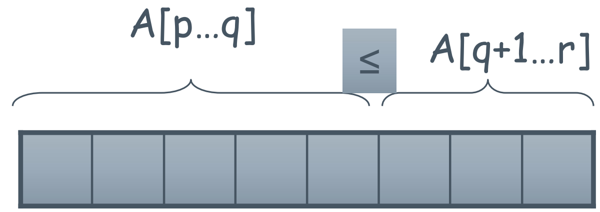
15(sorted)

Quick sort Average case analysis

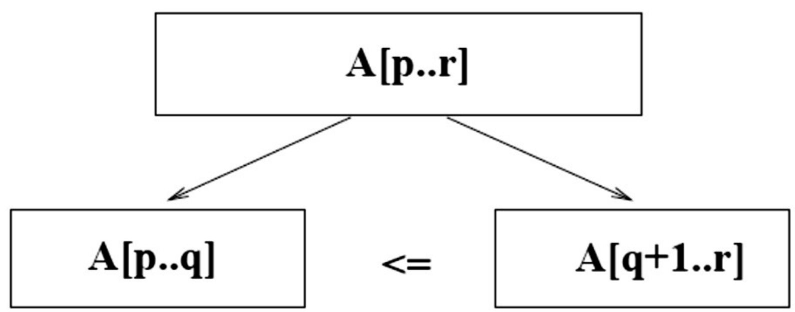


Quick sort

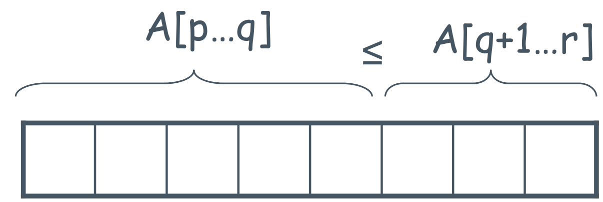
- *Sort an array A[p…r]*



- ***Divide***
  - *Partition the array A into 2 subarrays A[p..q] and A[q+1..r], such that each element of A[p..q] is smaller than or equal to each element in A[q+1..r]*
  - *Need to find index q to partition the array*



Quick sort



- **Conquer**
  - *Recursively sort A[p..q] and A[q+1..r] using Quicksort*
- **Combine**
  - *Trivial: the arrays are sorted in place*
  - *No additional work is required to combine them*
  - *The entire array is now sorted*

QUICK SORT

Alg.: QUICKSORT(A, p, r)

Initially: p=1, r=n

if *p < r* ***then*** *q* $\leftarrow$ ***PARTITION****(A, p, r)*

QUICKSORT *(A, p, q)*

QUICKSORT *(A, q+1, r)*

$Q(n)=Q(q) + Q(n-q) + f(n)$ ($f(n)$ depends on *Split()*) ***PARTITION****())*

Recurrence:

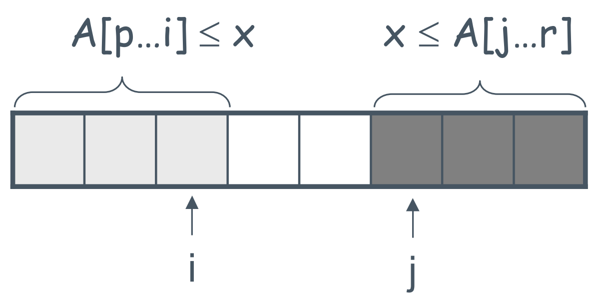
$$T(n) = T(q) + T(n - q) + f(n)$$

Partitioning the Array

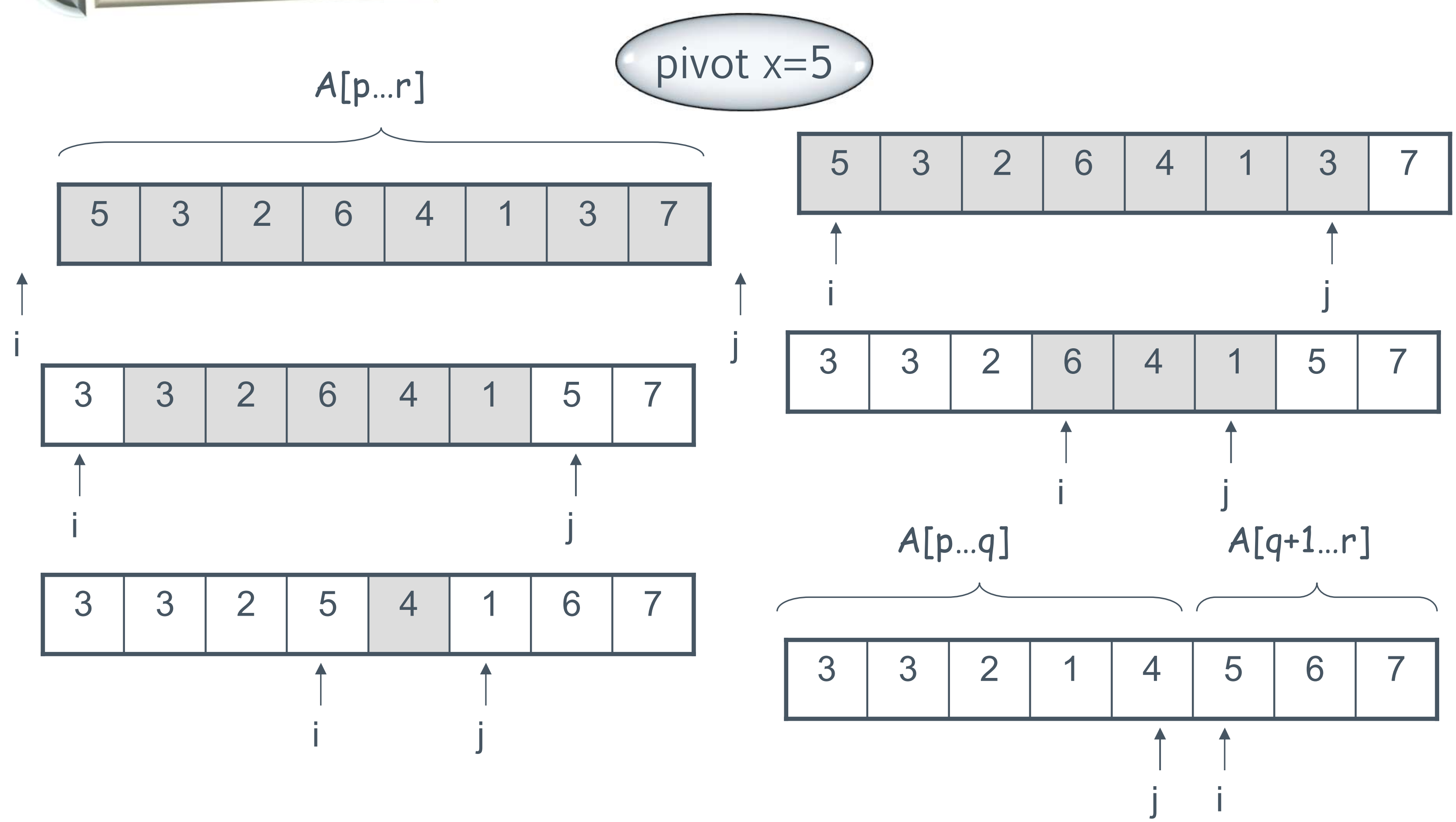
- Choosing ***PARTITION***()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition
  - Select a pivot element $\boldsymbol{x}$ around which to partition
  - Grows two regions

    $A[p \ldots i] \leq \boldsymbol{x}$

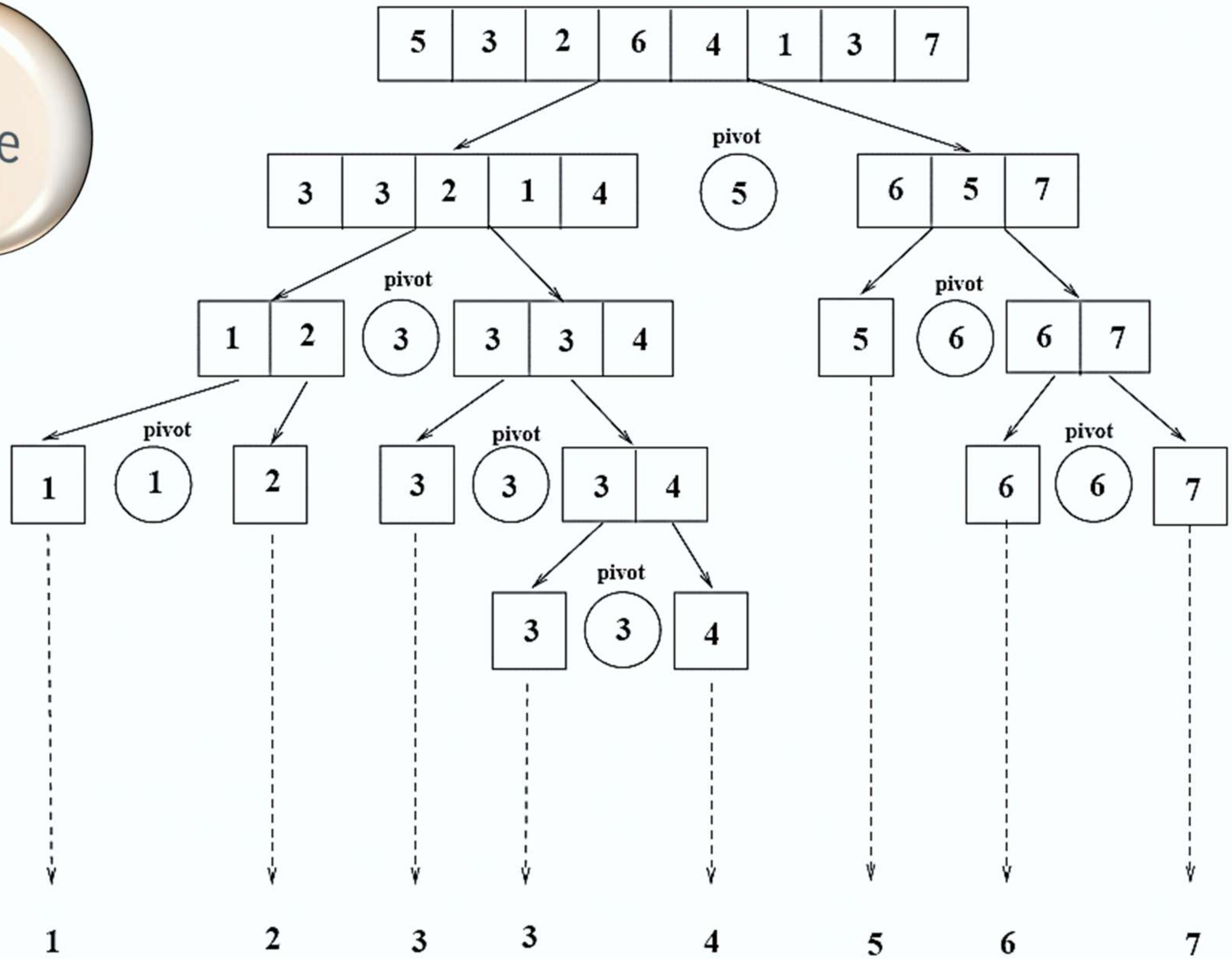
    $\boldsymbol{x} \leq A[j \ldots r]$



Example


pivot x=5
A[p…r]
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 5 4 1 6 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
A[p…q]
A[q+1…r]
3 3 2 1 4 5 6 7
j
i

