Foundations of Sorting Algorithms

{ik} INTERVIEW KICKSTART

{ik} INTERVIEW KICKSTART

Overview

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)

Along the way, learn how to analyse algorithms in general (time and space) for their performance.

Sorting problem definition

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7

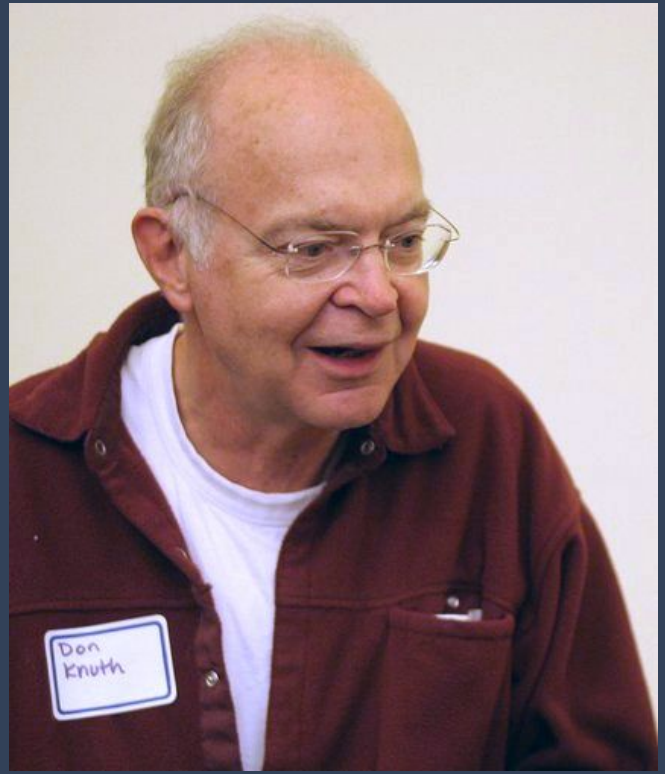
OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

“Computer manufacturers of the 1960s estimated that more than 25% of the running time on their computers was spent on sorting... in fact, there were many installations in which the task of sorting was responsible for more than half of the computing time.” (Donald Knuth, *The Art of Computer Programming, Vol 3*)



Applications

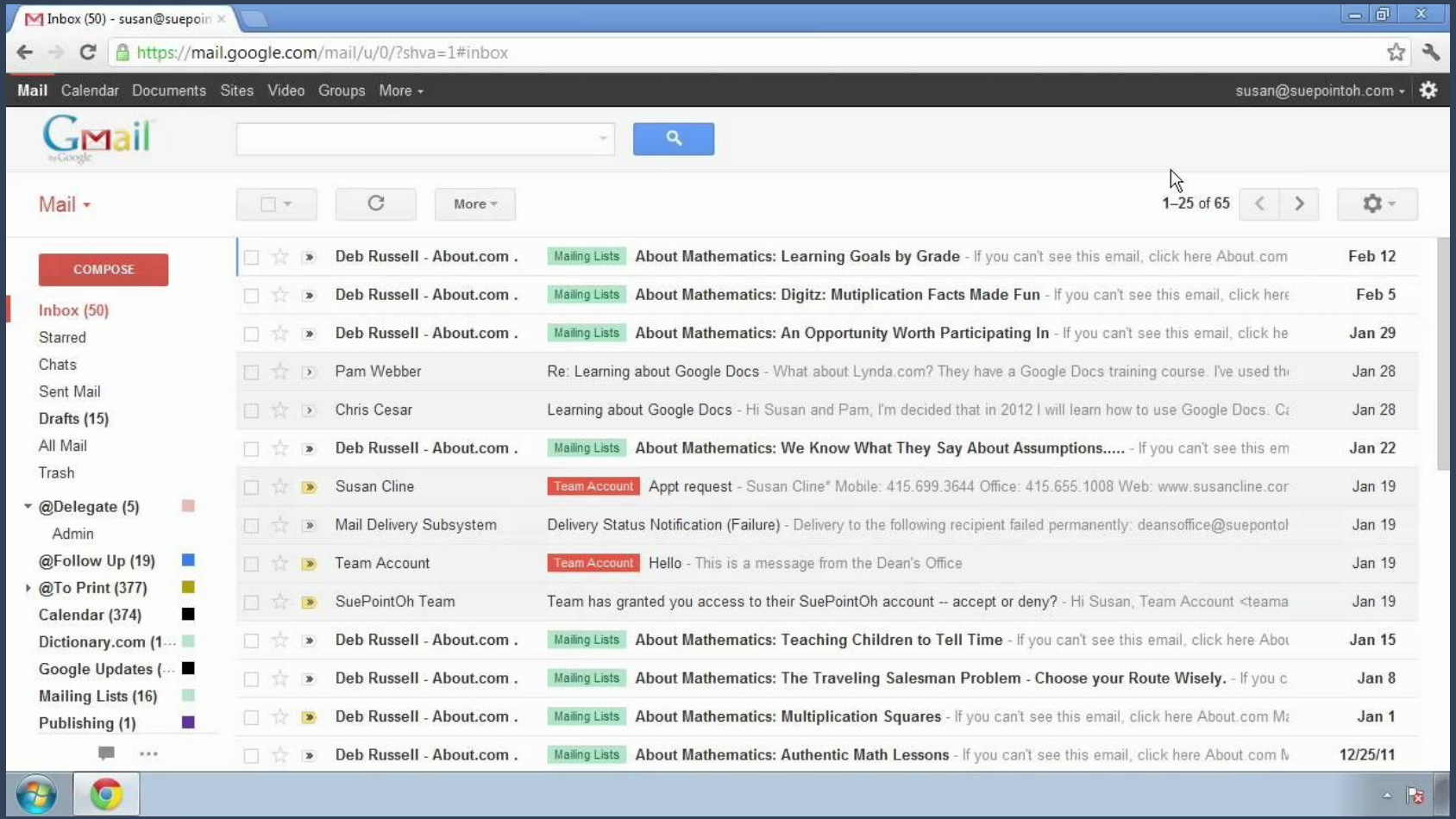
Searching for a value is faster in a sorted array than an unsorted array.

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.



Other Benefits

Sorting provides a valuable case study of how to attack programming problems in general. E.g, algorithm design strategies like “Decrease and conquer”, and “Divide and conquer”.

Sorting provides an excellent illustration of how to analyse algorithms to determine their performance, so that you can then choose between competing methods.

Internal Sorting

Memory locations A, A+1, A+2, A+3, ... A+(n-1) contain n numbers.

Write a computer program that rearranges these numbers, if necessary, so that they are in ascending order.

Design strategy 1: Brute force

- Simplest design strategy
- The most straightforward approach, usually based on the problem statement.

"Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves."
(Alan Kay, ACM Turing Award 2003)

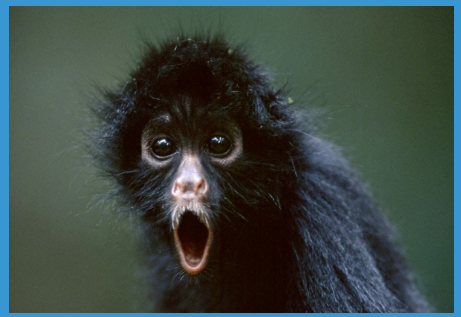
Monkey Sort / Stupid Sort / Bogo Sort

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

Is this a good algorithm?

{ik} INTERVIEW KICKSTART

Correctness and Efficiency



The algorithm appears to be correct.

But even if we had a machine that could run 10^8 operations per second, and even if we could generate and test one permutation in a single operation, we would need almost 800 years for an array of size 20.

For an array of size 100, it would take 10^{142} years.

A better brute force algorithm

Locate the smallest item and put it in the first place.

Then select the next smallest item and put it in the second place. And so on...

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7

OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

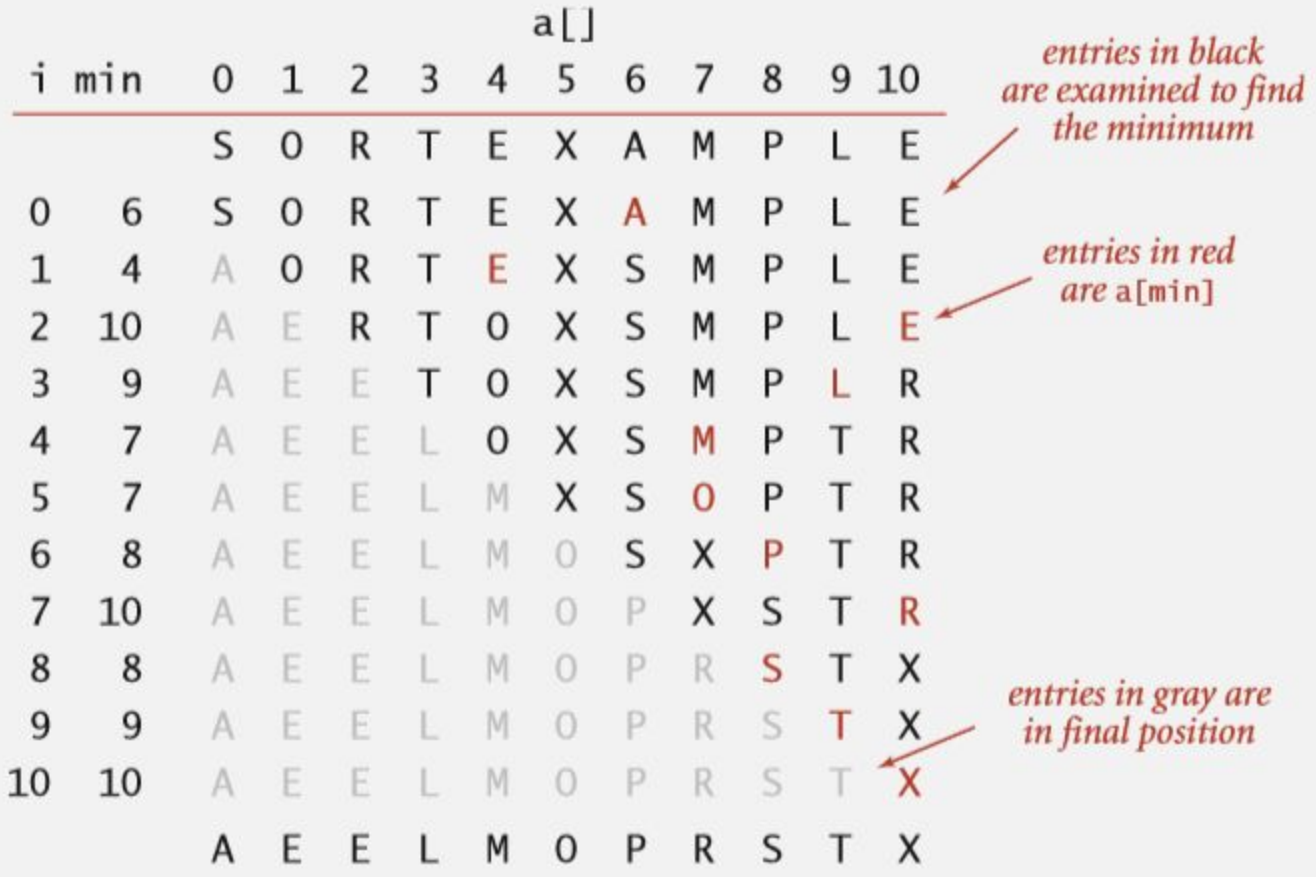
A brute force algorithm

Locate the smallest item and put it in the first place.

Then select the next smallest item and put it in the second place. And so on...

Sorting by repeated selection:
“Selection Sort”

Final output ordering generated one by one in sequence.



Trace of selection sort (array contents just after each exchange)

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]
```

How do we analyse this algorithm?

Selection Sort

```
def selectionsort(a):
    for i in range(len(a)):
        #Find the ith smallest element
and swap it with a[i]
        min = i
        for inner in range(i, len(a)):
            if a[inner] < a[min]:
                min = inner
        (a[i],a[min]) = (a[min],a[i])
```

How do we analyse this algorithm?

What does it mean to analyse an algorithm?

1. Establish its correctness.
2. Quantify its “efficiency”.
    a. Time: How long does the algorithm take to run? (“Running time”)
    b. Space: How much memory does the algorithm use to run?

Why is running time given more importance than space?

Running time depends on speed of the processor, which is not easy to change.

Memory is easier to augment.

How do we measure running time?

We can code up the algorithm as a program, and run it on a machine. Measure how much time (in seconds) it takes to run.

But that value would be different for different machines. It may be different for different programming languages. It may be different for different inputs.

Better to focus on system-independent factors here, and leave system-dependent factors to the experts.

Input size clearly matters, and is system-independent

A larger array will take a longer time to sort.

So why not define running time as a function of input size ?

T(n): where n is the input size.

How do we compute an expression for T(n) in terms of n?

How do we measure running time?

For an input of size n, measure running time of the algorithm as the “number of basic operations” in the pseudocode, in a way that it can be ‘counted’ even before writing it as a program.

How do we measure running time?

Number of basic operations in a high-level language.
Examples of basic operations, with a fixed execution time, but different constant values for each:

```
hours_until_dinner = 2                  //Variable assignment

class_end_time = t + 2.5                //Simple arithmetic operation on a variable

s = “Hope you are not already feeling sleepy!”  //Variable assignment

if (sleeping_time < class_end_time)
    then print “Watch recording later”      //Slightly longer basic operation
```

" *It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings.* " — *Alan Turing*

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

Two basic operations in sorting algorithms

These would dominate the running time.

Comparison: `if a[inner] < a[min]:`

Exchange/Swap: `(a[i],a[min]) = (a[min],a[i])`

Proposition. Selection sort uses $(N-1)+(N-2)+\ ...\ +1+0 \sim N^2/2$ compares and N exchanges.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Running time insensitive to input. Quadratic time, even if input is sorted.

Data movement is minimal. Linear number of exchanges.

Running time expression for selection sort

$c_1*N^2/2 + c_2*N + c_3$

Running time expression for selection sort

$c_1*N^2/2 + c_2*N + c_3$

Which term dominates as N becomes large?

Running time expression for selection sort

$c_1*N^2/2 + c_2*N + c_3$

Which term dominates as N becomes large?

The first term. So let's drop the lower-order terms. To make the analysis system-independent, we also ignore the constant factors.

The asymptotic complexity becomes $O(N^2)$

A variation of the brute force algorithm

To get the minimum elements one by one, we scanned the array from left to right and kept track of the next minimum.

Instead, we could have scanned the array from right to left, and bubbled up the minimum to the left by repeated exchanges.

```
Repeat n times:

    While scanning the array from right to left:

        if A[i-1] > A[i], swap them
```

```
def bubblesort(a):
    for start in range(len(a)):
        for i in range(len(a)-1,start,-1):
            if a[i-1] > a[i]:
                (a[i-1],a[i]) = (a[i],a[i-1])
```

“I think the bubble sort is the wrong way to go”

Running time expression for bubble sort

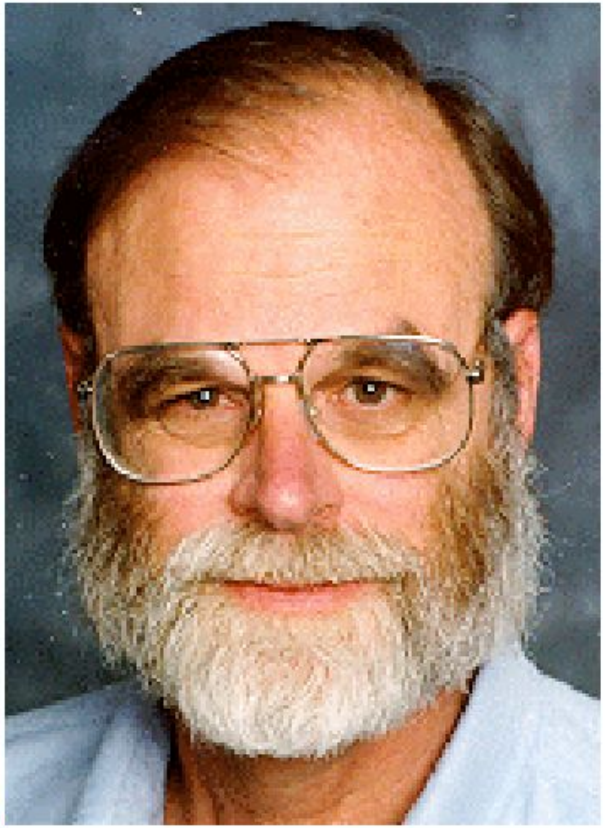
$c_1*N^2/2 + c_2*N + c_3$

The asymptotic complexity becomes $O(N^2)$

Jim Gray (Turing 1998)

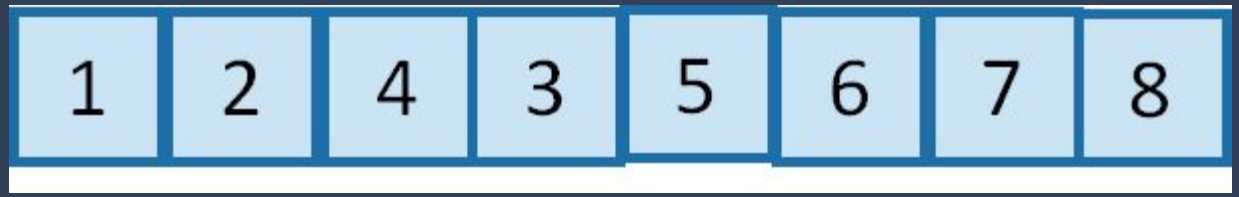
- **Bubble sort is a good argument for analyzing algorithm performance. It is a perfectly correct algorithm. But it's performance is among the worst imaginable. *So, it crisply shows the difference between correct algorithms and good algorithms.***

(italics mine)



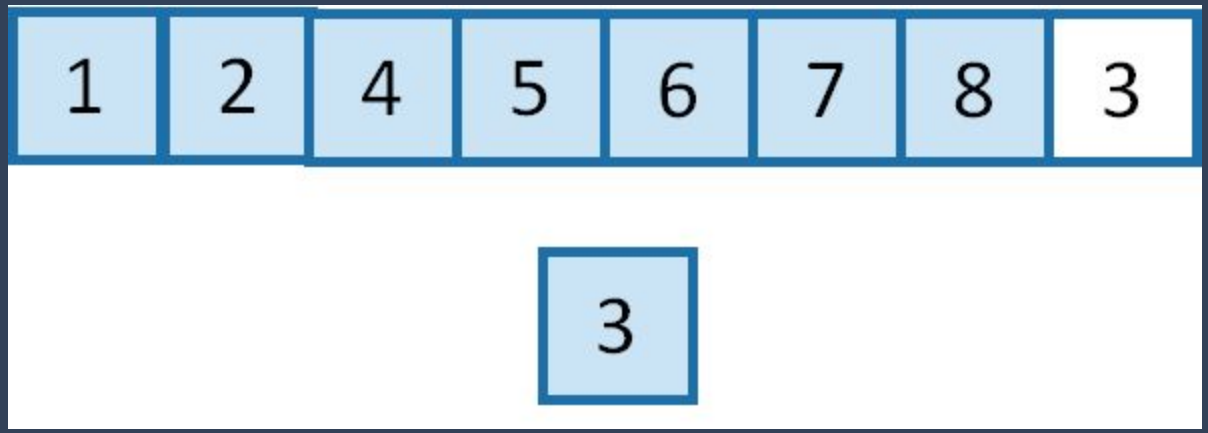
Design strategy 2: Decrease-and-conquer

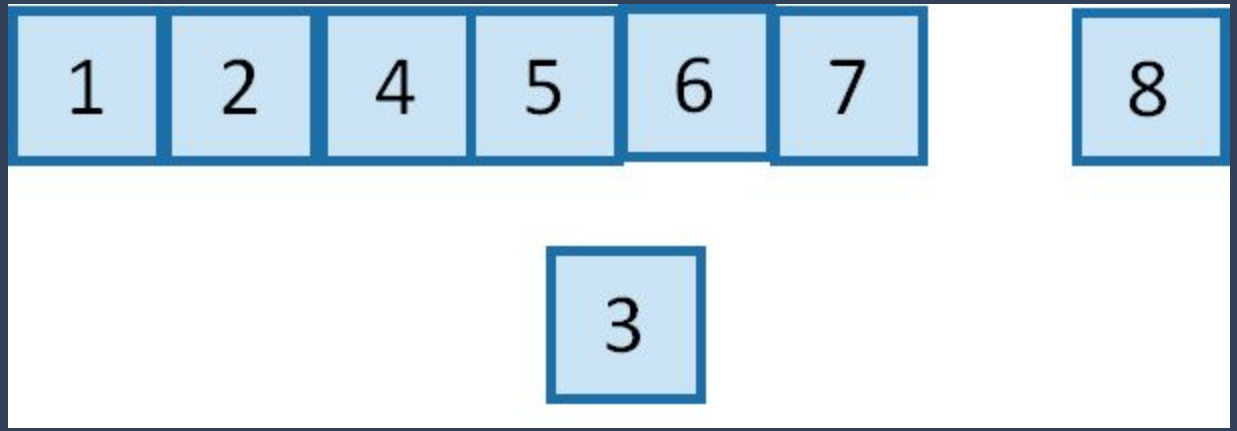
- Decrease the problem of size n to size n-1
- Assume you have solved the problem of size n-1
- Solve the problem of size n using the solution to the problem of size n-1
- Recursive (if top-down) and iterative (if bottom-up)

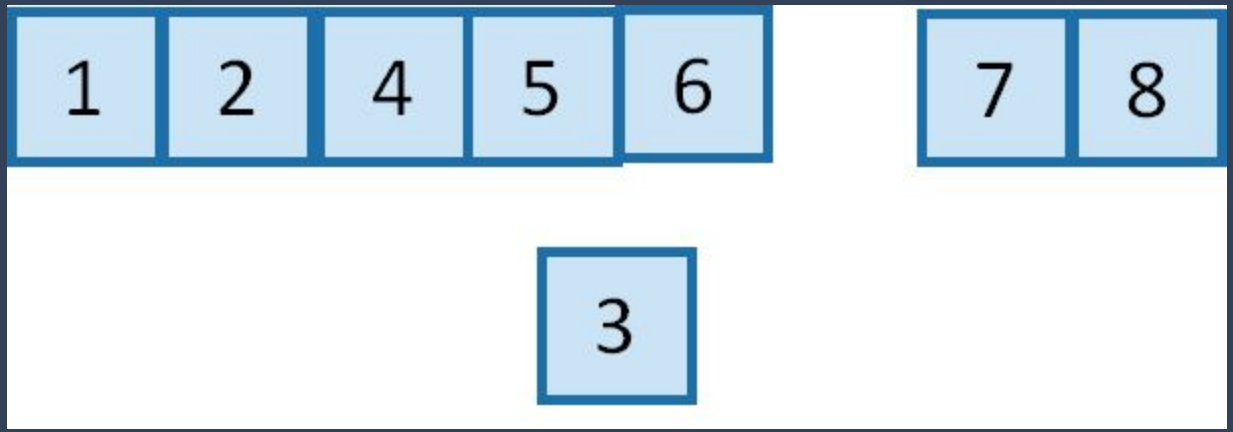


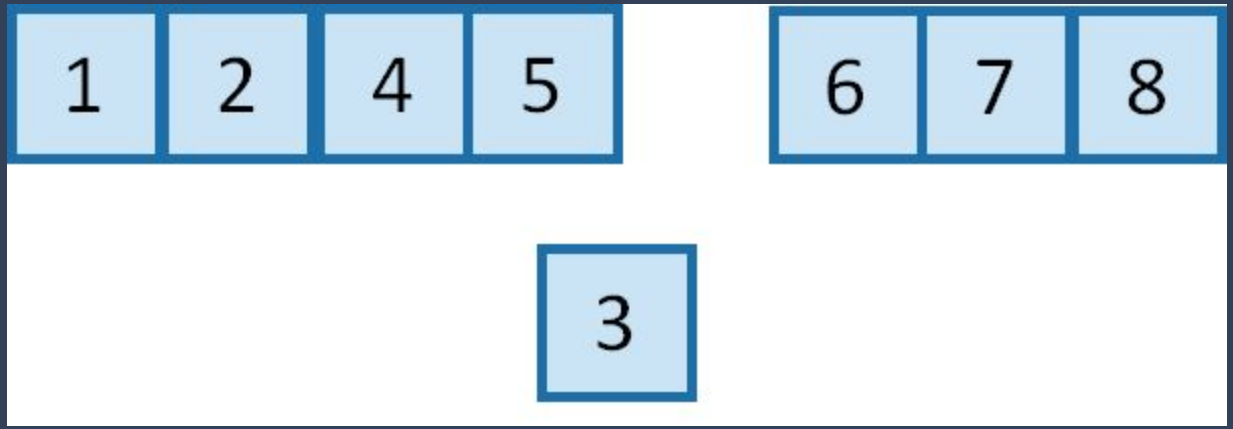
```
def insertionsort(A,n): //array A[1..n]
   if (n <= 1) return;
   insertionsort(A,n-1);
   //now insert the nth element into its place.
   j = n-1
   while j >= 1 and A[j] > A[j+1]:
      swap(A[j+1],A[j])
      j = j - 1
   return
```

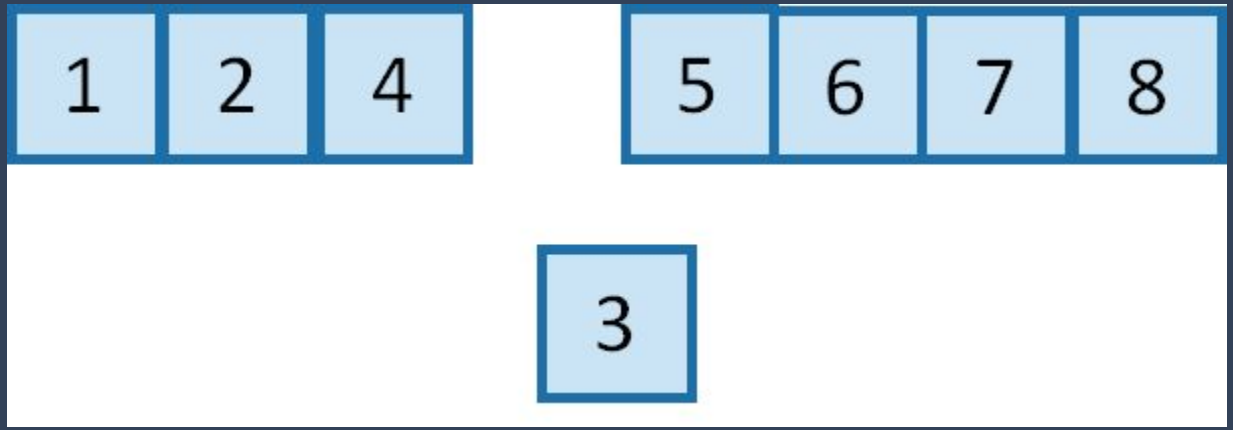
Recursive version
(Top-down)

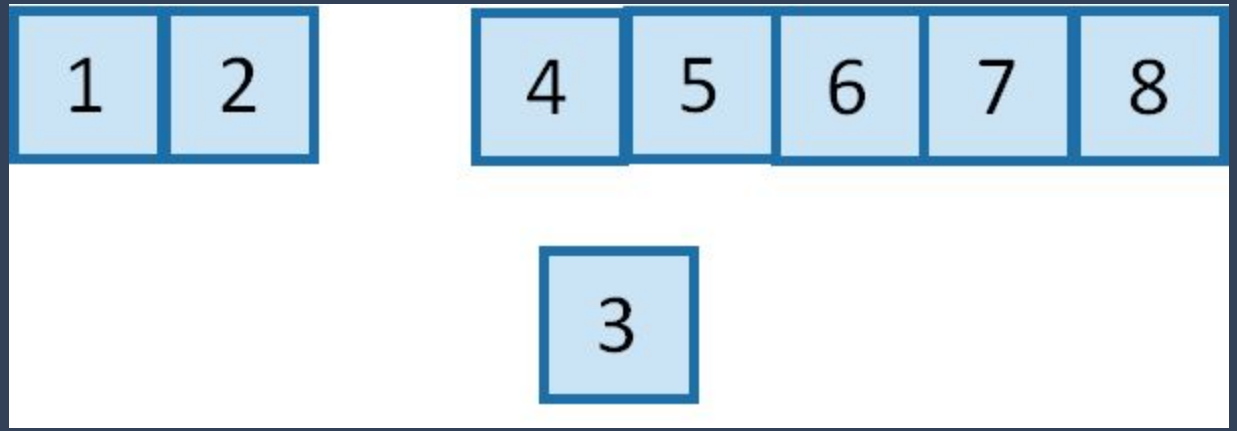










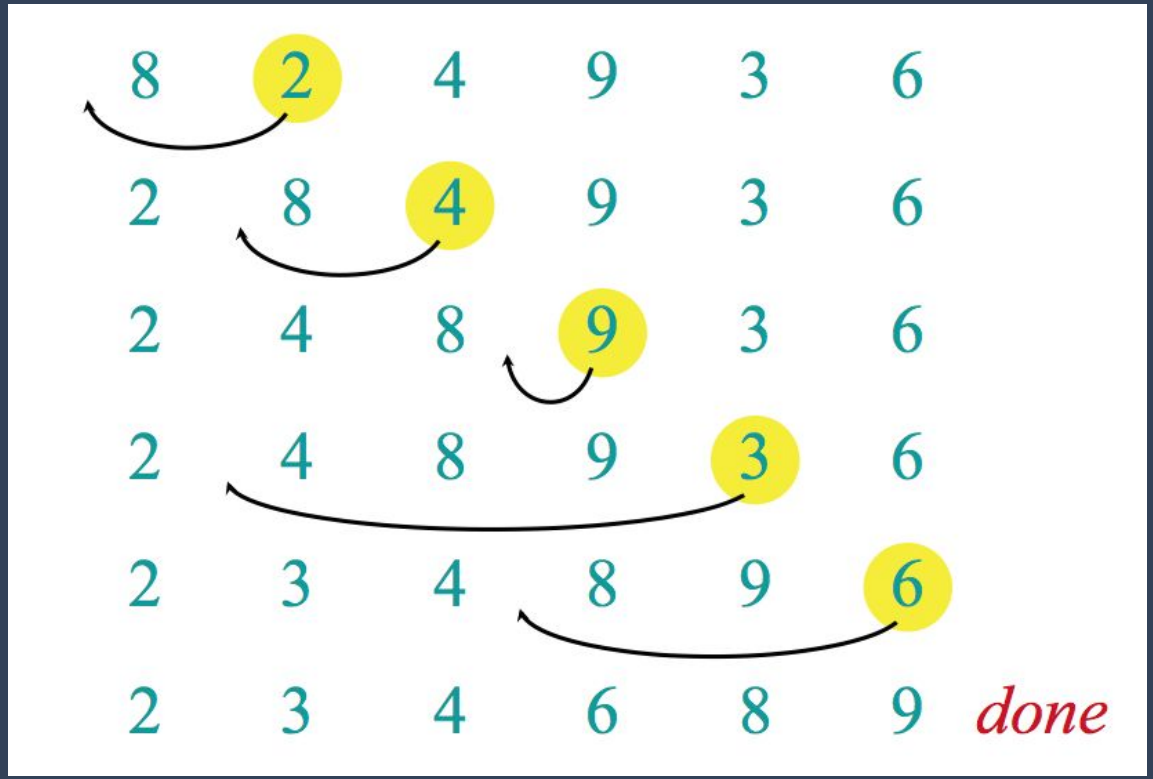


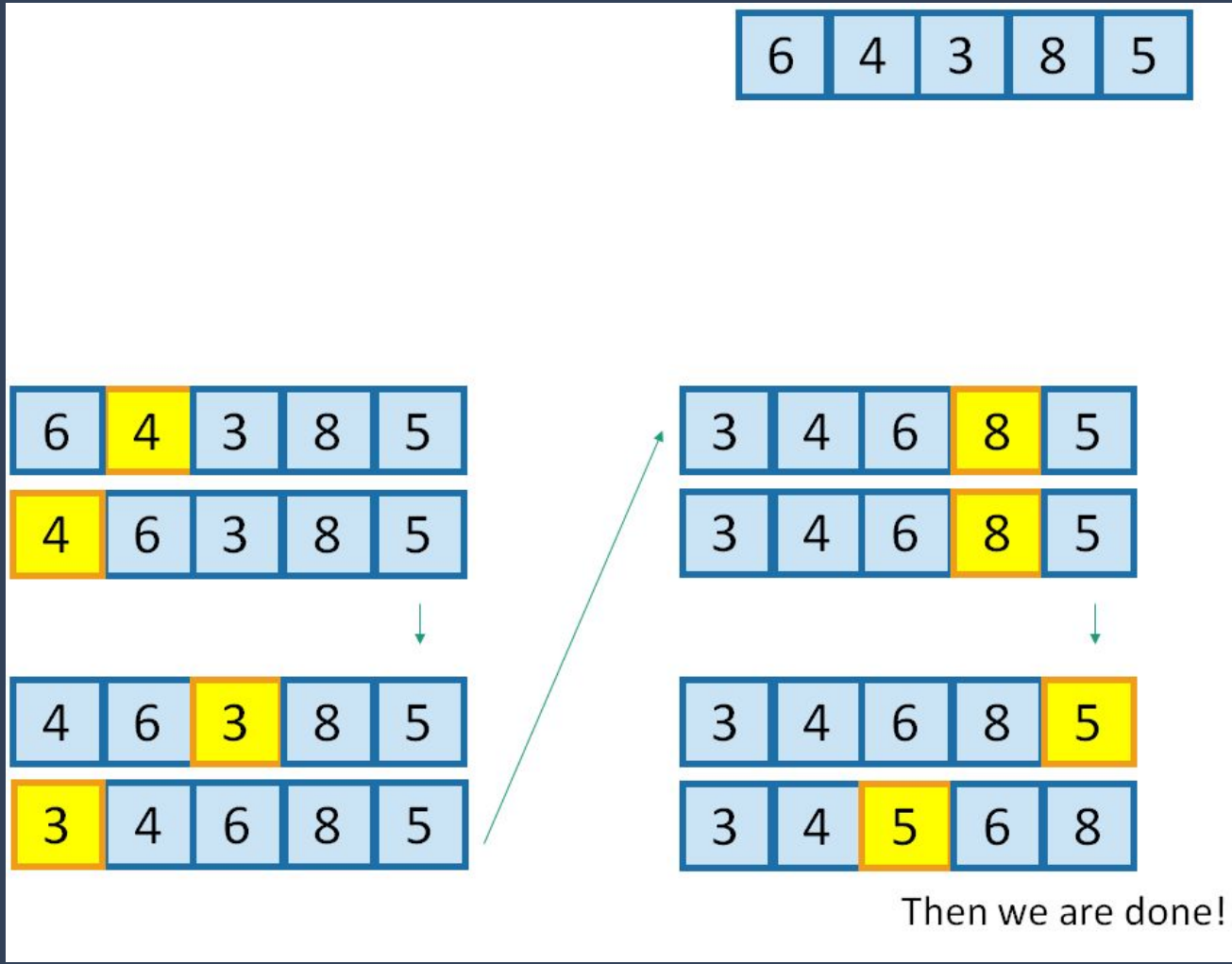
```
def insertionsort(A,n): //array A[1..n]
  if (n <= 1) return;
  insertionsort(A,n-1);
  //now insert the nth element into its place.
  nth = A[n]
  j = n-1
  while j >= 1 and A[j] > nth:
     A[j+1] = A[j]
     j = j - 1
  A[j+1] = nth
  return
```

Slightly more efficient: shift elements to right instead of swapping repeatedly

```
def insertionsort(A): //array A[1..n]
  if (n <= 1) return;
  for i = 2 to n:
    ith = A[i]
    j = i-1
    while j >= 1 and A[j] > ith:
       A[j+1] = A[j]
       j = j - 1
    A[j+1] = ith
  return
```

Iterative version (bottom-up)





Time complexity T(n) = ?

```
def insertionsort(A,n): //array A[1..n]
  if (n <= 1) return;
  insertionsort(A,n-1);
  //now insert the nth element into its place.
  nth = A[n]
  j = n-1
  while j >= 1 and A[j] > nth:
     A[j+1] = A[j]
     j = j - 1
  A[j+1] = nth
  return
```

T(n-1)

c_1*(#shift operations)

Remaining lines = c_2