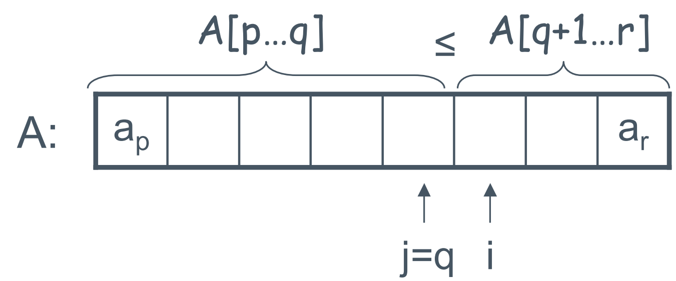
Example


5 3 2 6 4 1 3 7
3 3 2 1 4
pivot
5
6 5 7
1 2
pivot
3
3 3 4
5
pivot
6
6 7
1
pivot
1
2
3
pivot
3
3 4
6
pivot
6
7
3
pivot
3
4
1 2 3 3 4 5 6 7

Partitioning the Array

Alg. PARTITION (A, p, r)

```
x ← A[p]
i ← p – 1
j ← r + 1
while TRUE
        do repeat j ← j – 1
              until A[j] ≤ x
        do  repeat i ← i + 1
              until A[i] ≥ x
         if i < j
              then exchange A[i] ↔ A[j]
           else return j
```



Each element is visited once!

Running time: $\Theta(n)$

$n = r - p + 1$

Recurrence

Alg.: QUICKSORT(A, p, r)

Initially: p=1, r=n

if $p < r$ ***then*** $q \leftarrow$ *PARTITION(A, p, r)*

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence: $T(n) = T(q) + T(n - q) + n$

Worst Case Partitioning

› Worst-case partitioning

- *One region has one element and the other has n – 1 elements*
- *Maximally unbalanced*

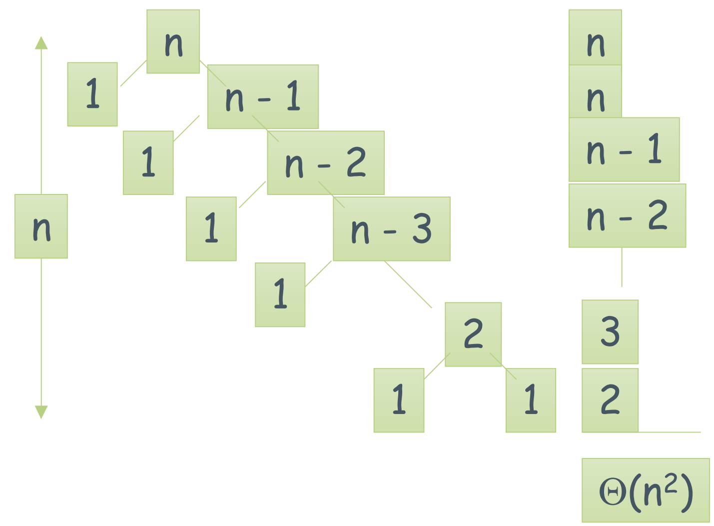
› *Recurrence: q=1*

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

$T(1) = \Theta(1)$

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

$$= n + \left(\sum_{k=1}^{n} k \right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$



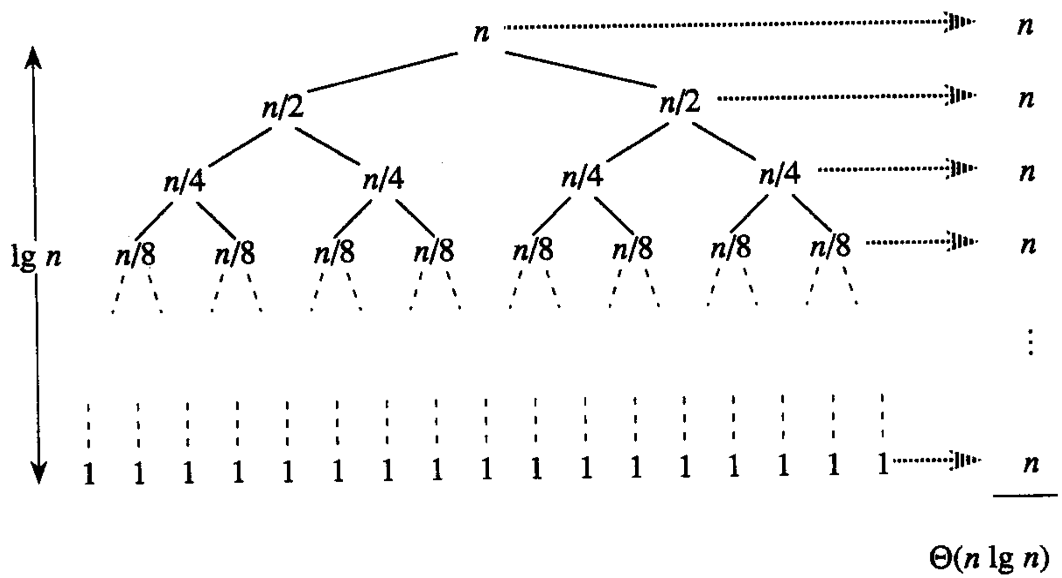
When does the worst case happen?