```
def insertionsort(a):
    if len(a) < 2:
        return
    for i in range(1,len(a)):
        j = i-1
        curr_element = a[i] #Find the right spot for it
        while j >= 0 and curr_element < a[j]:
            a[j+1] = a[j]
            j -= 1
        a[j+1] = curr_element
```

(Recursive)

$$T(n) = T(n-1) + c_1*(\#\text{shift operations}) + c_2$$

(Iterative)

$$T(n) = \sum_{i=1}^{n} [c_1*(\#\text{shift operations for } i^{th} \text{ number}) + c_2]$$

#shift operations for i^{th} number could be

- As small as 0 (best-case)
- As large as i (worst-case)
- On average i/2 (average-case)

Not only input size but nature of input clearly matters

We defined running time T(n) as a function of input size n.

But now we see that for the same input size n, there can be a variety of running times, depending on the nature of the input array.

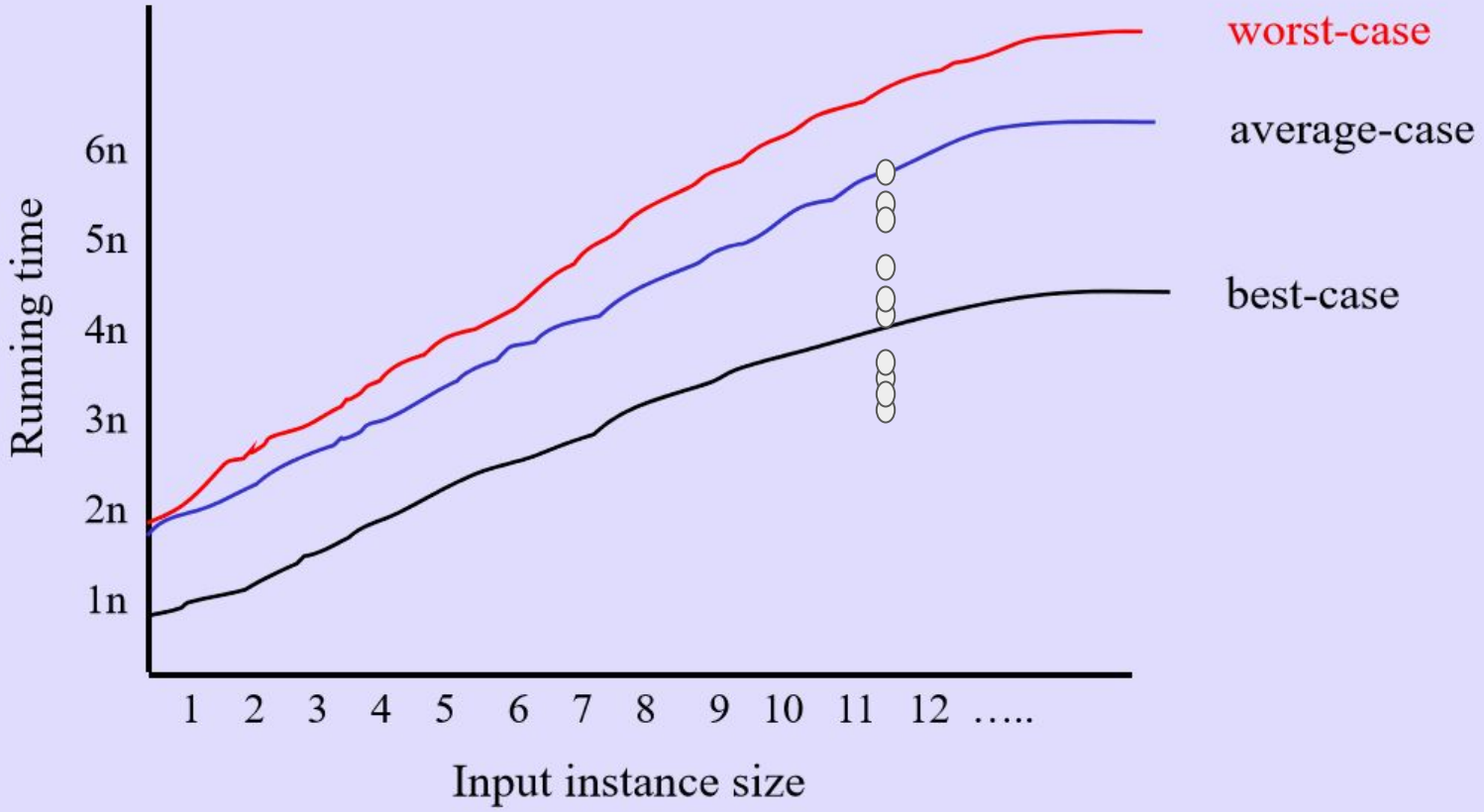
What if we took the maximum of all of them?

T(n) = **Worst-case** running time for input of size n.

What if we took their average?

T(n) = **Average-case** running time for input of size n.

We can also define a best-case running time, but it is not useful.



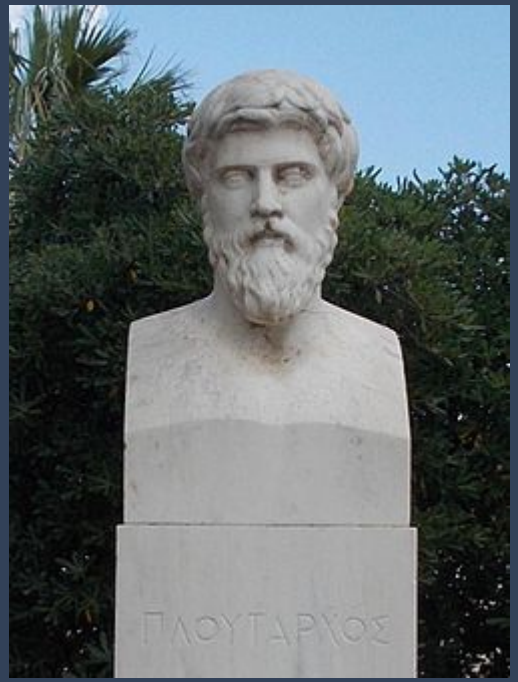
Time complexity of insertion sort

Worst-case = $O(n^2)$

Average-case = $O(n^2)$

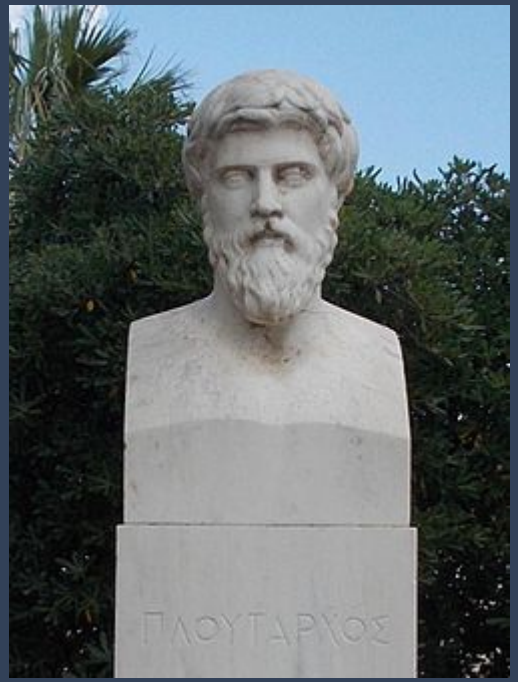
Best-case = $O(n)$

“He (Sertorius) called a general assembly and introduced before it two horses, one utterly weak and already quite old, the other large-sized and strong, with a tail that was astonishing for the thickness and beauty of its hair. By the side of the feeble horse stood a man who was tall and robust, and by the side of the powerful horse another man, small and of a contemptible appearance. At a signal given them, the strong man seized the tail of his horse with both hands and tried to pull it towards him with all his might, as though he would tear it off; but the weak man began to pluck out the hairs in the tail of the strong horse one by one. (Plutarch, *The Parallel Lives*)



Decrease-and-conquer > Brute force

The strong man gave himself no end of trouble to no purpose, made the spectators laugh a good deal, and then gave up his attempt; but the weak man, in a trice and with no trouble, stripped his horse's tail of its hair. Then Sertorius rose up and said: "Ye see, men of my allies, that perseverance is more efficacious than violence, and that many things which cannot be mastered when they stand together yield when one masters them little by little. (Plutarch, *The Parallel Lives*)



“Divide and rule”

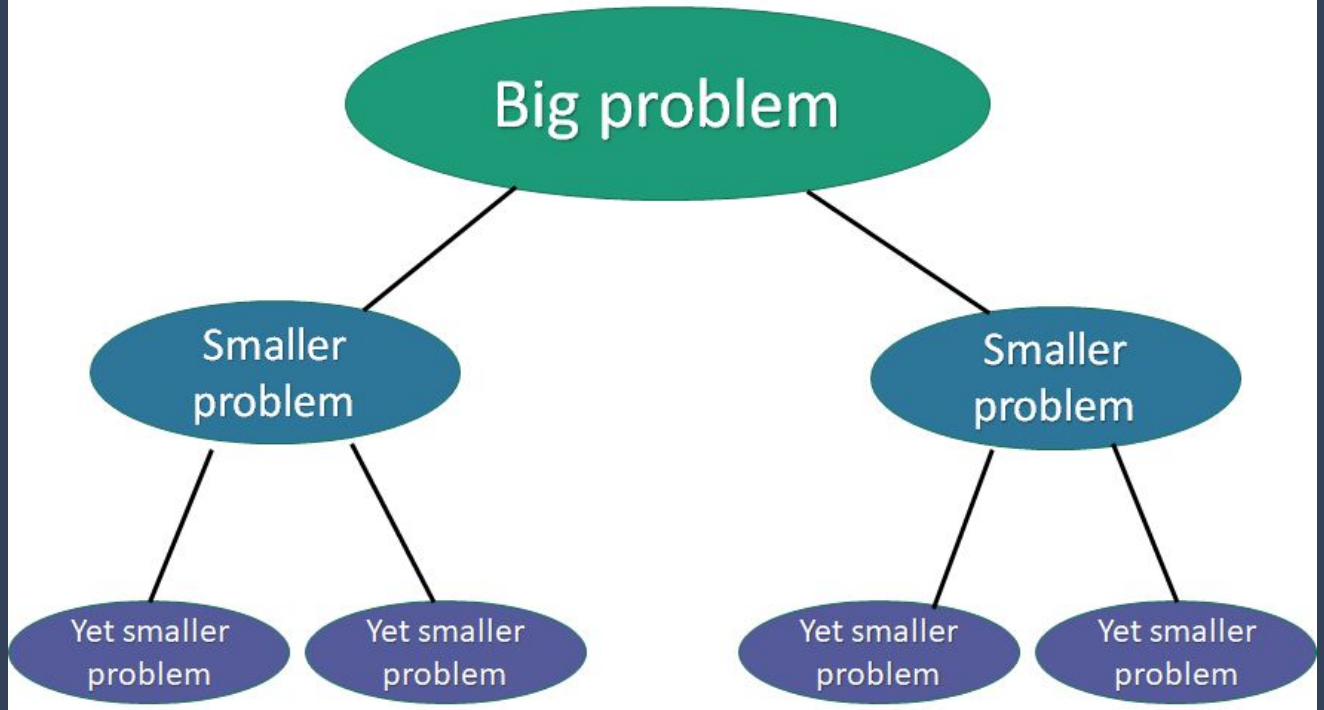
(attributed to Philip II of Macedon)

Design strategy 3: Divide-and-conquer

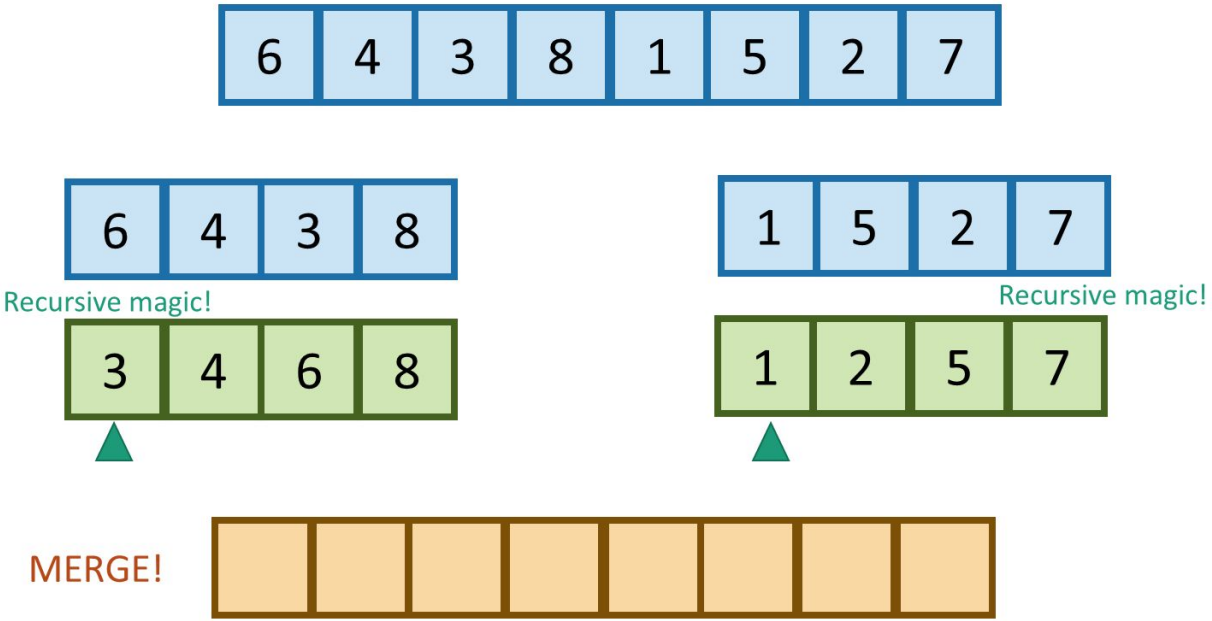
- **Divide** the problem into several smaller instances of the problem (most often two), generally of same size
- **Solve** the smaller instances (typically using recursion)
- **Combine** the solutions to the smaller instances to get the solution to the original problem.

Divide and conquer

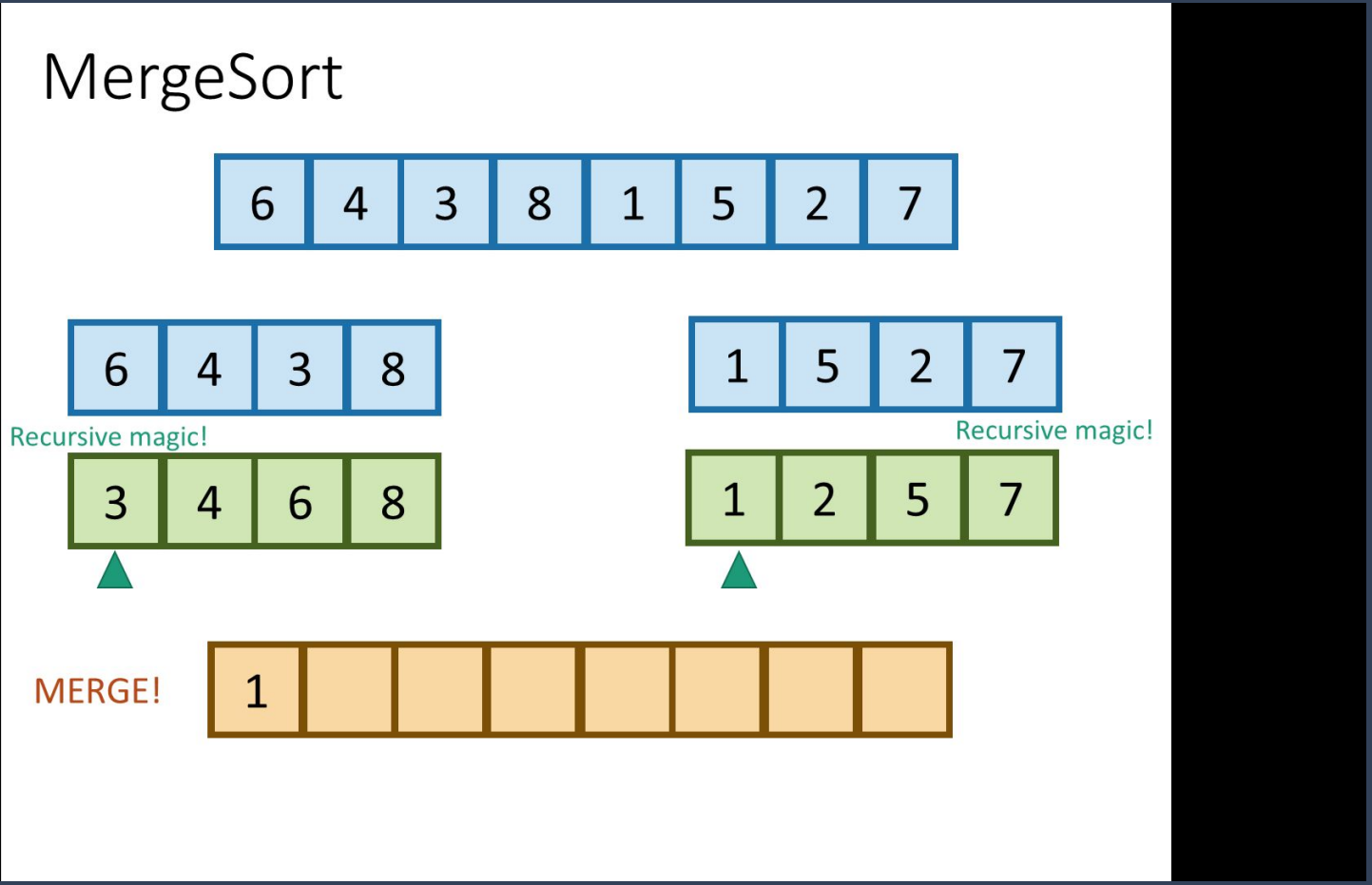
Break problem up into smaller (easier) sub-problems



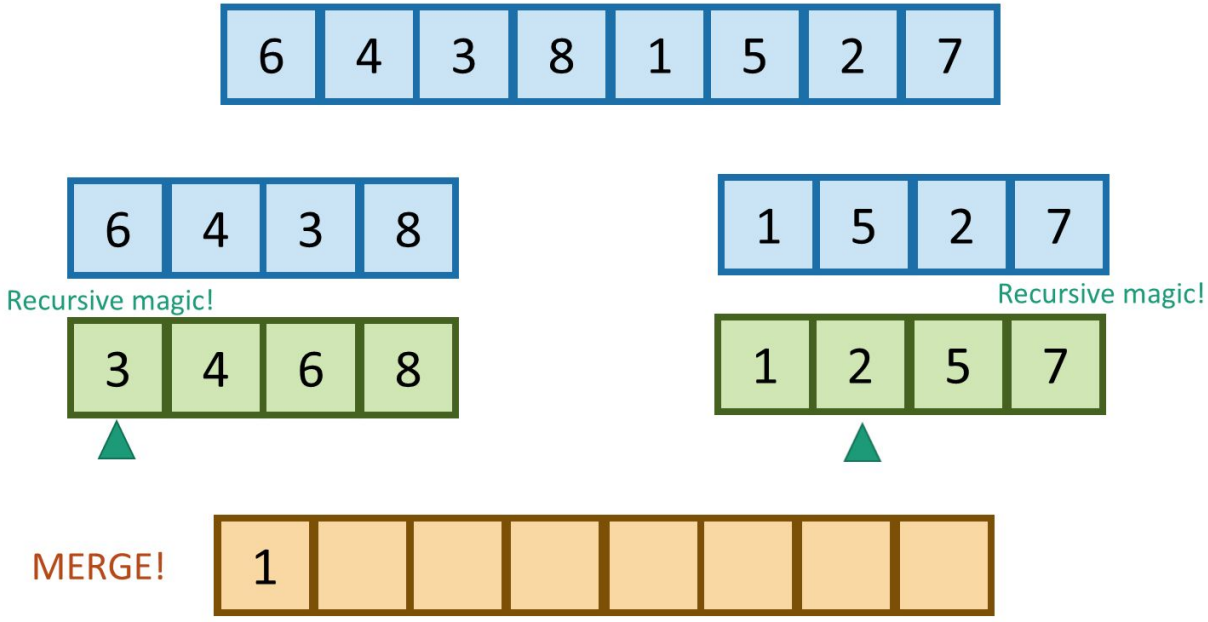
MergeSort



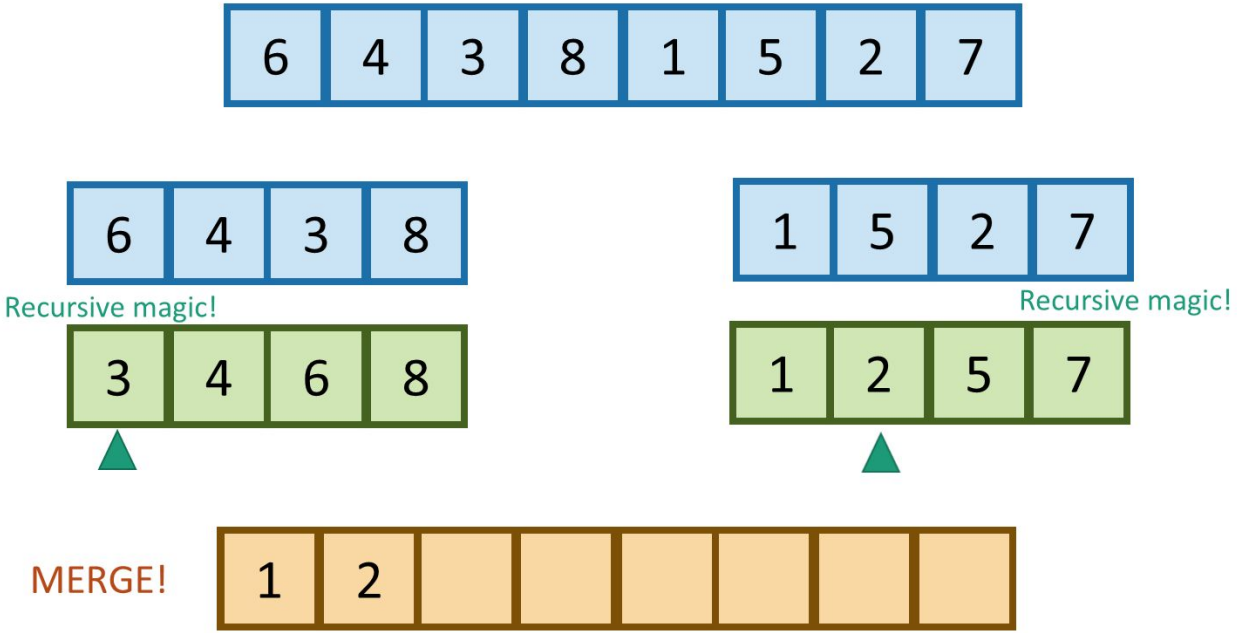
MergeSort



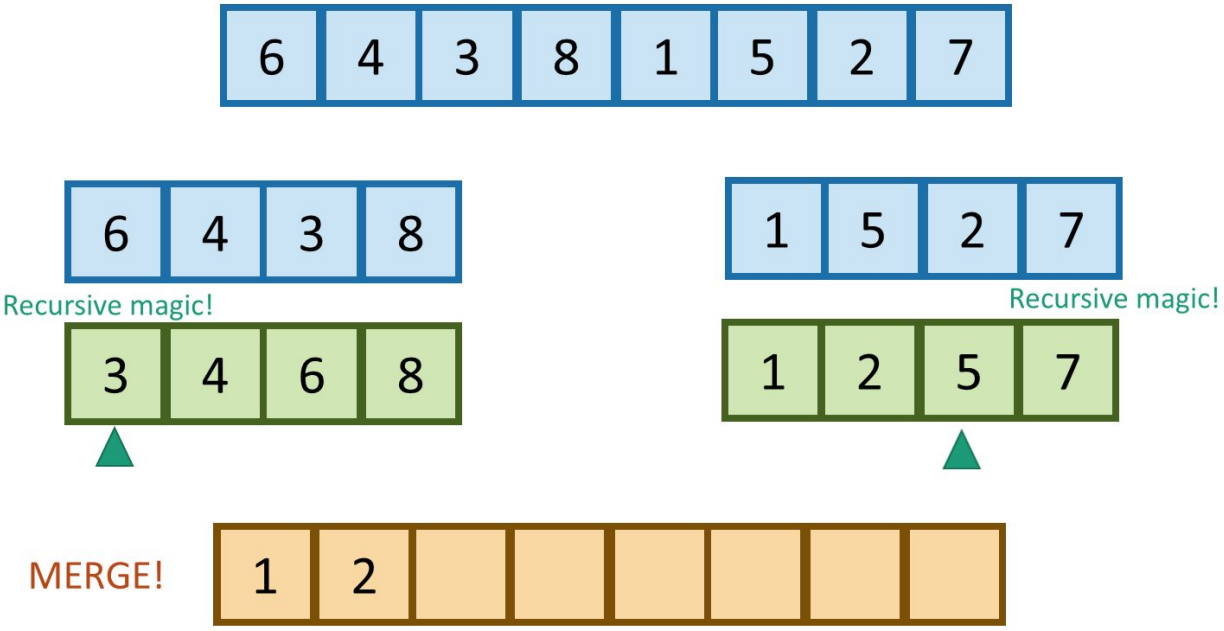
MergeSort



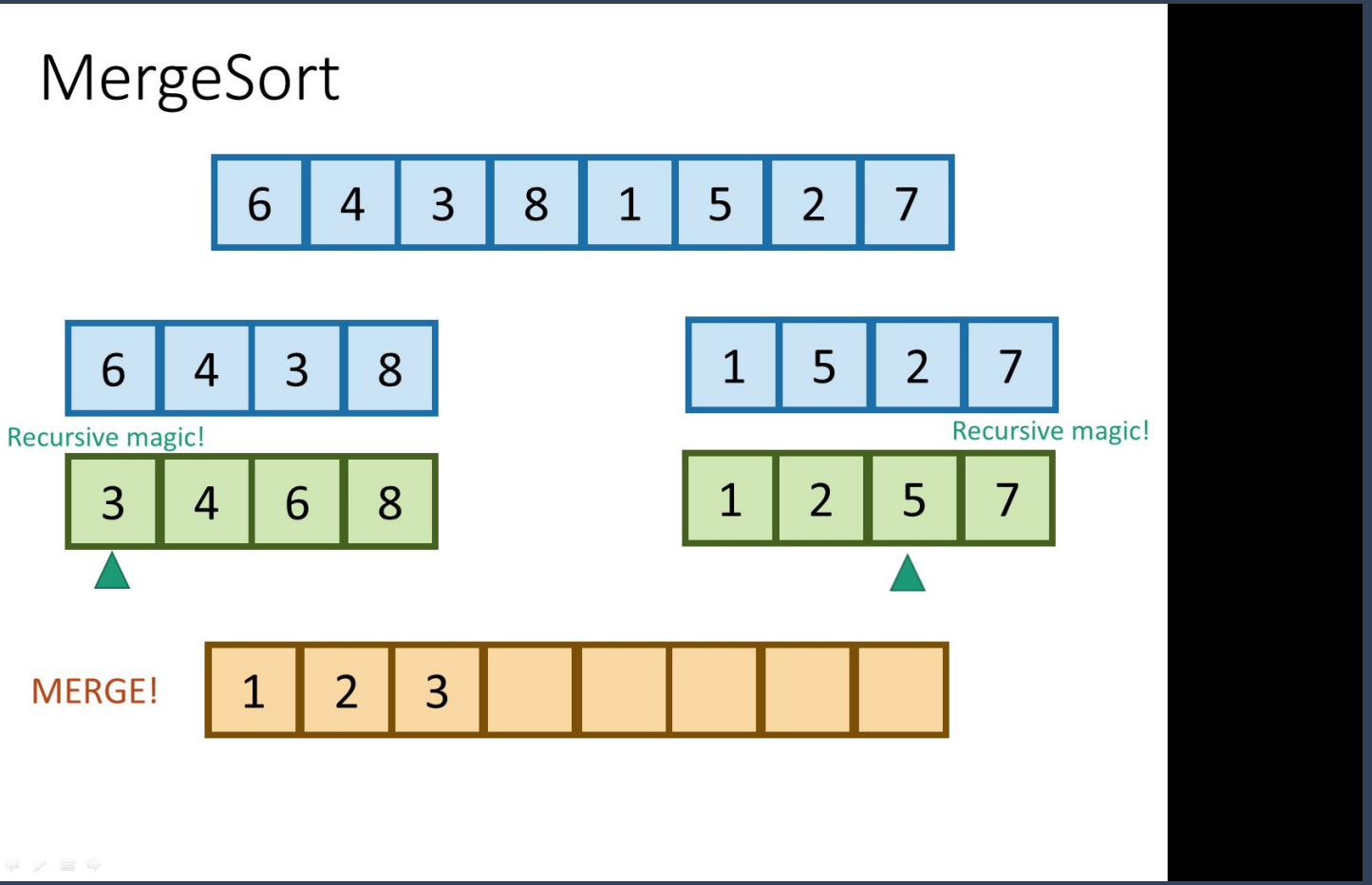
MergeSort



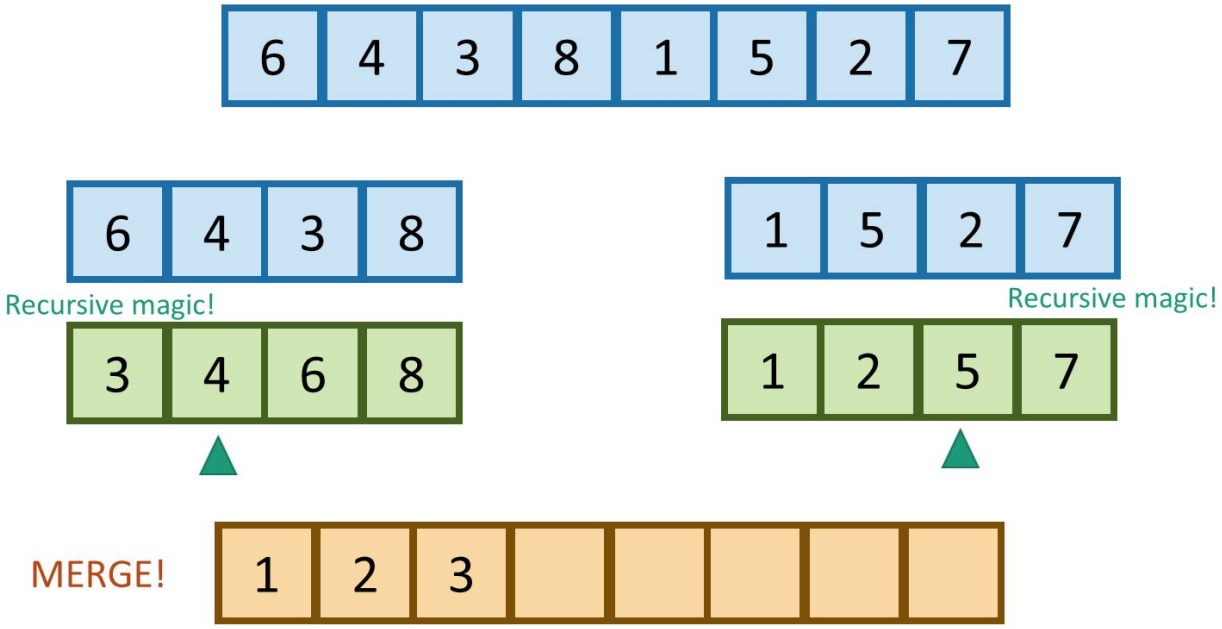
MergeSort



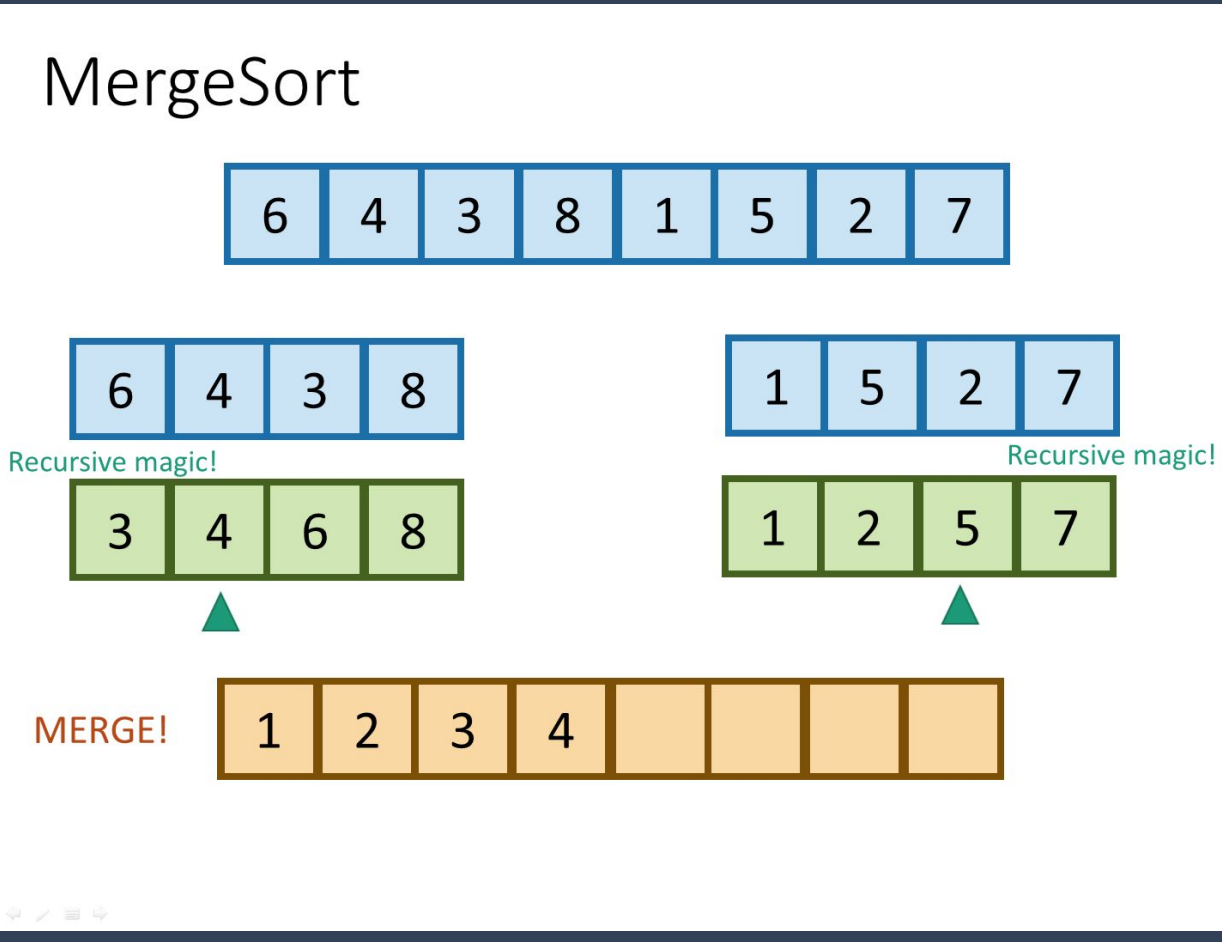
MergeSort



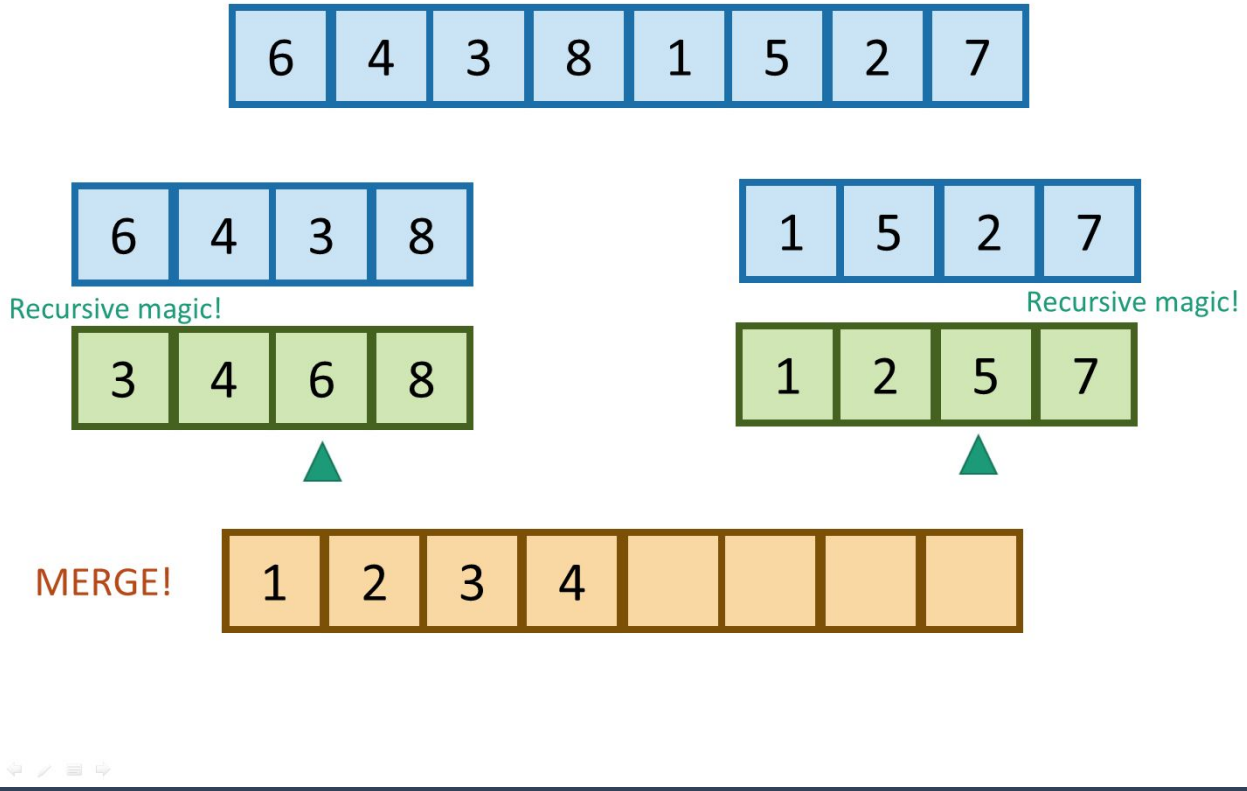
MergeSort



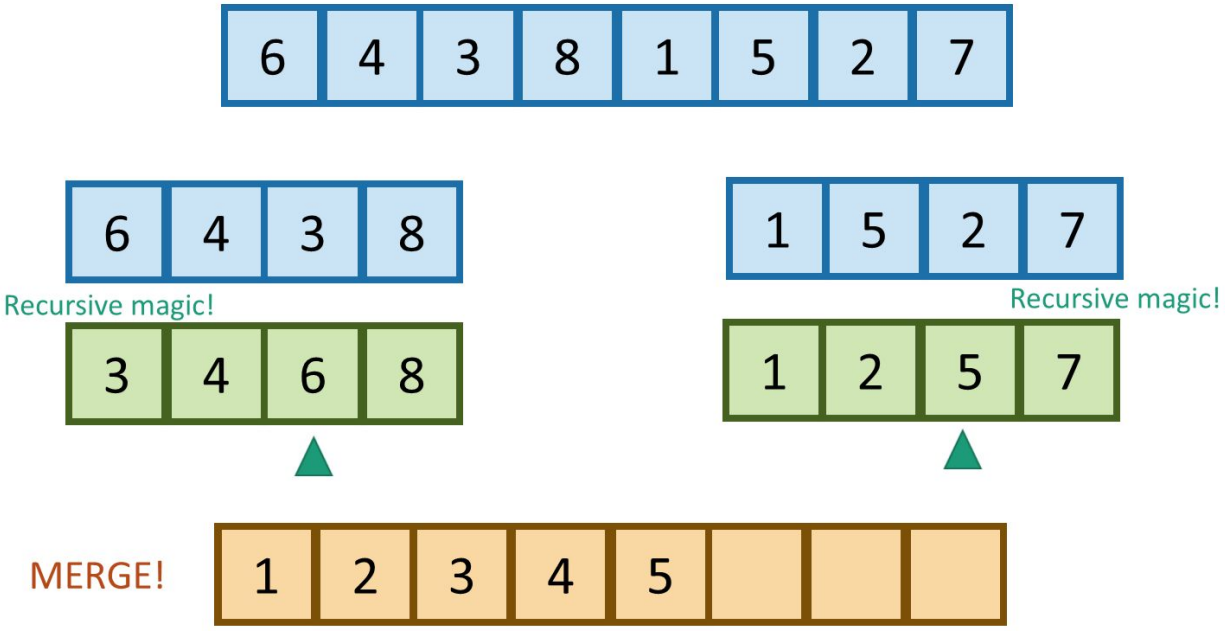
MergeSort



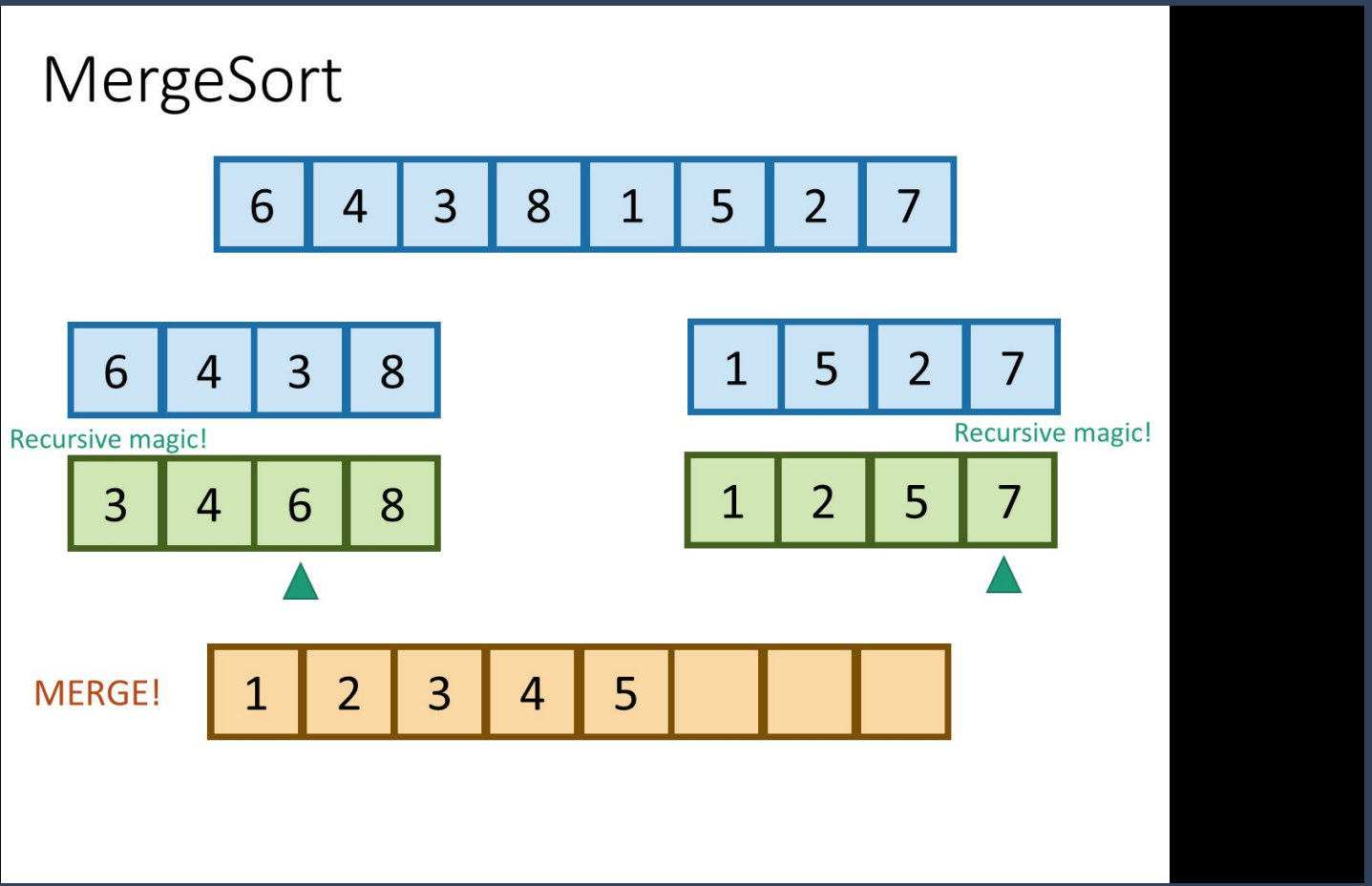
MergeSort



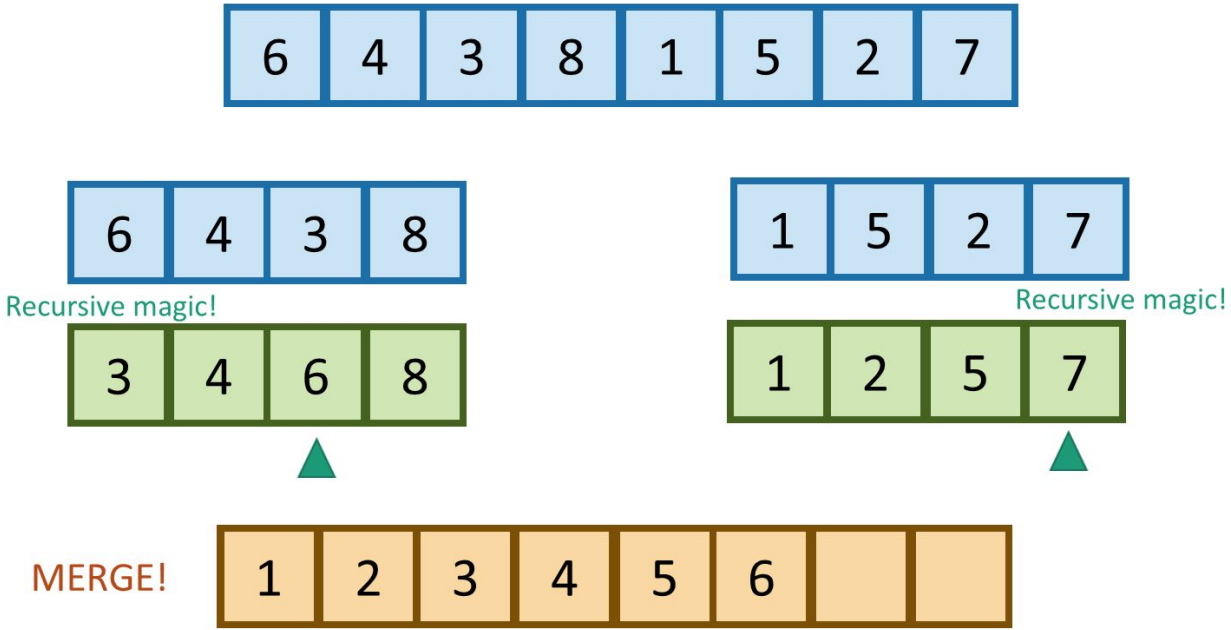
MergeSort



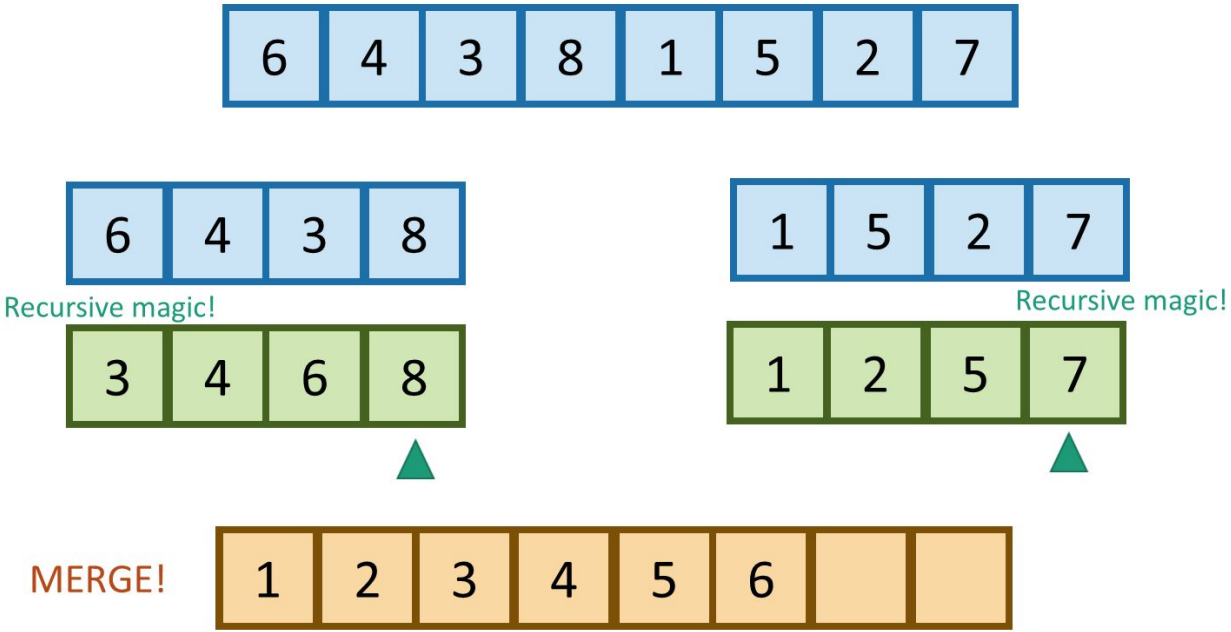
MergeSort



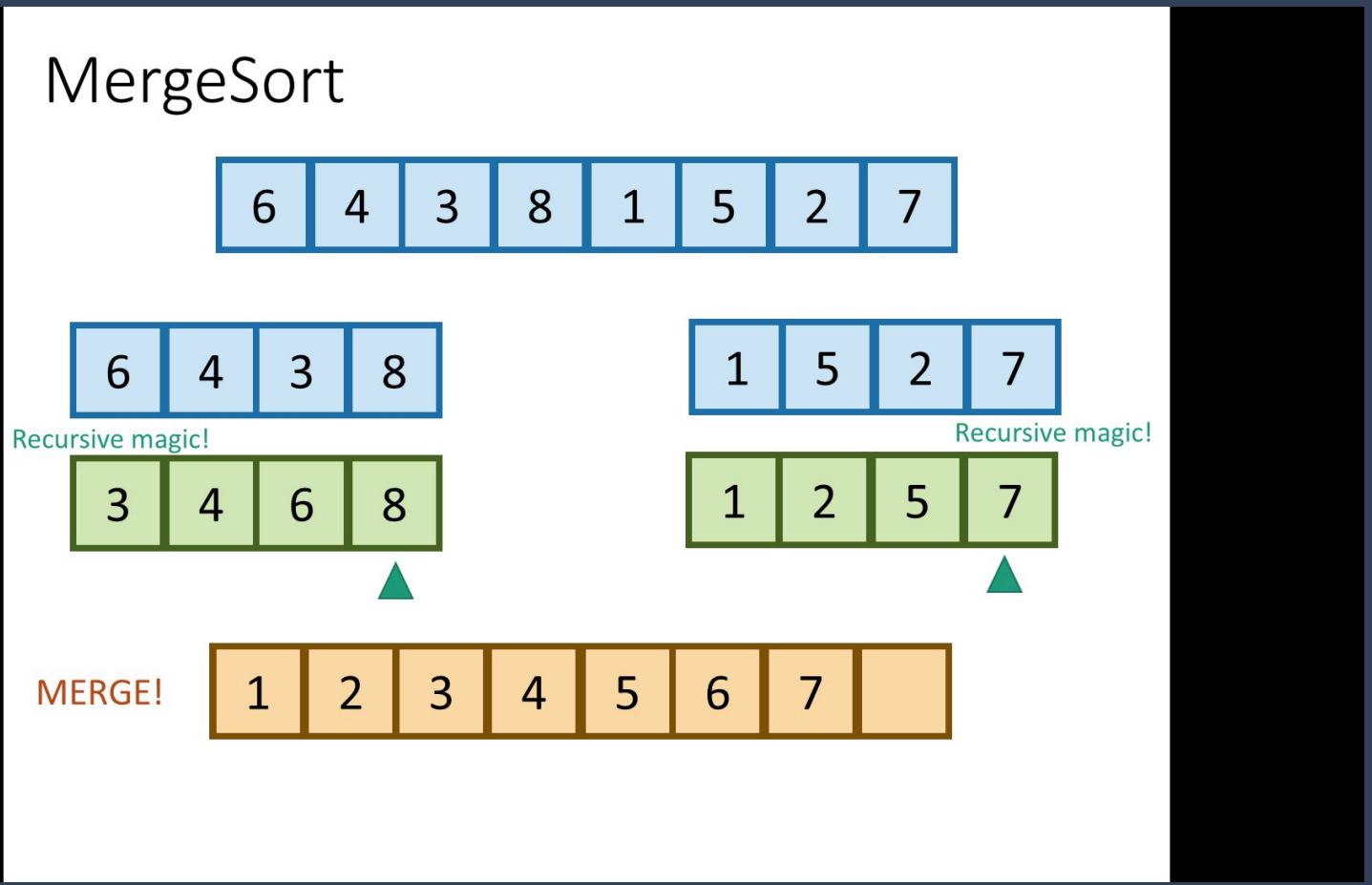
MergeSort



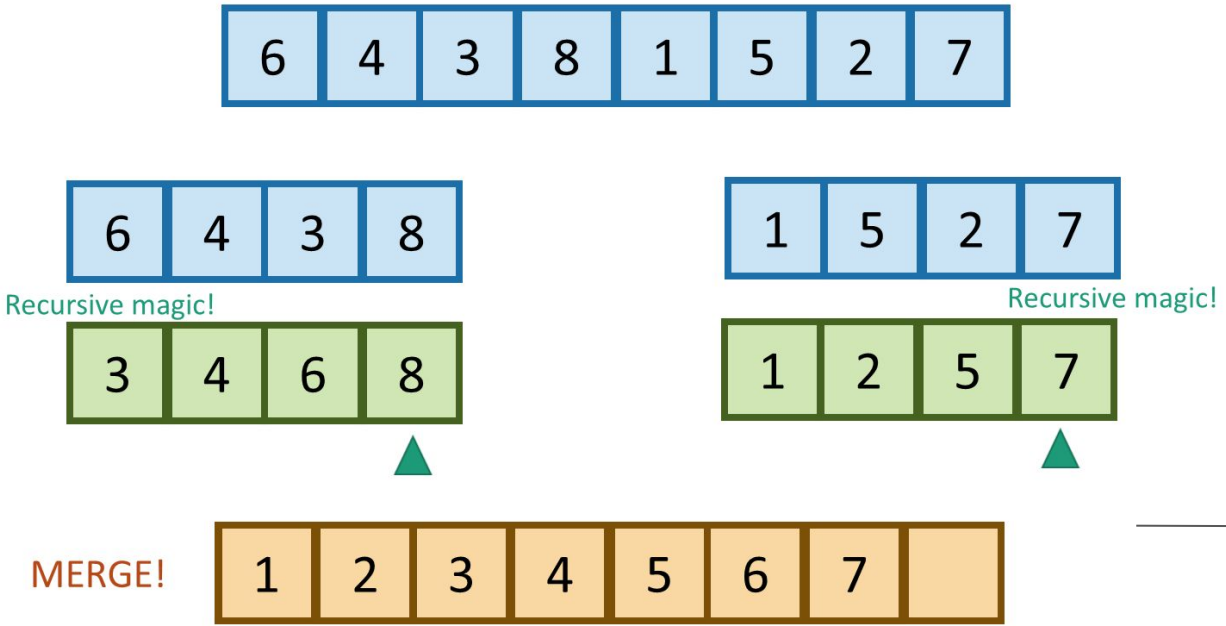
MergeSort



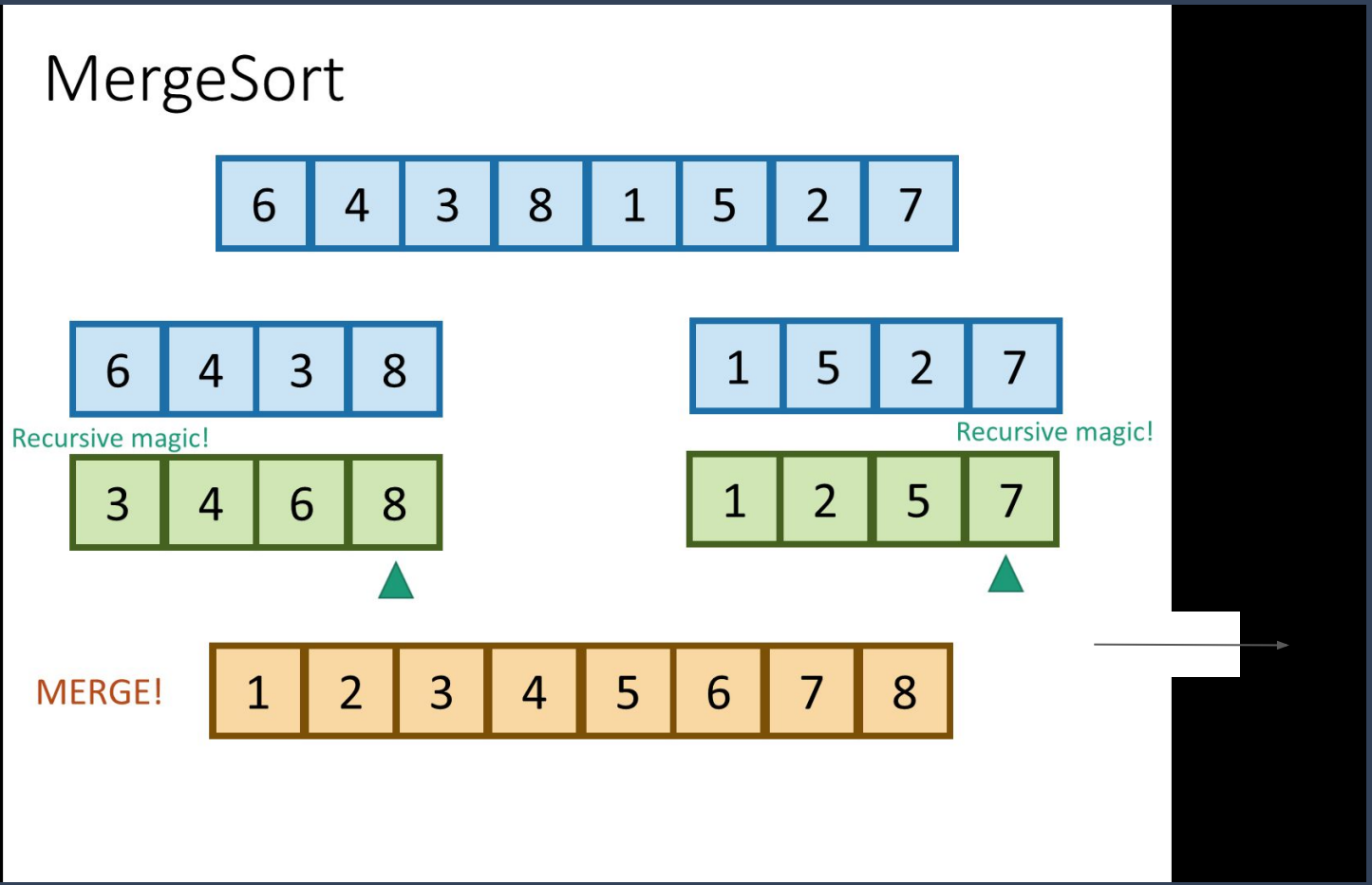
MergeSort



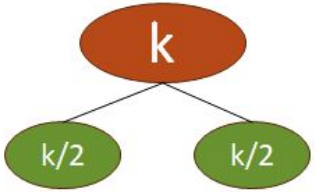
MergeSort

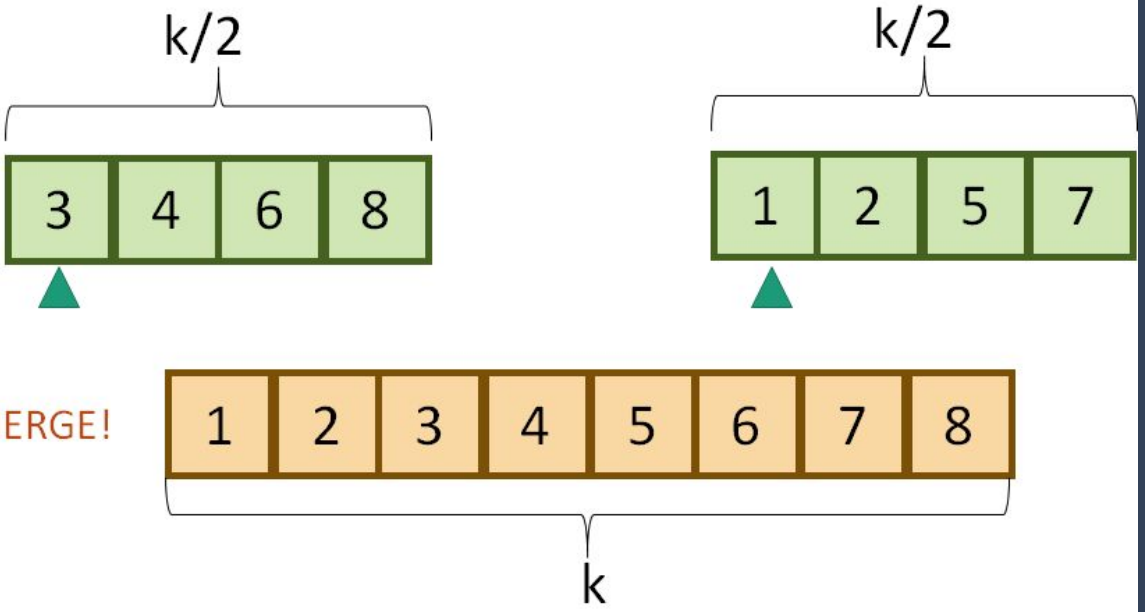


MergeSort



How long does it take to MERGE?





MERGESORT(A):

- n = length(A)
- **if** n ≤ 1:
  - **return** A

If A has length 1,
It is already sorted!

Partitioning based on position

- L = MERGESORT(A[1 : n/2])

Sort the left half

- R = MERGESORT(A[n/2+1 : n])

Sort the right half

- **return** MERGE(L,R)

Merge the two halves

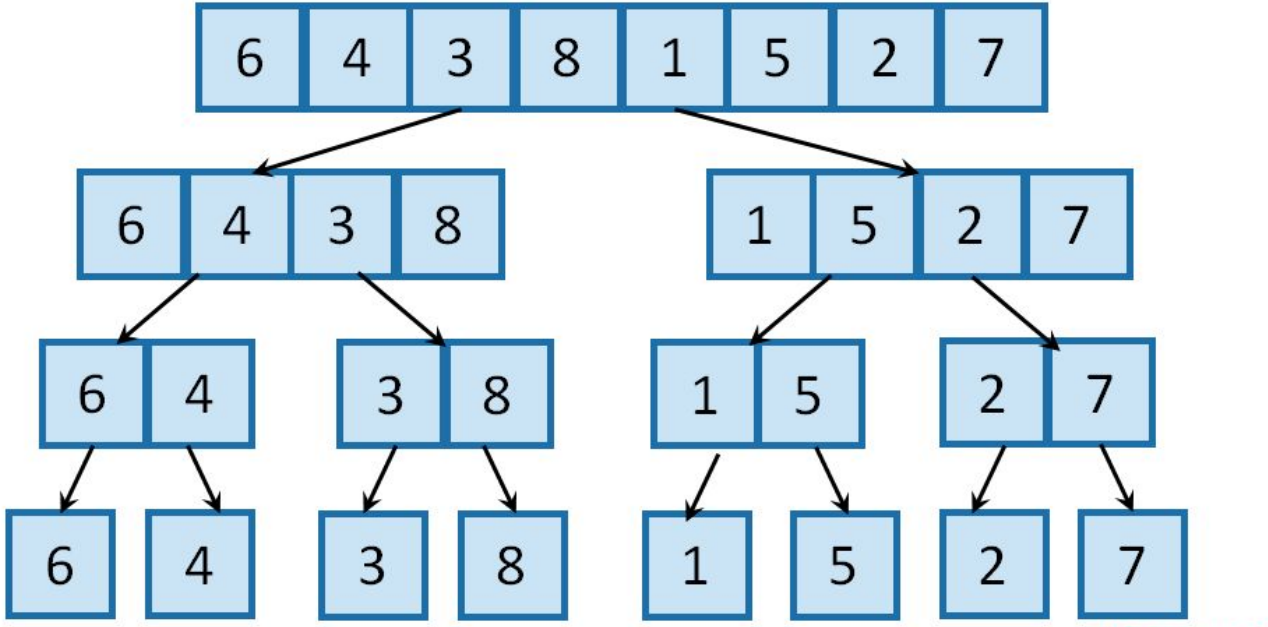
```
def msort(l, start, end):
    if start >= end:
        return
    mid = (start + end)/2
    msort(l, start, mid)
    msort(l, mid+1, end)
    i = start
    j = mid+1
    mlist = []
    while i <= mid and j <= end:
        if l[i] <= l[j]:
            mlist.append(l[i])
            i += 1
        elif l[j] < l[i]:
            mlist.append(l[j])
            j += 1
    while i <= mid:
        mlist.append(l[i])
        i += 1
    while j <= end:
        mlist.append(l[j])
        j += 1
    l[start:end+1] = mlist
```