Best Case Partitioning

- *Best-case partitioning*
  - *Partitioning produces two regions of size n/2*
- *Recurrence: q=n/2*

  $T(n) = 2T(n/2) + \Theta(n)$

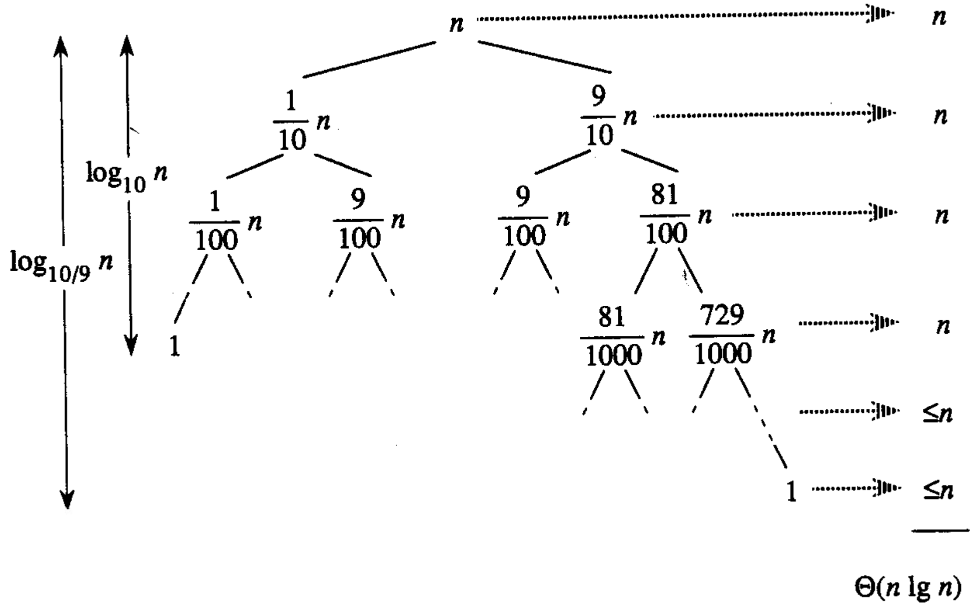
  $T(n) = \Theta(n\lg n)$ *(Master theorem)*



Case Between Worst and Best

- *9-to-1 proportional split*

Q(n) = Q(9n/10) + Q(n/10) + n



- Using the recursion tree:

longest path: $Q(n) \leq n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 nlgn$

shortest path: $Q(n) \geq n \sum_{i=0}^{\log_{10} n} 1 = n \log_{10} n = c_1 nlgn$

Thus, $Q(n) = \Theta(nlgn)$

How does partition affect performance?

- **Any splitting of constant proportionality** yields $\Theta(n lgn)$ time !!!

- Consider the $(1 : n-1)$ splitting:

ratio=$1/(n-1)$ not a constant !!!

- Consider the $(n/2 : n/2)$ splitting:

ratio=$(n/2)/(n/2) = 1$ it is a constant !!

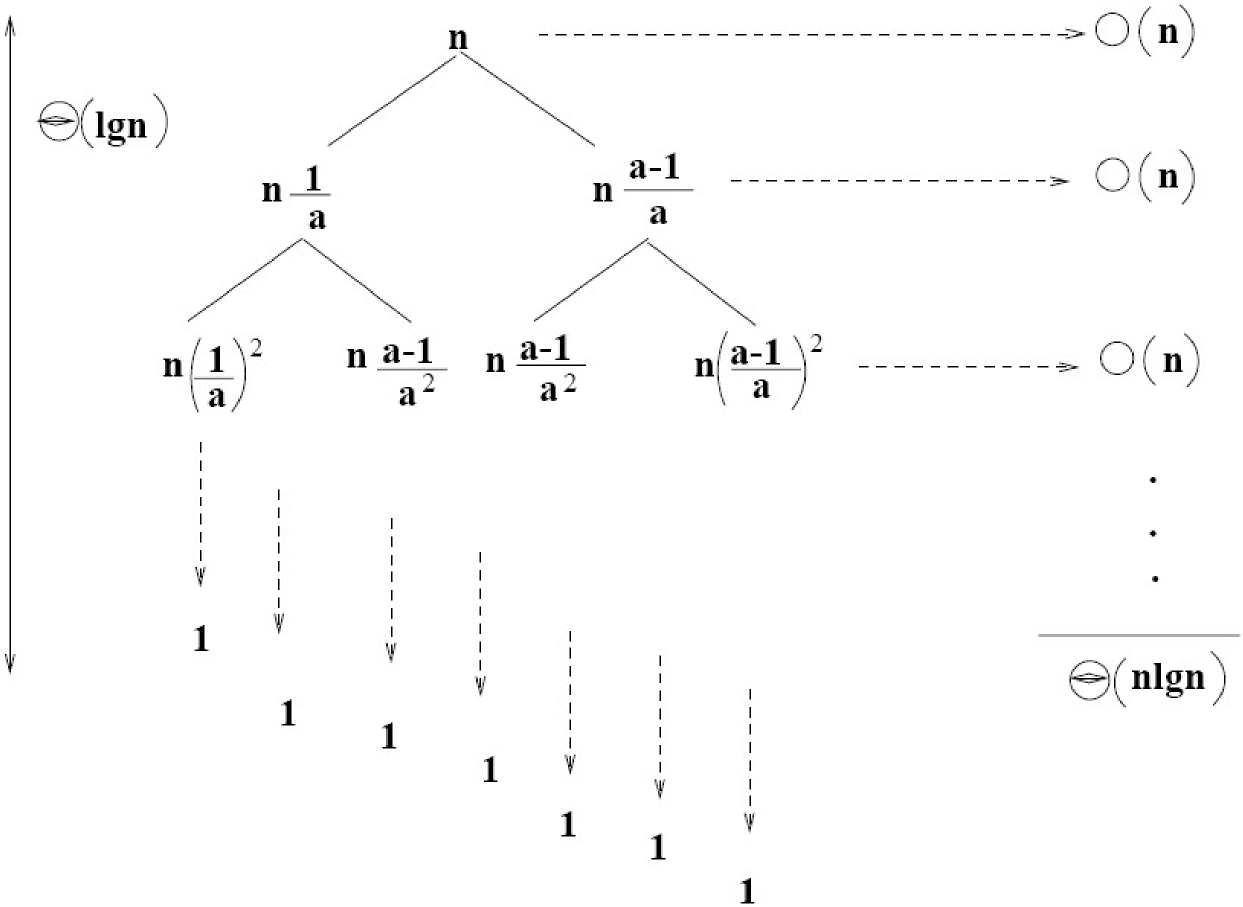
- Consider the $(9n/10 : n/10)$ splitting:

ratio=$(9n/10)/(n/10) = 9$ it is a constant !!

How does partition affect performance?

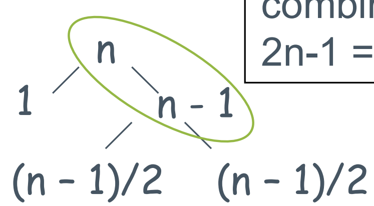
- Any $((a-1)n/a : n/a)$ splitting:

$$\text{ratio}=((a-1)n/a)/(n/a) = a-1 \text{ it is a constant !!}$$



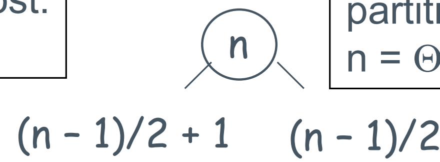
Performance of Quick sort

- *Average case*
  - *All permutations of the input numbers are equally likely*
  - *On a random input array, we will have a **mix** of well balanced and unbalanced splits*
  - *Good and bad splits are randomly distributed across throughout the tree*



combined partitioning cost:
2n-1 = Θ(n)

Alternate of a good and a bad split



partitioning cost:
n = Θ(n)

Nearly well-balanced split

- *Running time of Quicksort when levels alternate between good and bad splits is O(nlgn)*

Summary

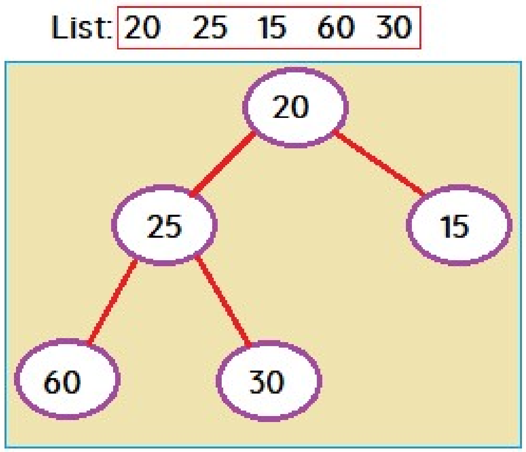
- *Merge and quick sort are both divide and conquer based sorting technique.*
  - *Partitioning of array (Problem divide into sub problems)*
  - *recursive calling for each sub problems.*
  - *Combining result of each sub problem*
- *Merge sort is not in place sort because t need extra space to combine or merge values*
- *In quick sort focus is on pivot element which help to divide the array/problem into sub problems.*
- *Complexity of quick sort vary because it depends on good selection of pivot element.*

Heap Sort

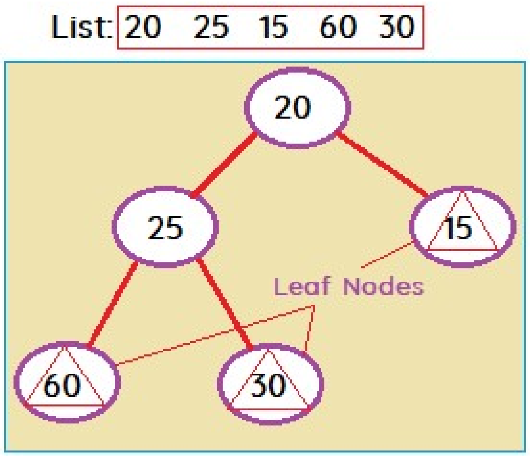
It is a comparison-based sorting technique based on binary heap data structure

Heap Sort

- 20 25 15 60 30
- Build Max or Min Heap as binary tree
- Child as left as possible