```
#Now merge the two sorted
subarrays l[start...mid]
and l[mid+1...end]
```

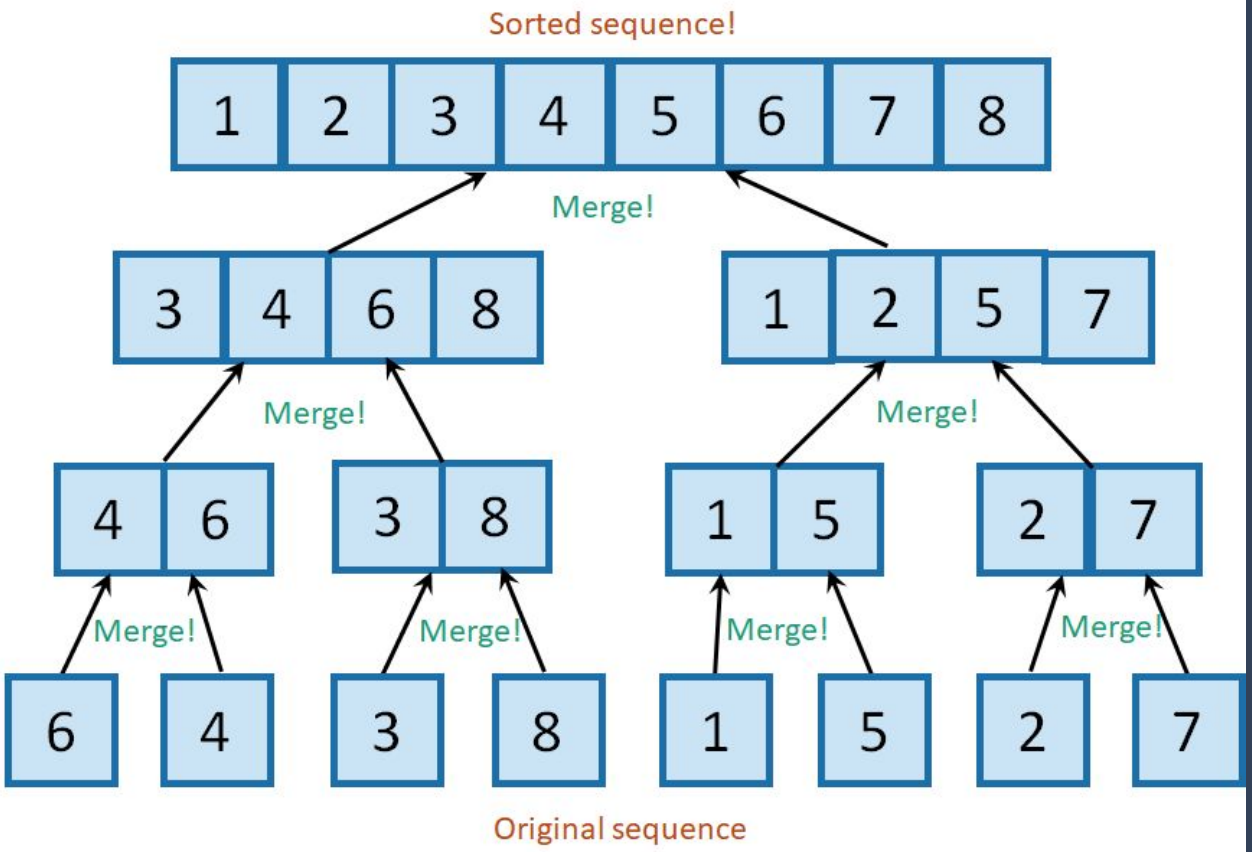
```
def mergesort(a):
    msort(a, 0, len(a)-1)
```

What actually happens?

First, recursively break up the array all the way down to the base cases



Schematic of recursive calls



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

Aside:

All logarithms in this course are base 2

Quick log refresher

- Def: log(n) is the number so that $2^{\log(n)} = n$.
- Intuition: log(n) is how many times you need to divide n by 2 in order to get down to 1.

32, 16, 8, 4, 2, 1 ⇒ log(32) = 5

Halve 5 times

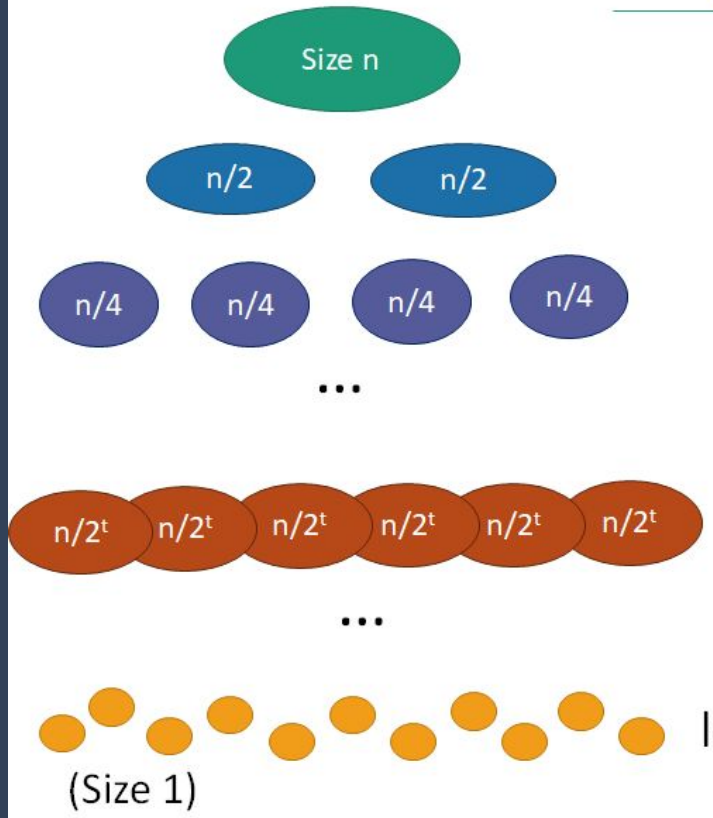
64, 32, 16, 8, 4, 2, 1 ⇒ log(64) = 6

Halve 6 times

log(128) = 7
log(256) = 8
log(512) = 9
....
log(**# particles in the universe**) < 280

- log(n) grows very slowly!

Recursion tree



Level	# problems	Size of each problem	Amount of work at this level (just MERGEing)
0	1	n	cn
1	2	n/2	cn
2	4	n/4	cn
...	...		
t	2^t	$n/2^t$	cn
...	...		
log(n)	n	1	cn

Time complexity

Merge Sort: O(n log n)

Worst-case? Best-case? Average-case?

Recall:

Insertion Sort: Worst-case and average-case, $O(n^2)$, best-case O(n)
Selection Sort and Bubble Sort: $O(n^2)$

TimSort

Combines merge-sort and insertion-sort for wo
best-case O(n) time.

Timsort has been Python's standard sorting algorithm

since version 2.3

Tim Peters

How important is the distinction between n log n and n^2?

US population = 320 million

Suppose we had to sort census data based on names.

Assume that a CPU can perform 100 million basic operations per second (10^8).

Executing n log n operations would take ~90 seconds or 1.5 minutes.

Executing n^2 operations would take 32.5 years!

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

Merge sort requires extra memory

An algorithm is said to be **in place** if it does not require extra memory, except a constant amount of memory units.

Is merge sort in place?

Insertion sort? Selection sort? Bubble sort?

Recall:

Design strategy 3: Divide-and-conquer

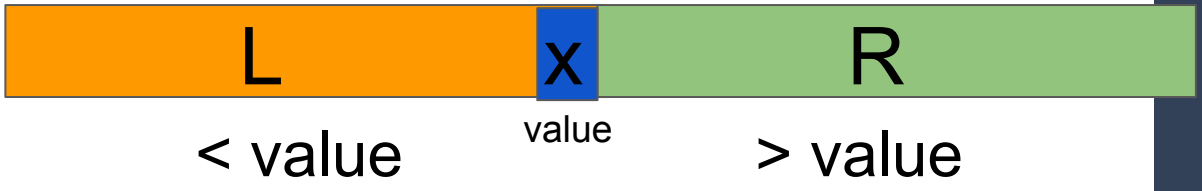
- **Divide** the problem into several smaller instances of the problem (most often two), generally of same size
- **Solve** the smaller instances (typically using recursion)
- **Combine** the solutions to the smaller instances to get the solution to the original problem.

Merge Sort

- **Divide** the array into two subarrays, based on **position** (trivial).
- **Solve** the problem on the smaller subarrays by recursively sorting them.
- **Combine** the two sorted subarrays by merging them together (took “cn” work, and extra space)

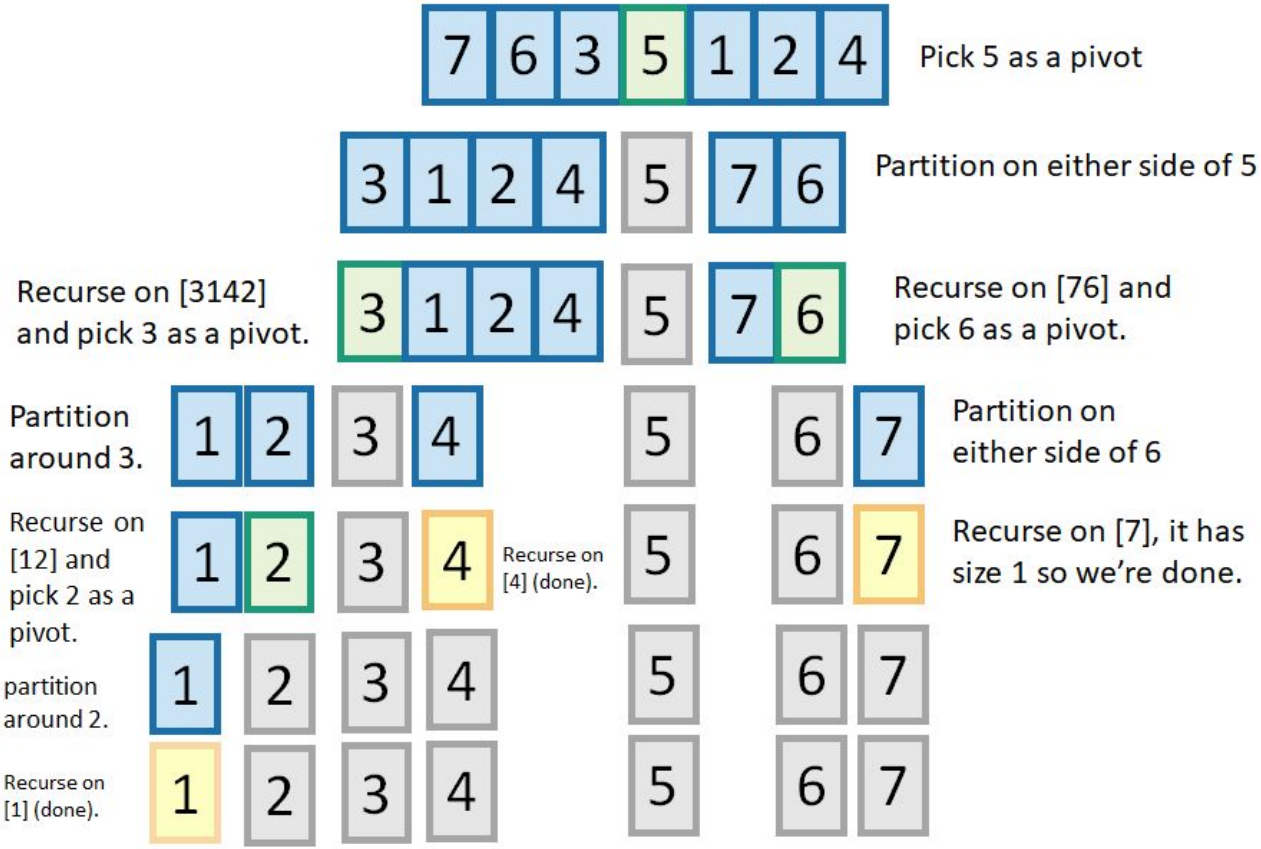
- **Divide** the array into two subarrays, based on **value**. (this will take some work)
- **Solve** the problem on the smaller subarrays by recursively sorting them.
- **Combine** the two sorted subarrays by merging them together (trivial)

- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick some x = A[i] at random. Call this the **pivot**.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Replace A with [L, x, R] (that is, rearrange A in this order)
  - QuickSort(L)
  - QuickSort(R)

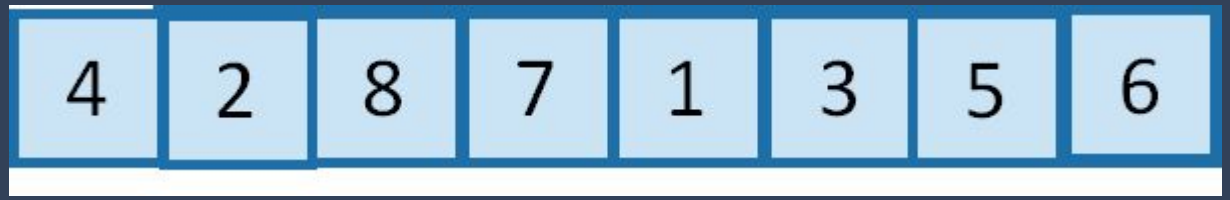


Note: x is in the right spot

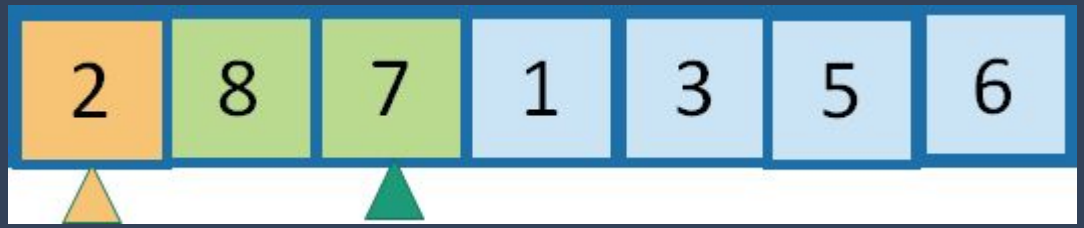
Example of recursive calls

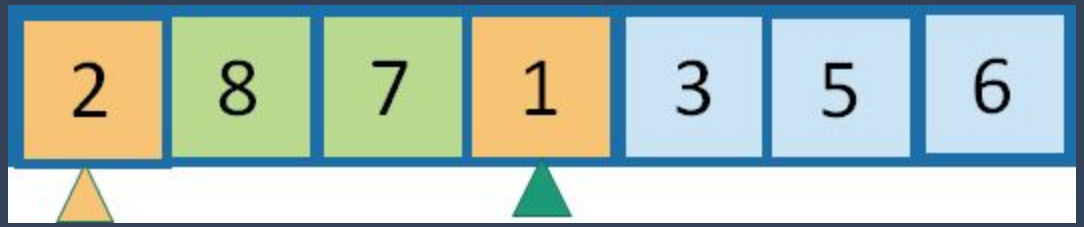


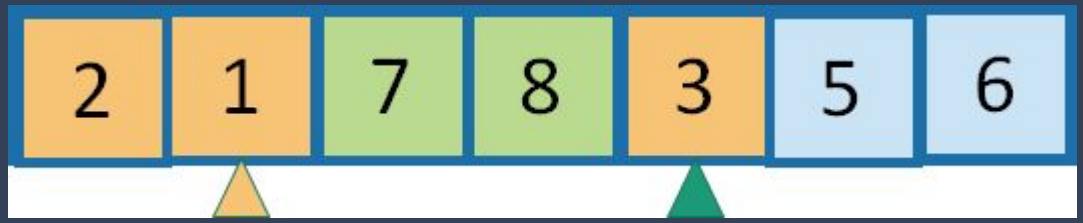
Can partitioning be done in-place?

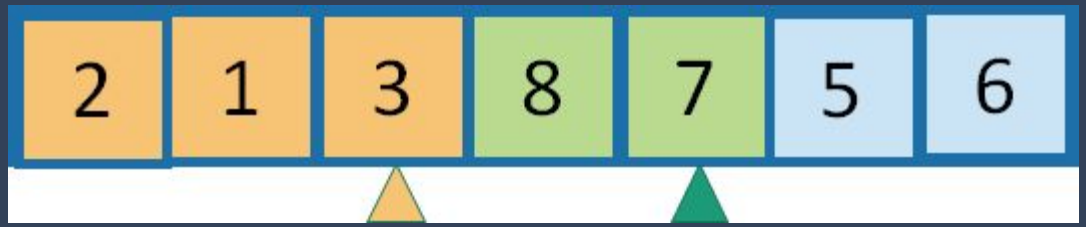


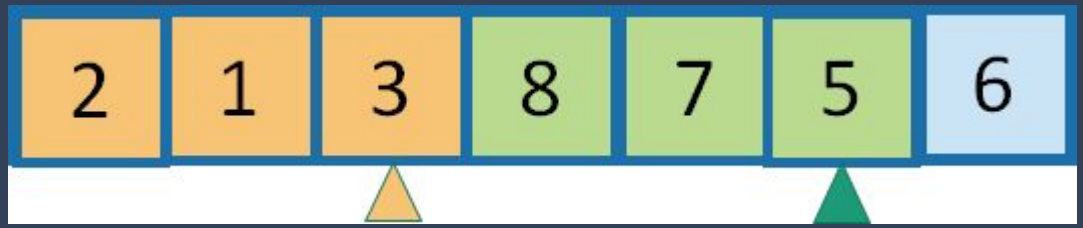
< value > value

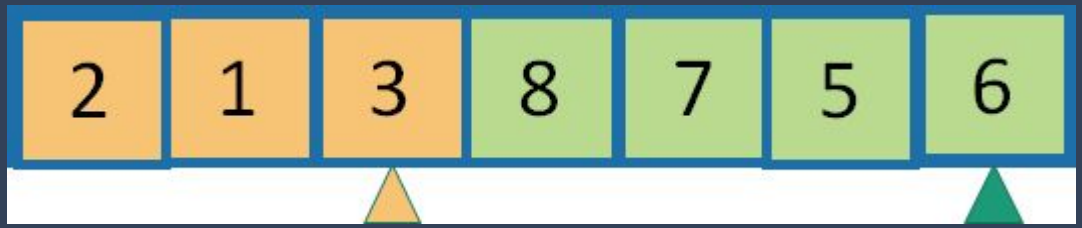


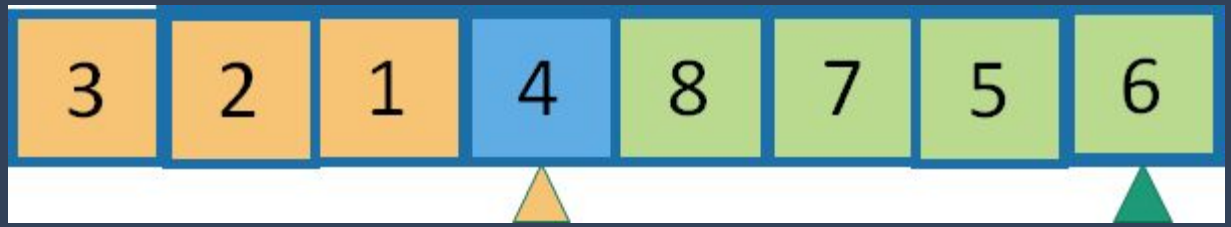








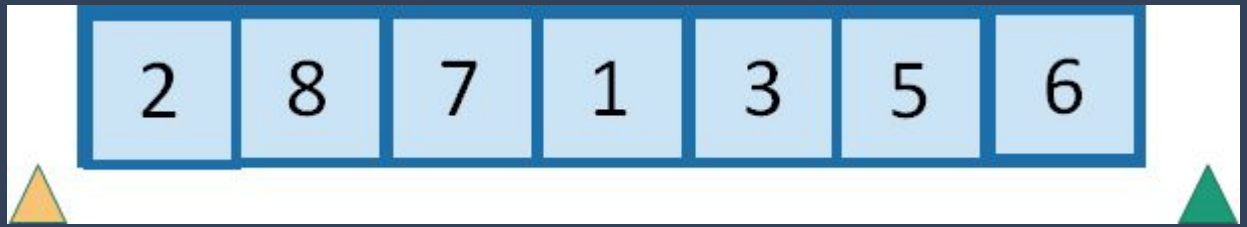


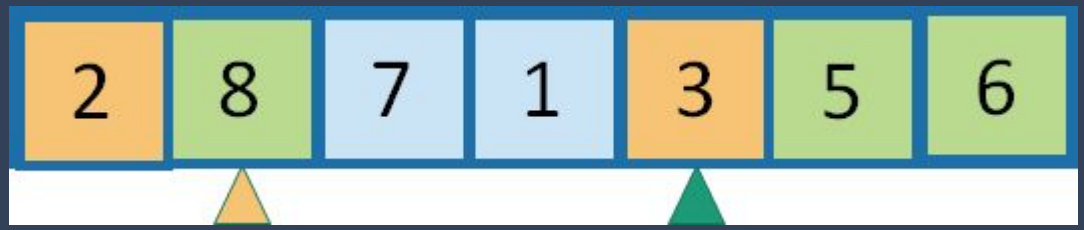


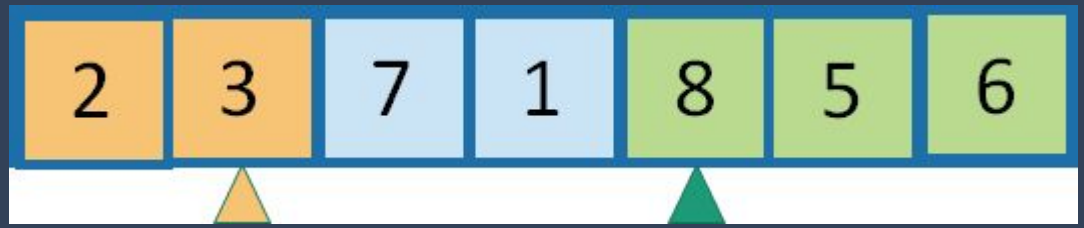
Lomuto’s partitioning: O(n) time, in place

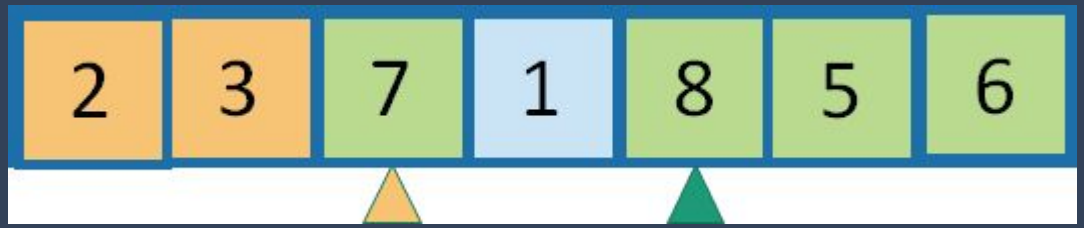
```
def qsort(l, start, end):
    if start >= end:
        return
    #Pick a random element as pivot
    randindex = random.randint(start,
end)
    (l[randindex],l[start]) =
(l[start],l[randindex])
    pivot = l[start]
    smaller = start
    bigger = start
    for bigger in range(start+1, end+1):
        if l[bigger] <= pivot:
            smaller += 1
            (l[smaller],l[bigger]) =
(l[bigger],l[smaller])
    #Place pivot in the right spot
    (l[start],l[smaller]) =
(l[smaller],l[start])
    qsort(l, start, smaller - 1)
    qsort(l, smaller + 1, end)
```

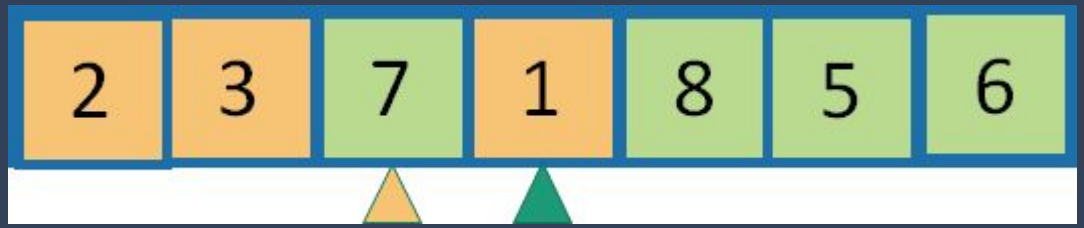
```
def quicksort(a):
   qsort(a,0,len(a)-1)
```



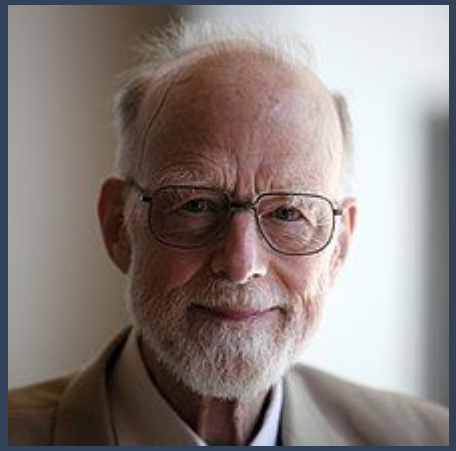








Hoare’s partitioning: O(n) time, in place



Tony Hoare (Quicksort, 1959)
ACM Turing Award 1980

John von Neumann (Mergesort, 1945)

Running time of Quicksort = ?

It depends on the sizes of the two partitions. If the pivot was the ith smallest element,

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

Best-case = ?

Worst-case = ?

Running time of Quicksort = ?

It depends on the sizes of the two partitions. If the pivot was the ith smallest element,

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

Best-case = Partition splits the numbers evenly, and we get a similar recursion as mergesort.

Worst-case = ?

Running time of Quicksort = ?

It depends on the sizes of the two partitions. If the pivot was the ith smallest element,

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

Best-case = Partition splits the numbers evenly, and we get a similar recursion as mergesort.

Worst-case = One partition empty, the other has all the elements. We get a similar recursion as insertion sort.

Running time of Quicksort = ?

It depends on the sizes of the two partitions. If the pivot was the ith smallest element,

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

Best-case = $O(n \log n)$

Worst-case = $O(n^2)$

What about average case?

Let T(n) = Expected number of operations needed to quicksort n numbers

During partitioning, we would pick the i^{th} smallest element as the pivot, with probability 1/n.

$$T(n) = 1/n \sum_{i=1}^{n} [T(i-1) + T(n-i) + cn]$$

This simplifies to

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

Running time of Quicksort = ?

It depends on the sizes of the two partitions. If the pivot was the ith smallest element,

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

Best-case & Average case = $O(n \log n)$

Worst-case = $O(n^2)$

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 $O(n^2)$ worst-case

But Quick Sort is also in-place, which Merge Sort is not. (although call stack space in Quicksort should also be noted)

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

If Randomized Quicksort is implemented well, it is more likely that your computer will be struck by a lightning bolt than the Quicksort running in $O(n^2)$ time.

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 $O(n^2)$ worst-case

Quick Sort runs faster in empirical analysis.

Quick Sort is also in-place, which Merge Sort is not.

But Merge Sort does have one advantage: **stability**

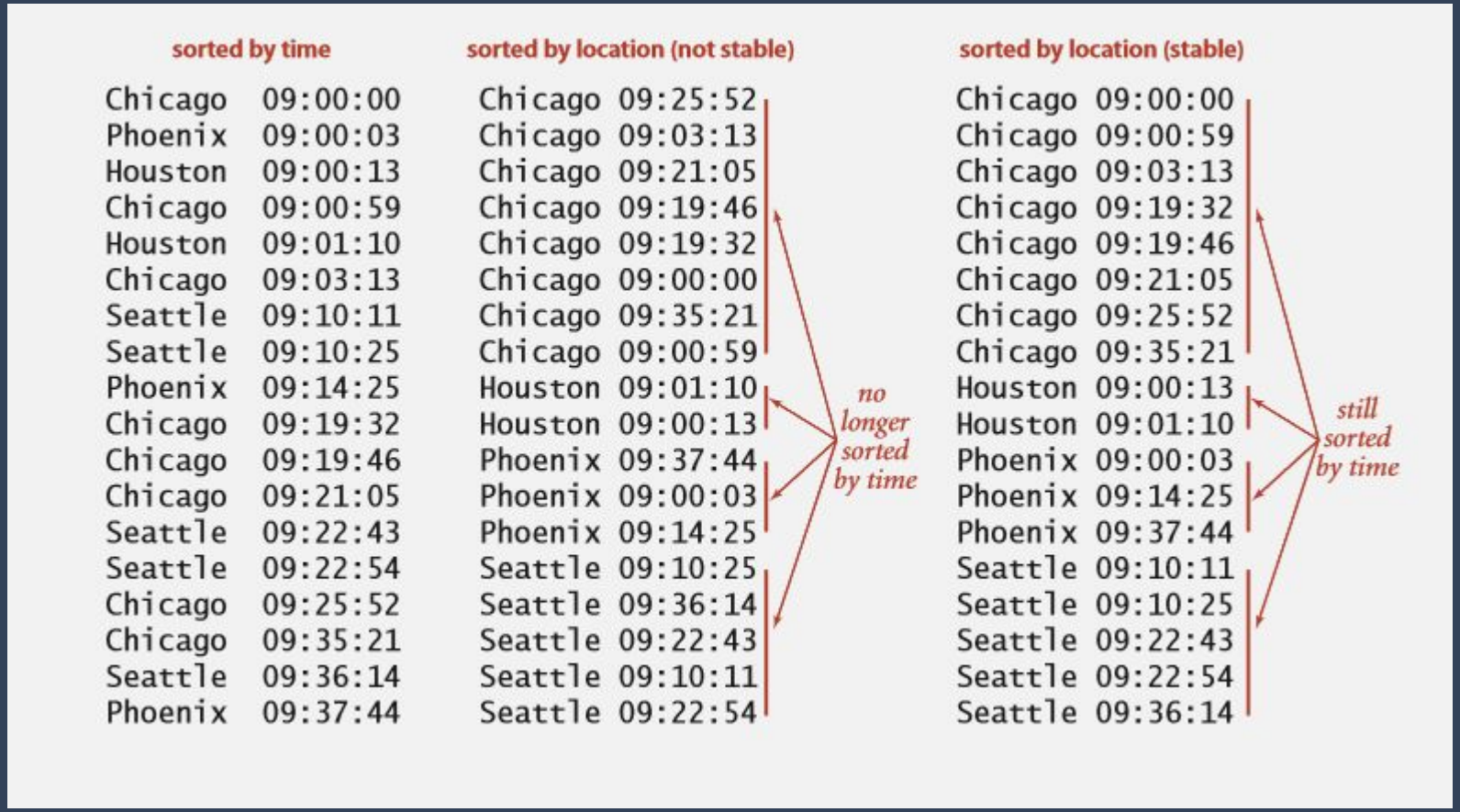
A typical application. First, sort by name; then sort by section.