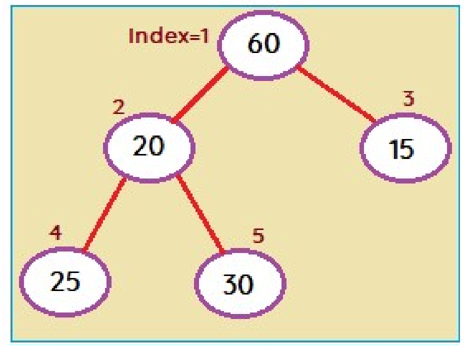
- Max heap Property:
  - Parent Node > Child node
  - Leaf node either be min or max heap – no problem



Heap Sort

- 20 25 15 60 30
- Build Max or Min Heap as binary tree
- Max heap Property:
  - Parent Node > Child node

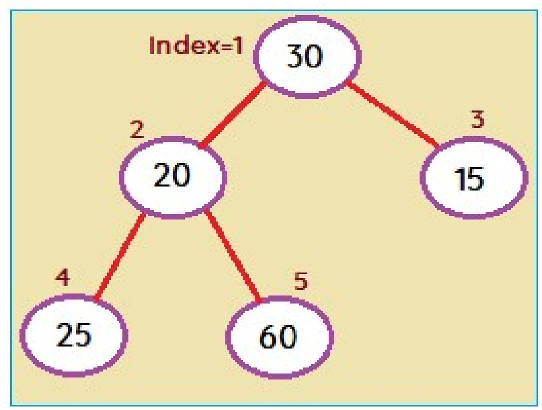
Build Max heap



Array Representation of Binary Tree

Index	1	2	3	4	5
Value	60	20	15	25	30

- Latest Element 30 added to tree, will be replaced with root element 60.
- New root node: 30, Delete 60 as it is the maximum element.

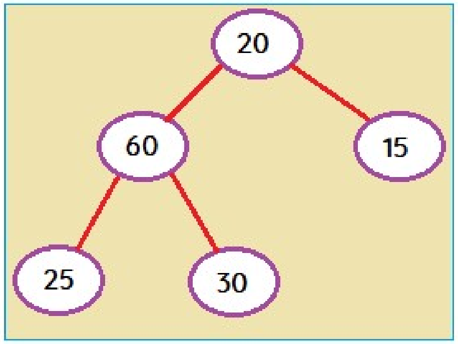


Index	1	2	3	4	5
Value	60	20	15	25	30
Value	30	20	15	25	60

Heap Sort

- 20 25 15 60 30
- Build Max or Min Heap as binary tree
- Max heap Property:
  - Parent Node > Child node

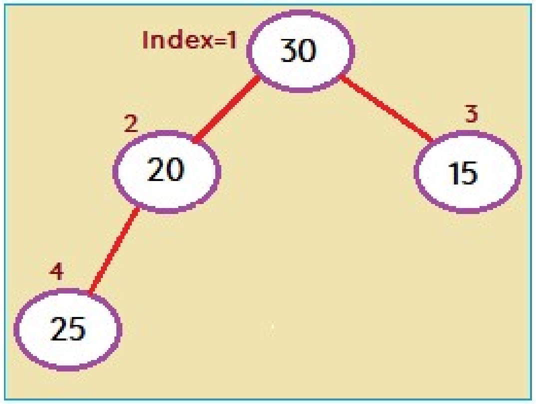
Build Max heap



Array Representation of Binary Tree

Index	1	2	3	4	5
Value	60	20	15	25	30

- Last element is now at the right place, delete it from the tree.

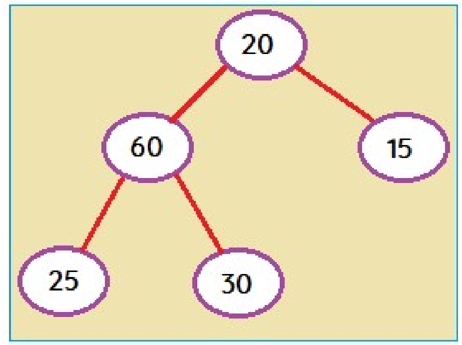


Index	1	2	3	4	5
Value	60	20	15	25	30
Value	30	20	15	25	60

Heap Sort

- 20 25 15 60 30
- Build Max or Min Heap as binary tree
- Max heap Property:
  - Parent Node > Child node

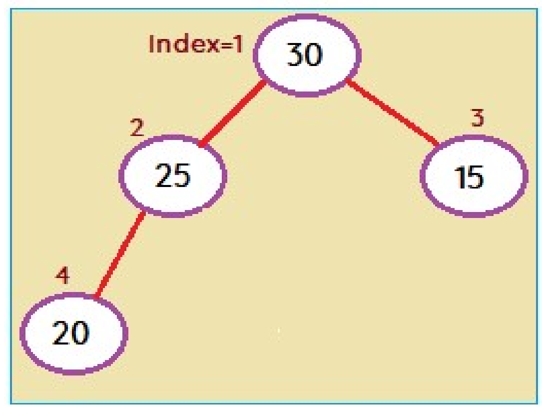
Build Max heap



Array Representation of Binary Tree

Index	1	2	3	4	5
Value	60	20	15	25	30

- Now check for Max heap, if not then swap 2nd and 4th node values

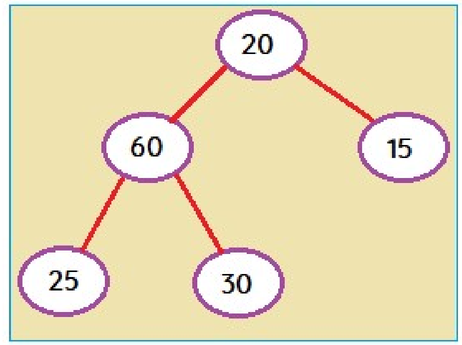


Index	1	2	3	4	5
Value	60	20	15	25	30
Value	30	20	15	25	60
Value	30	25	15	20	60

Heap Sort

- 20 25 15 60 30
- Build Max or Min Heap as binary tree
- Max heap Property:
  - Parent Node > Child node

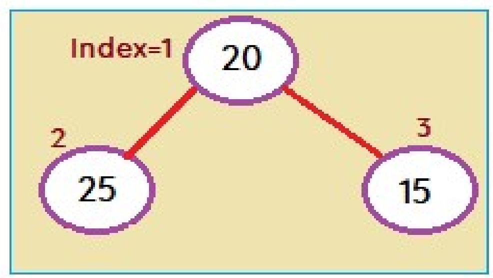
Build Max heap



Array Representation of Binary Tree

Index	1	2	3	4	5
Value	60	20	15	25	30

- Delete root element with the most recent added element to perform swap operation on 2nd and 4th element.
- The new tree after deletion of a node and array (RHS)

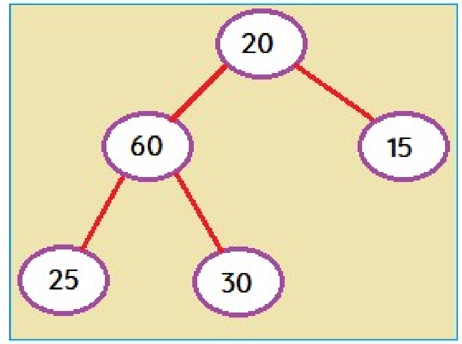


Index	1	2	3	4	5
Value	60	20	15	25	30
Value	30	20	15	25	60
Value	20	25	15	30	60

Heap Sort

- 20 25 15 60 30
- Build Max or Min Heap as binary tree
- Max heap Property:
  - Parent Node > Child node

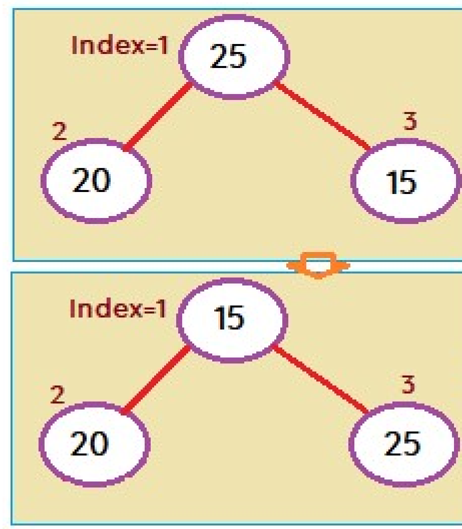
Build Max heap



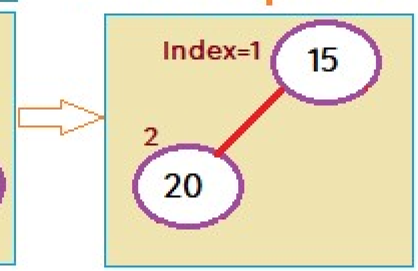
Array Representation of Binary Tree

Index	1	2	3	4	5
Value	60	20	15	25	30

- To make the tree as max heap, swap root and 2nd node
- Delete root node by replacing it with the last node.



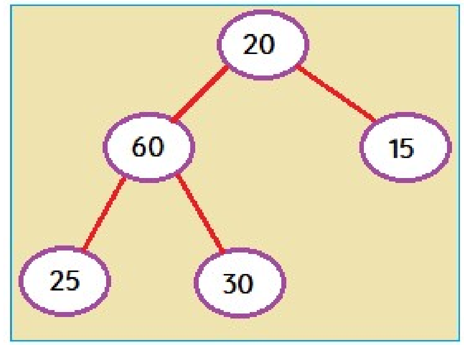
Index	1	2	3	4	5
Value	60	20	15	25	30
Value	30	20	15	25	60
Value	25	20	15	30	60
Value	15	20	25	30	60



Heap Sort

- 20 25 15 60 30
- Build Max or Min Heap as binary tree
- Max heap Property:
  - Parent Node > Child node

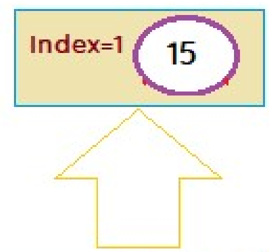
Build Max heap

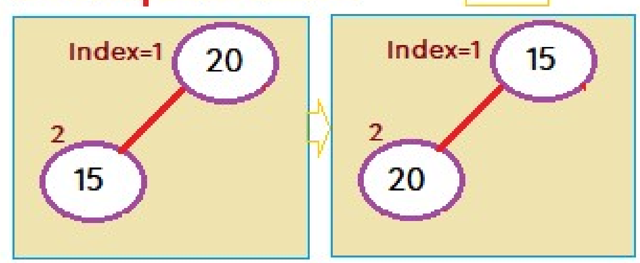


Array Representation of Binary Tree

Index	1	2	3	4	5
Value	60	20	15	25	30

Index	1	2	3	4	5
Value	60	20	15	25	30
Value	30	20	15	25	60
Value	25	20	15	30	60
Value	20	15	25	30	60
Value	15	20	25	30	60