```
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



A stable sort preserves the relative order of items with equal keys.

Selection Sort is not stable

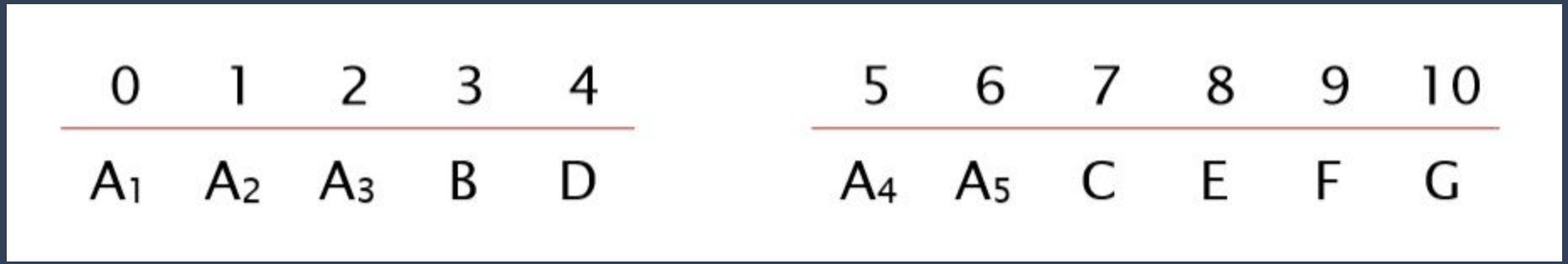
i	min	0	1	2
0	2	B_1	B_2	A
1	1	A	B_2	B_1
2	2	A	B_2	B_1
		A	B_2	B_1

Insertion Sort is stable

What about bubble sort?

i	j	0	1	2	3	4
0	0	B_1	A_1	A_2	A_3	B_2
1	0	A_1	B_1	A_2	A_3	B_2
2	1	A_1	A_2	B_1	A_3	B_2
3	2	A_1	A_2	A_3	B_1	B_2
4	4	A_1	A_2	A_3	B_1	B_2
		A_1	A_2	A_3	B_1	B_2

Merge Sort is stable. Why?



And what about Quick Sort?

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 $O(n^2)$ worst-case

Quick Sort runs faster in empirical analysis.

Quick Sort is in-place, Merge Sort is not.

Merge Sort is stable, Quick Sort is not.

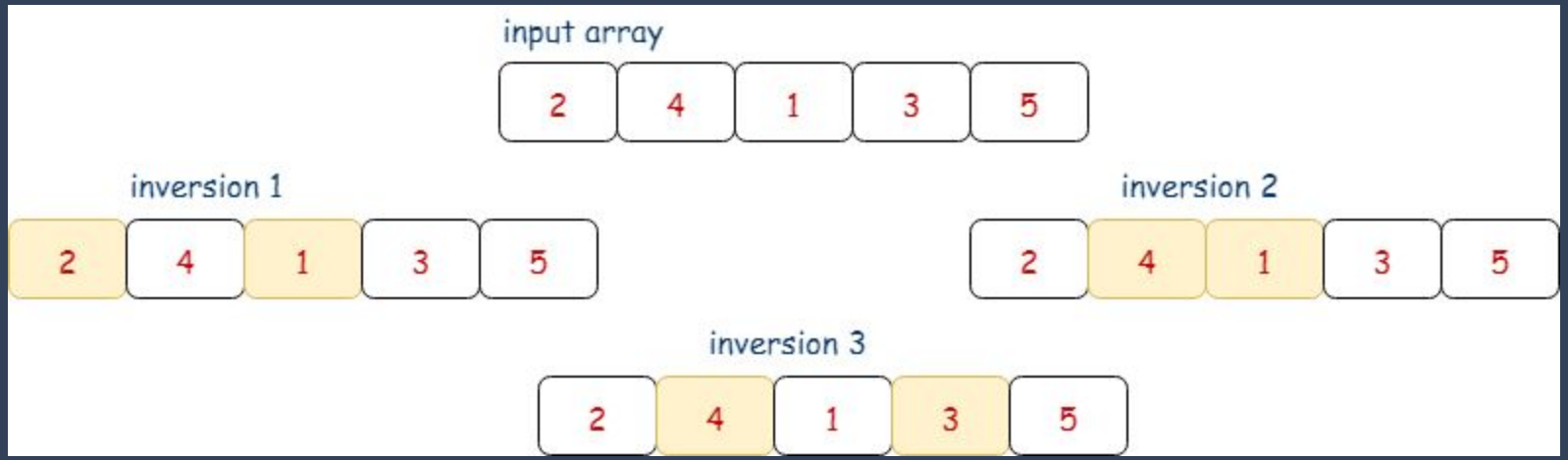
Java uses Quicksort for primitive types, and Merge Sort (Timsort) for Objects.

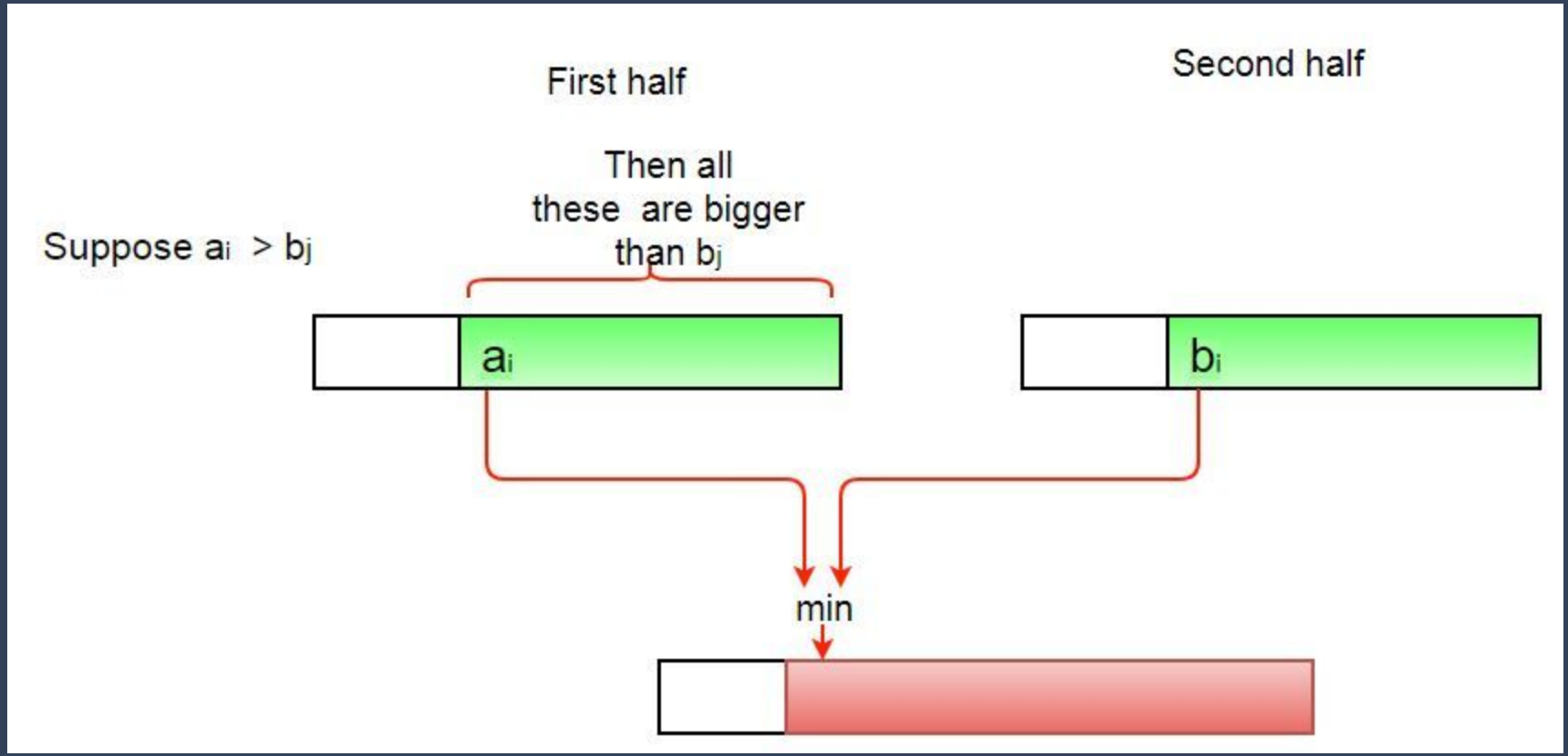
Since Merge Sort and Insertion Sort are both stable, Timsort is stable.