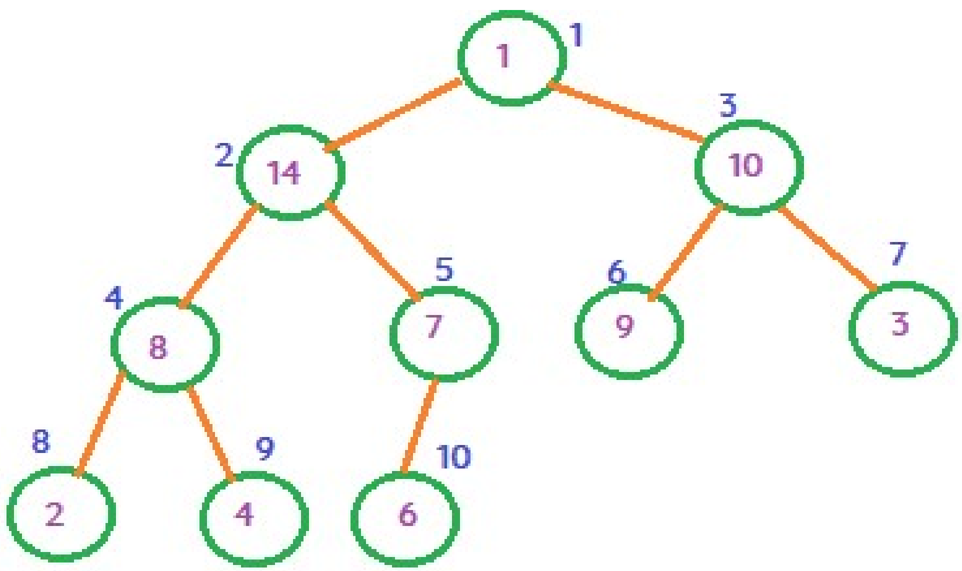




- Build Max heap to bring max elements at root
- Delete root element by Swapping elements (Tree and Array)

How to Start from the tree?

- Leaf nodes: 6, 7, 8, 9, 10 (either max or min heap)
- Heapify: Non-Leaf Nodes
- Total Nodes: n=10
- Formula to use
  - $\left\lfloor \frac{n}{2} \right\rfloor + \mathbf{1}\ \boldsymbol{to}\ \boldsymbol{n}$ are leaf nodes
  - $\left\lfloor \frac{n}{2} \right\rfloor$ to 1 are non leaf nodes



Heap Sort Algorithm

$\boldsymbol{Step - 1: Build\_MAX\_HEAP(A)}$

$A.heap\_size = A.length$

$FOR\ i = \left\lfloor \frac{A.length}{2} \right\rfloor\ to\ 1$ // loop for non-leaf nodes

Max $\_heapify(A, i)$

$\boldsymbol{Step - 2: Max\_heapify(A, i)}$

$l = 2i;$ // left child

$r = 2i + 1$ // right child

$IF\ (l \leq A.heap\_size\ \&\&\ A[l] > A[i])\ THEN$

$largest = l$

$ELSE$

$largest = i$

$END\ IF$

$IF\ (r \leq A.heap\_size\ \&\&\ A[r] > A[largest])\ THEN$

$largest = r$

$END\ IF$

$IF\ (largest \neq i)\ THEN$

$Exchange\ A[i]\ with\ A[largest]$

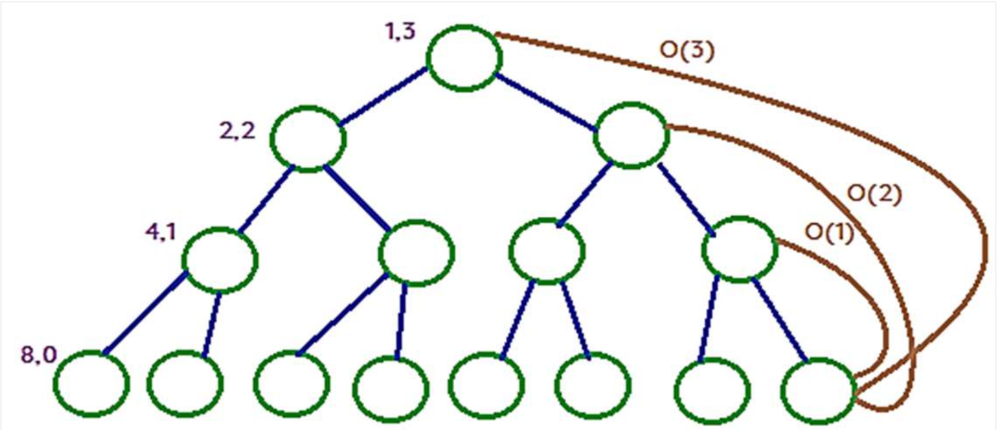
max $\_heapify(A, largest)$

END IF

$\boldsymbol{Step - 3: HEAP\_SORT(A, i)}$

Swap $(A[l], A[i])$ // Delete Operation performed

max $\_heapify(A, l)$



$\boldsymbol{Number\ of\ Nodes\ of\ tree\ can\ be\ determined}$

$\boldsymbol{if\ height\ of\ the\ tree\ is\ given\ i.e.h = \left\lceil \frac{n}{2^{h+1}} \right\rceil}$

$$\boldsymbol{Total\ Time = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(n) = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h} * 2} \right\rceil O(cn)}$$

harmoni progression which result is equal to 2.

$$\boldsymbol{= \frac{cn}{2} \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h}} \right\rceil = = \frac{cn}{2} * 2 = O(n) \longrightarrow to\ build\ heap}$$

$\boldsymbol{To\ heapify: T(n) = O(\log n)}$

$\boldsymbol{\therefore\ Overall\ Time\ Complexity\ is\ O(n \log n)}$

Thank You!!!

Have a good day