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

C++: sort uses quicksort, and stable_sort uses mergesort.

Count the number of inversions in an integer array

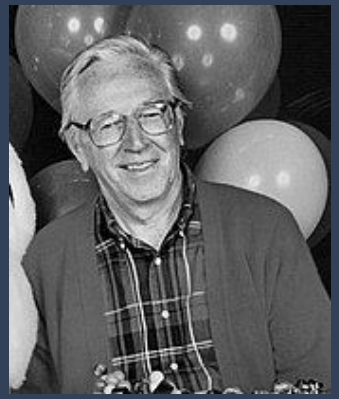




Runners numbered from 0 to n-1 race on a straight one-way road to a common finish line. The runners have different (constant) speeds and start at different distances from the finish line. Specifically, runner i has speed s[i] and begins at distance d[i] from the finish line. Each runner stops at the finish line, and the race ends when all the runners have reached the finish line. How many times does one runner pass another?

Design Strategy 4: Transform-and-Conquer

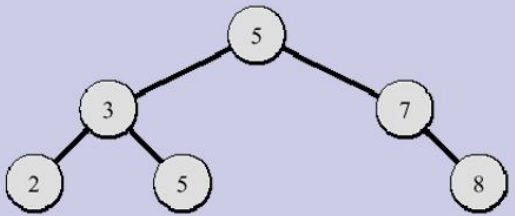
1. Transform the problem to a different representation (representation change)
2. Solve the problem.

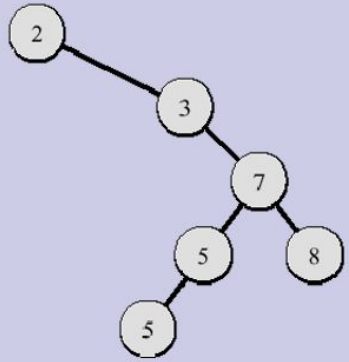


“That’s the secret to life... replace one worry with another.” (Charles M. Schulz, American cartoonist, the creator of *Peanuts*)

Binary Search Trees

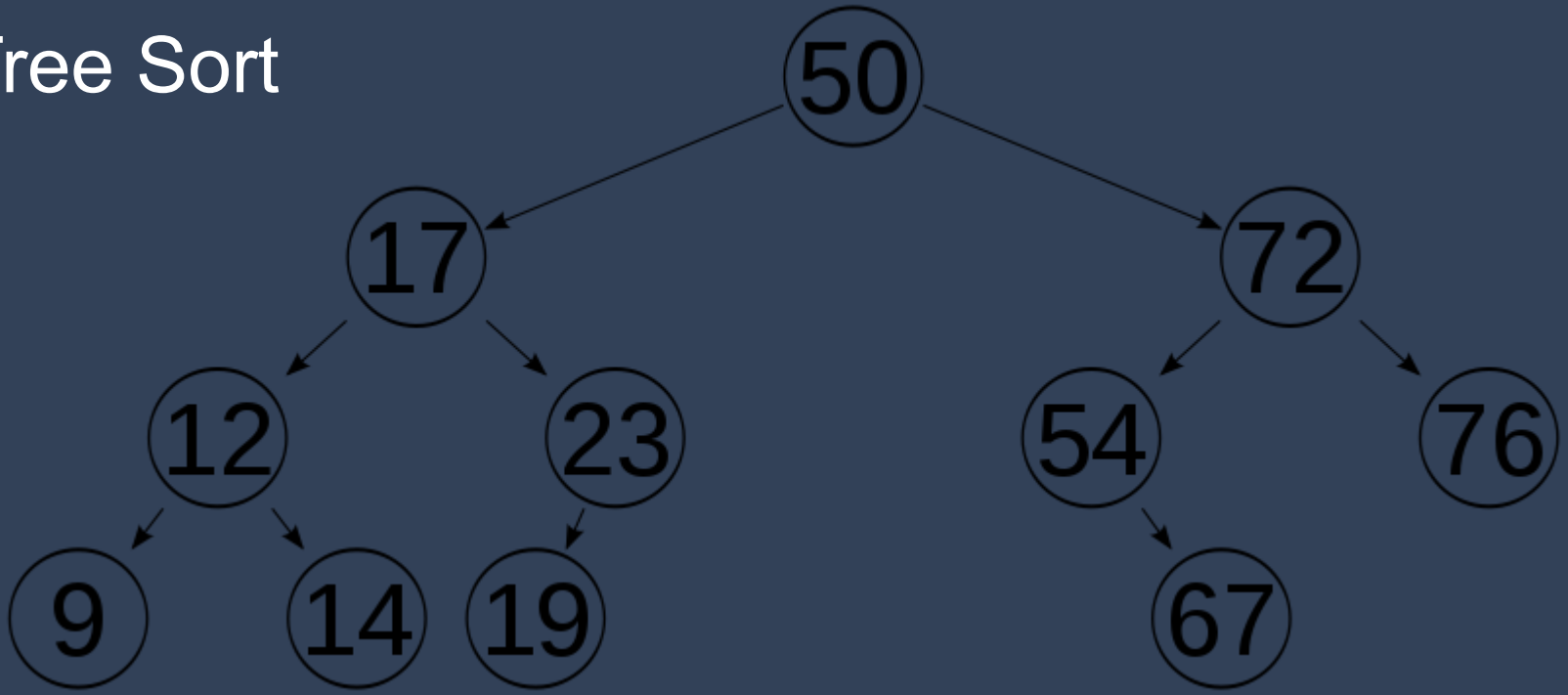
- A binary search tree is a binary tree T such that
  - each internal node stores an item (*k*,*e*) of a dictionary
  - keys stored at nodes in the **left subtree** of *v* are **less than or equal** to *k*
  - keys stored at nodes in the **right subtree** of *v* **are greater than or equal** to *k*
- Example sequence 2,3,5,5,7,8





For example, take the input data and change its representation to a Binary Search Tree.

Tree Sort



An inorder traversal of such a tree would give us the sorted sequence. (But the tree should be balanced)

Time Complexity Of Sorting

Sort n elements.

- n put operations => O(n log n) time.
- n remove max operations => O(n log n) time.
- total time is O(n log n).

Back to brute force...

Locate the smallest item and put it in the first place.

Then select the next smallest item and put it in the second place. And so on...

Sorting by repeated selection: "**Selection Sort**"

Final output ordering generated one by one in sequence.

Locating the smallest item took time O(n) each time… what if we could store the items in a way that allows finding and extracting the min (or max) item to be O(log n)?

Requirement for a new Abstract Data Type (ADT)

A specification of (1) what the data is, and (2) the operations that will be performed on the data.

It describes the interface the *user* of the ADT will see, not how it would be *implemented* under the hood. There can be different under-the-hood implementations of the same ADT.

For example, a Stack ADT can be implemented either using a dynamic array (Vector) or a linked list. Similarly for a queue ADT. The ADT description only mentions the operations (push, pop) and their semantics (LIFO behavior for stack, and FIFO behavior for queue)

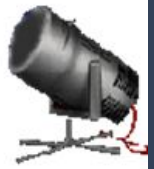
Priority Queue ADT

In our case, we need to take the elements that we want to sort, and insert them into an ADT, and be able to extract the minimum element from the ADT repeatedly.

So the two relevant operations we want are (1) insert, and (2) extract-minimum. The data itself can be assumed to be simple integers, or it could be any type of object for which there is a well-defined key field for comparison operation (that tells us whether object 1 is <, = or > object 2).

A priority queue ADT supports these operations.

Priority Queues



Two kinds of priority queues:

- Min priority queue.
- Max priority queue.

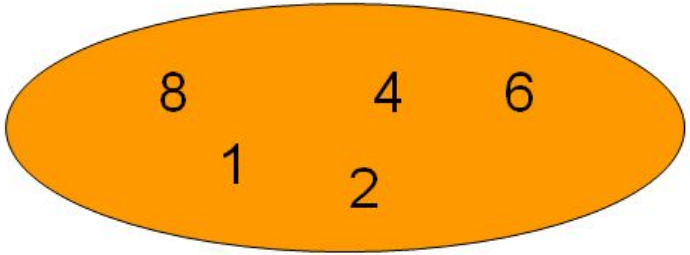
They just differ in whether higher priority means a smaller value or a larger value.

Sorting Example

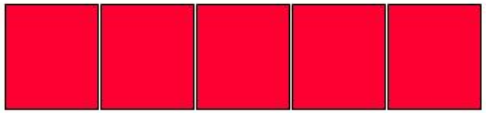
Sort five elements whose keys are 6, 8, 2, 4, 1 using a max priority queue.

- Put the five elements into a max priority queue.
- Do five remove max operations placing removed elements into the sorted array from right to left.

After Putting Into Max Priority Queue

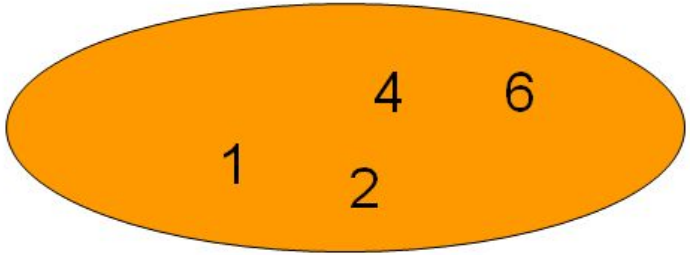


Max Priority Queue

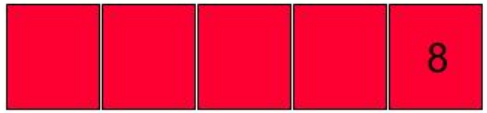


Sorted Array

After First Remove Max Operation

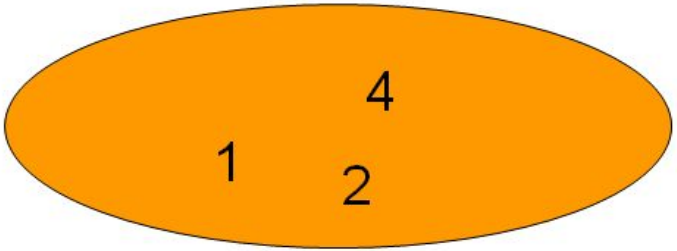


Max Priority Queue

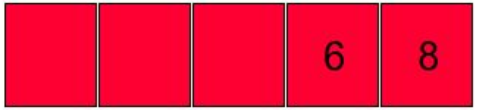


Sorted Array

After Second Remove Max Operation

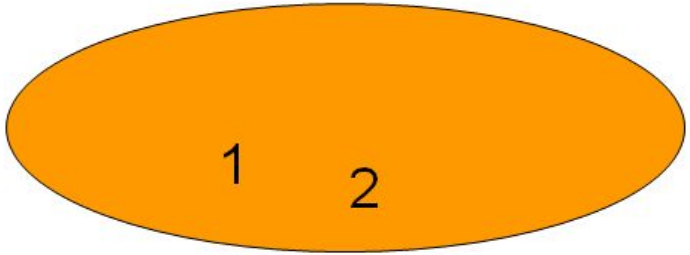


Max Priority Queue



Sorted Array

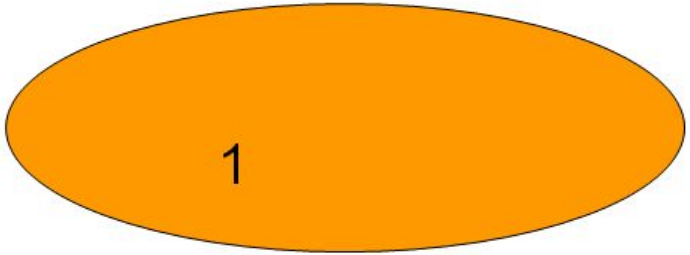
After Third Remove Max Operation



Max Priority Queue

Sorted Array

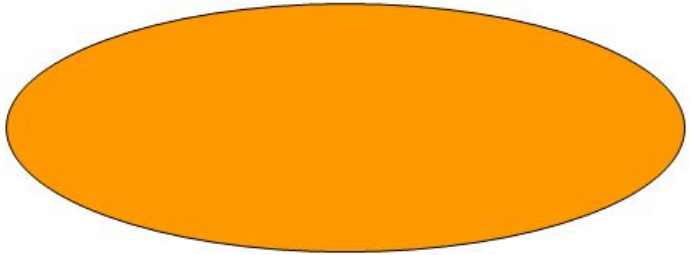
After Fourth Remove Max Operation



Max Priority Queue

Sorted Array

After Fifth Remove Max Operation



Max Priority Queue

Sorted Array

Priority Queue ADT isn't just to help in sorting

If we want to keep track of patients in an ER to determine in what order to serve them, we need a priority queue for them, with insert and extract-max operations.
(Perhaps a change-priority operation also?)

If want a printer to keep track of printing jobs with different priorities, or an operating system to keep track of processes with different priorities, we again need the same operations - insert, extract-max.

So even though we are covering priority queues in the context of the sorting problem, do keep in mind that they are used for many other problems.

Data structure

How the ADT is implemented under-the-hood.

The Priority Queue ADT can be implemented as a

1) Unsorted array
2) Sorted array
3) Unsorted list
4) Sorted list

How much time does insert and extract-max take for each option?

Implementation of Priority Queue ADT

We want an implementation that can do both insert and extract-max/min in O(log n) time. Only then can we get an overall O(n log n) sort.

Since the height of a complete binary tree with n elements is log n (why?), it makes sense to think of a structure like that under-the-hood.

And since the only element directly accessible for a binary tree is its root, it makes sense to somehow have the item with the highest priority at the root.

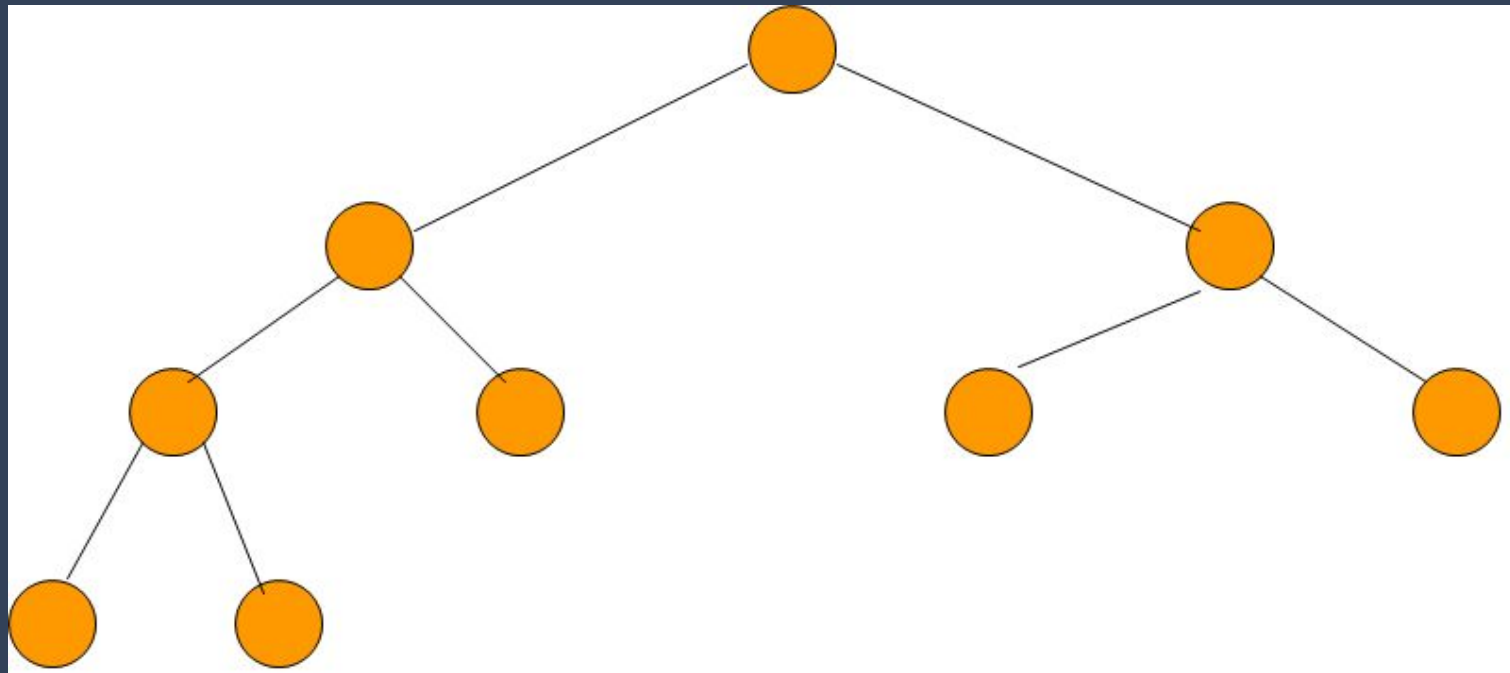
A **binary heap** is the most popular data structure that can do this.

Note that BSTs can also be used (as we briefly saw in tree sort) but binary heaps would be more efficient, as we can implement them using arrays instead of explicit tree nodes with pointers.

Binary Heap Definition

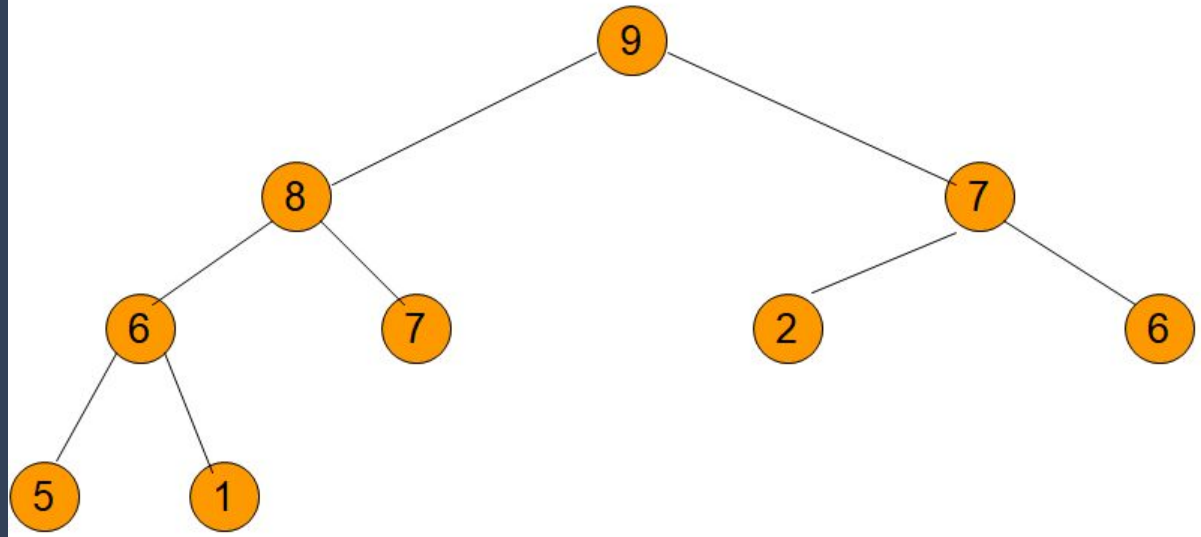
- Structural property: Complete binary tree
- Heap property

Priority of any node >= Priority of its children.



Complete binary tree with 9 nodes.

Max Heap With 9 Nodes

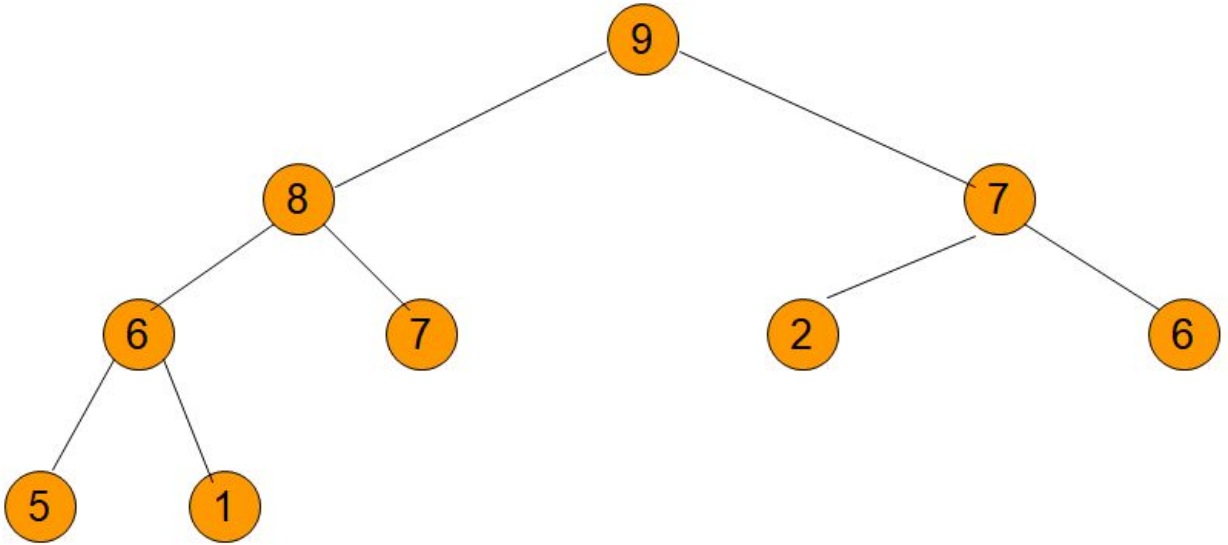


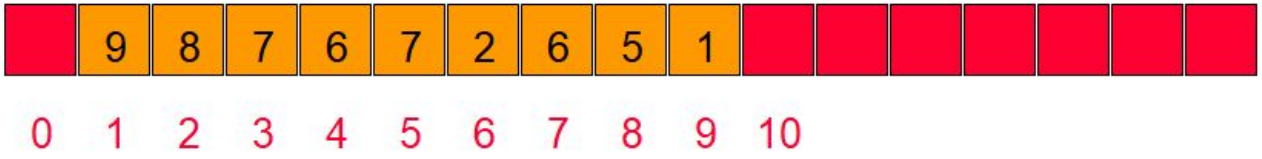
Complete binary tree with 9 nodes with max-heap property.

Max possible height of an n node heap?

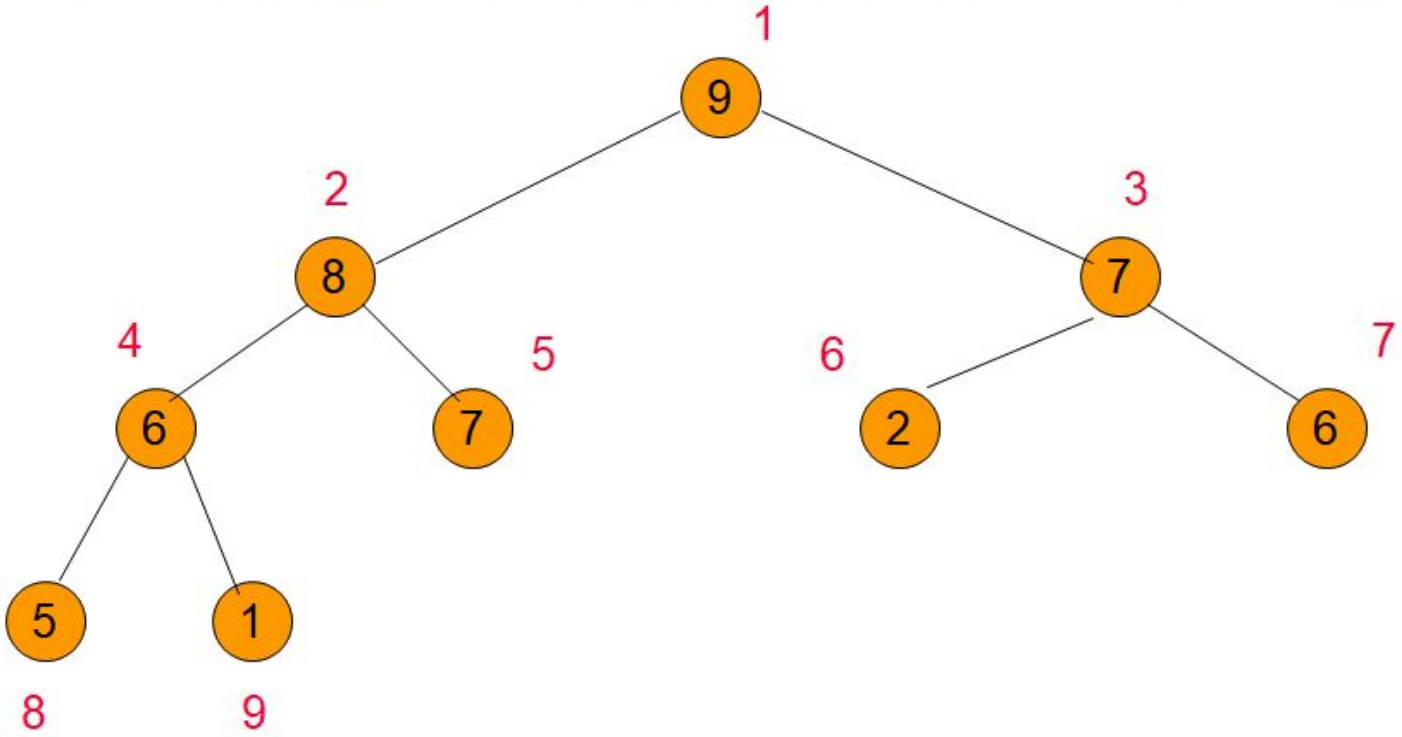
Max possible height of an n node heap?
O(log n)

A Heap Is Efficiently Represented As An Array

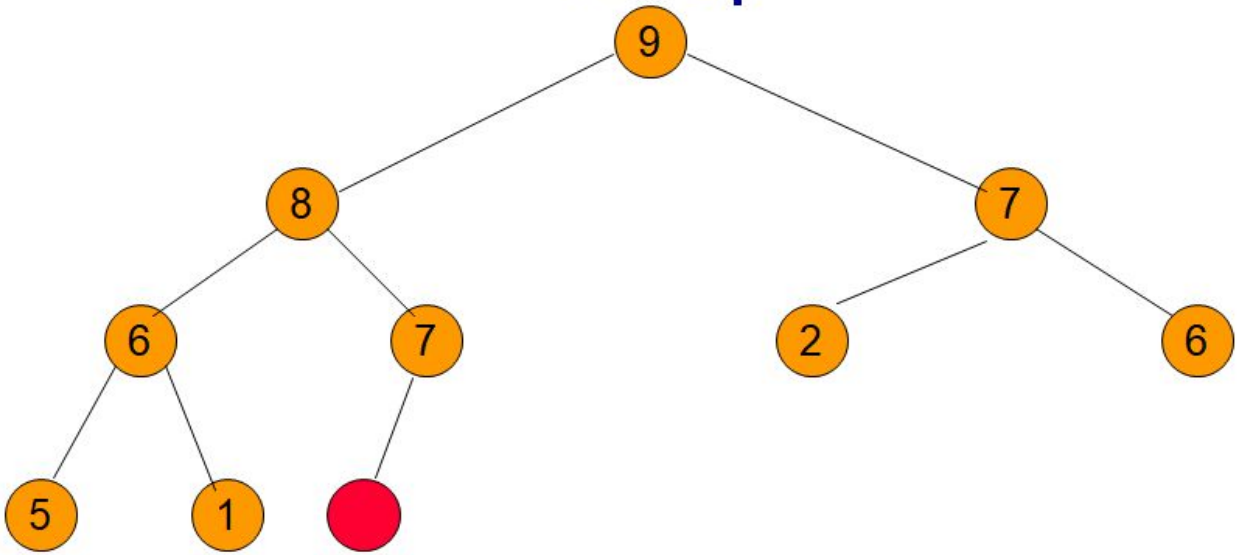




Moving Up And Down A Heap

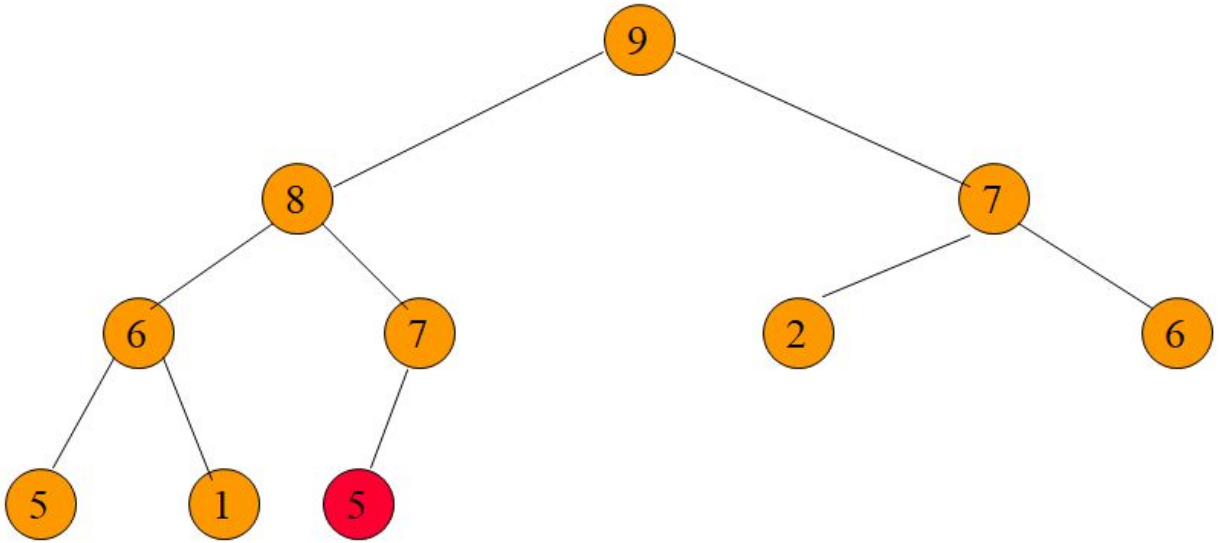


Putting An Element Into A Max Heap



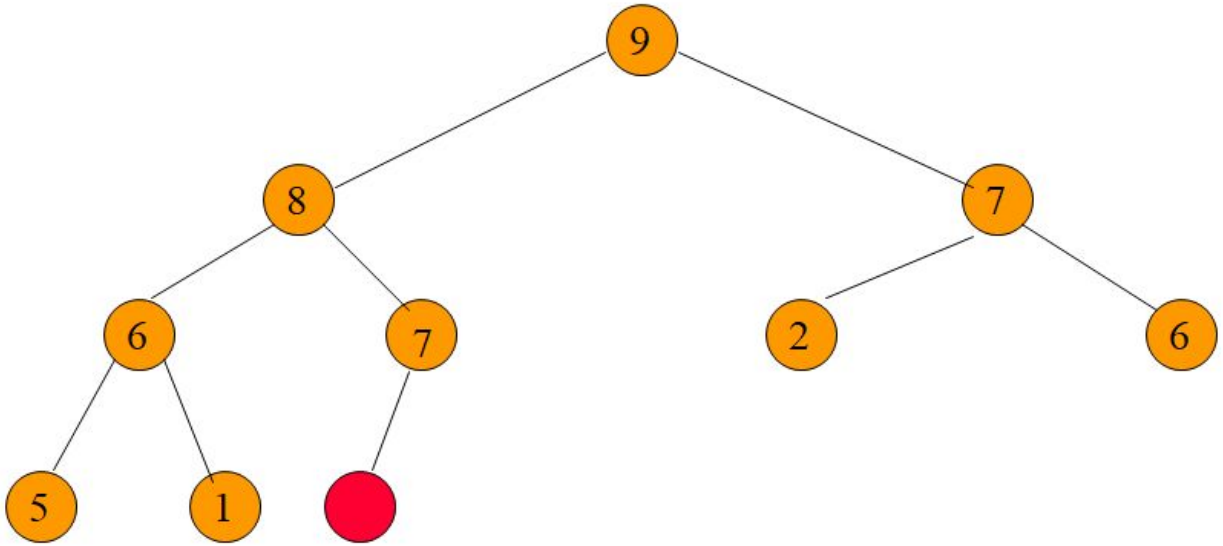
Complete binary tree with 10 nodes.

Putting An Element Into A Max Heap



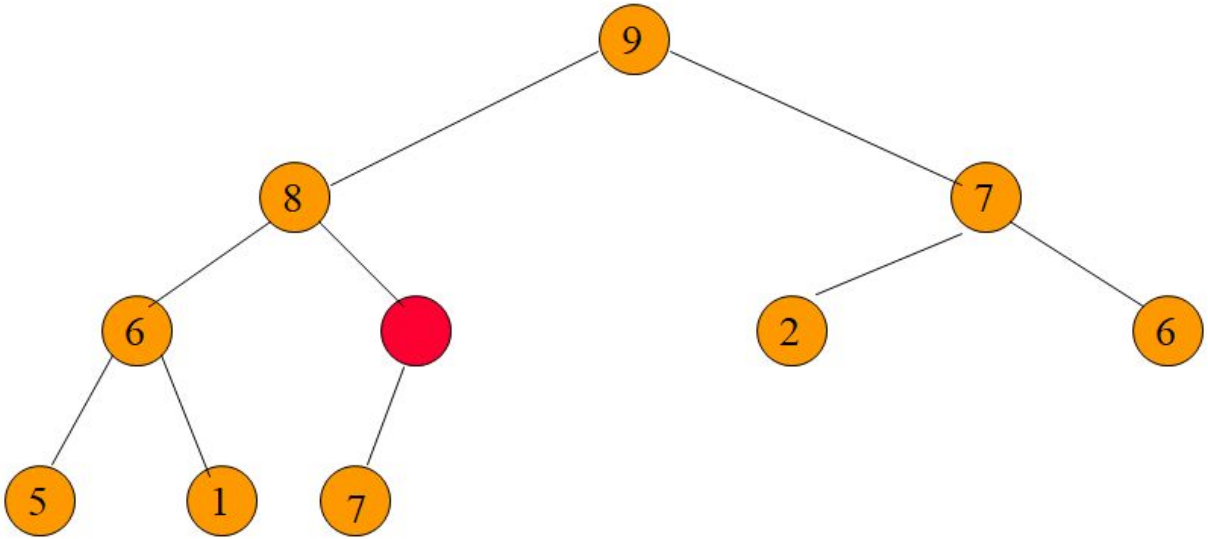
New element is 5.

Putting An Element Into A Max Heap



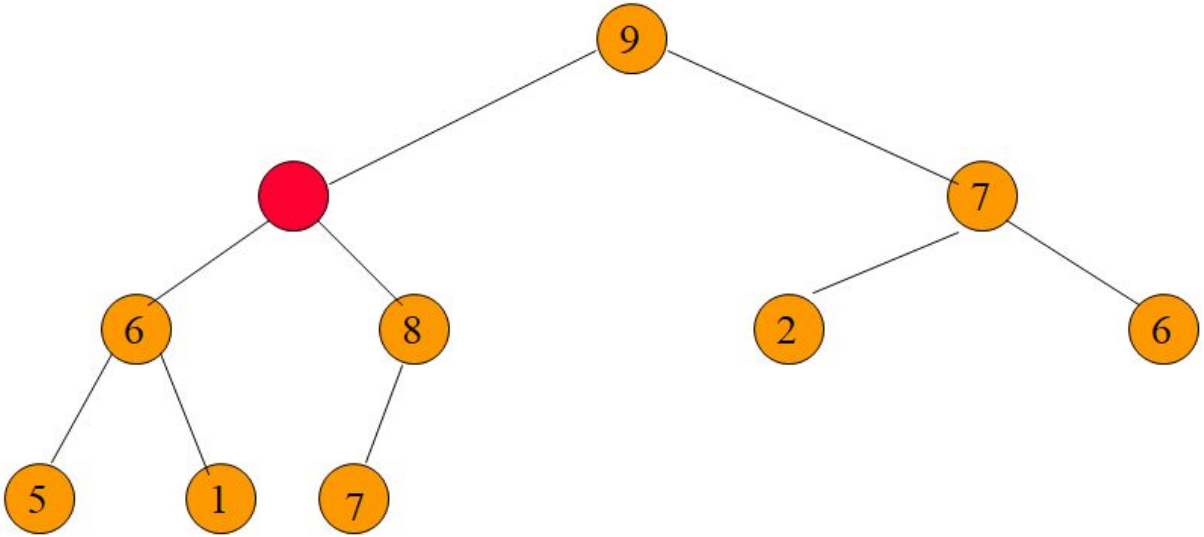
New element is 20.

Putting An Element Into A Max Heap



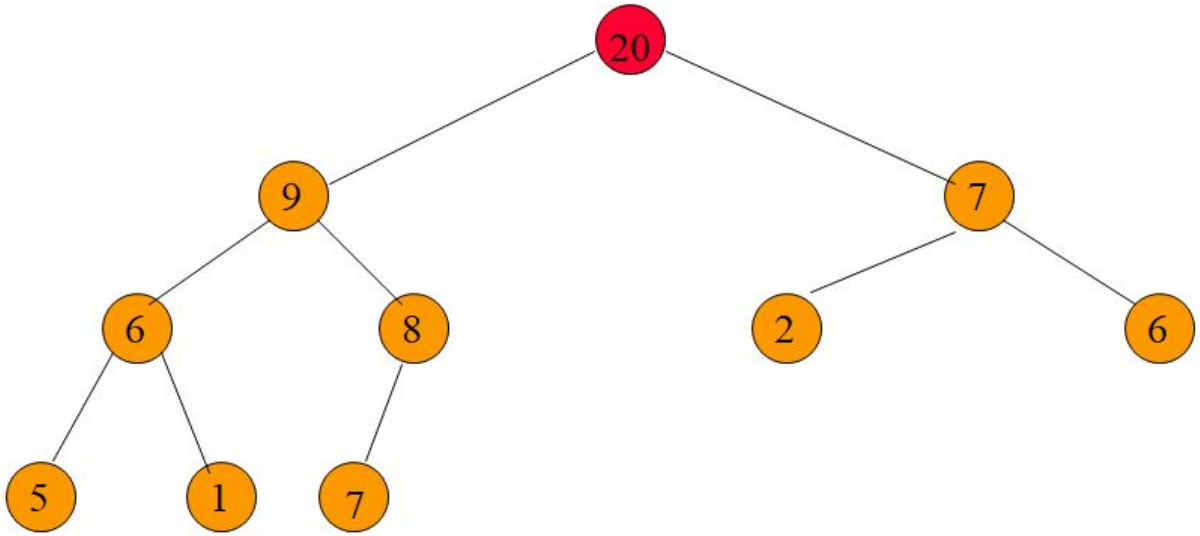
New element is 20.

Putting An Element Into A Max Heap



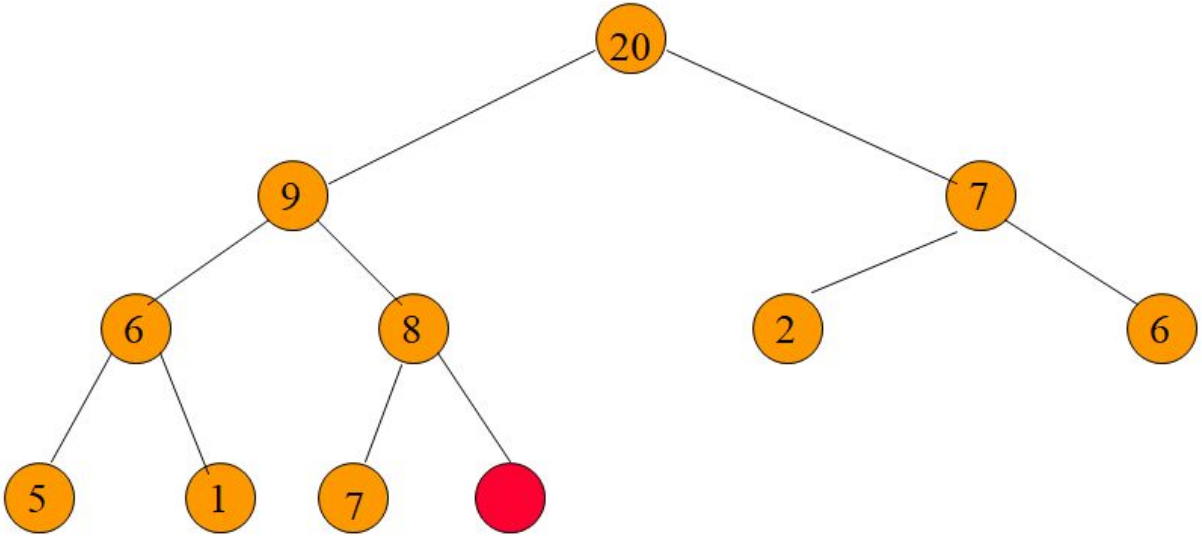
New element is 20.

Putting An Element Into A Max Heap



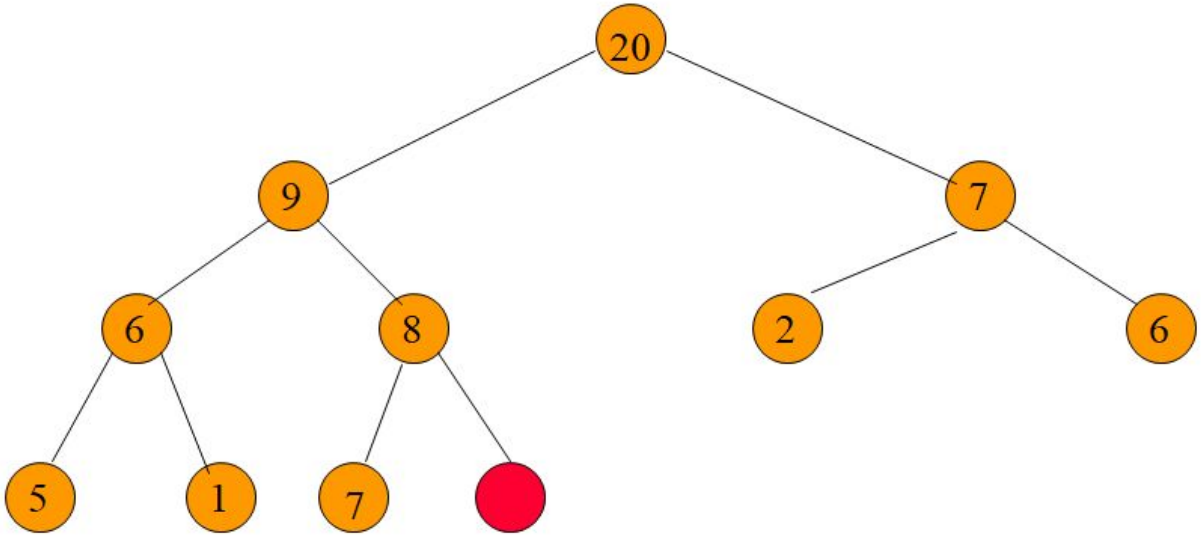
New element is 20.

Putting An Element Into A Max Heap



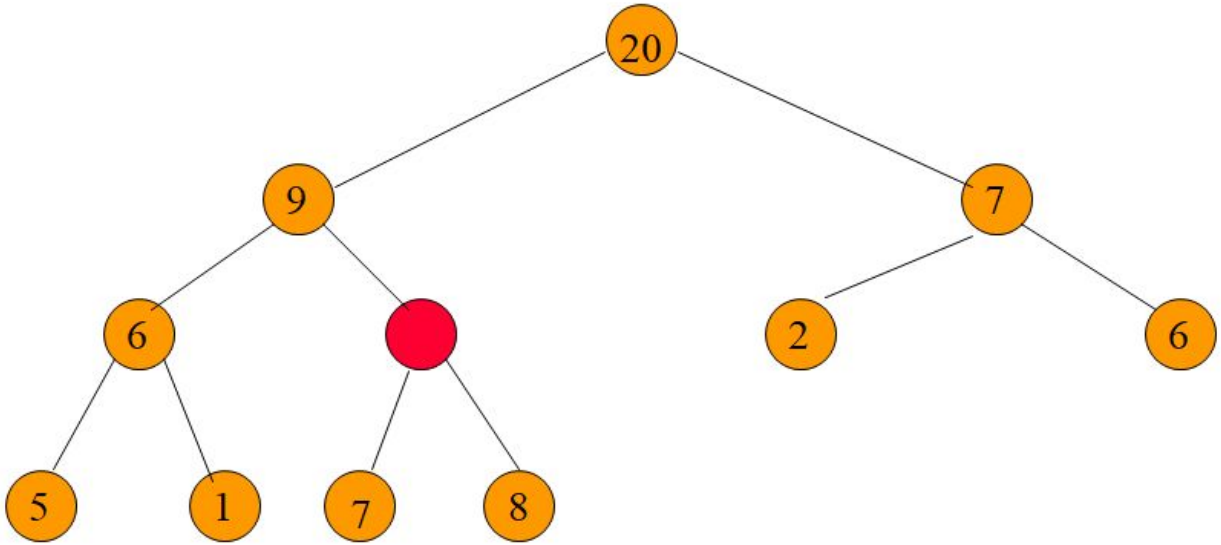
Complete binary tree with 11 nodes.

Putting An Element Into A Max Heap



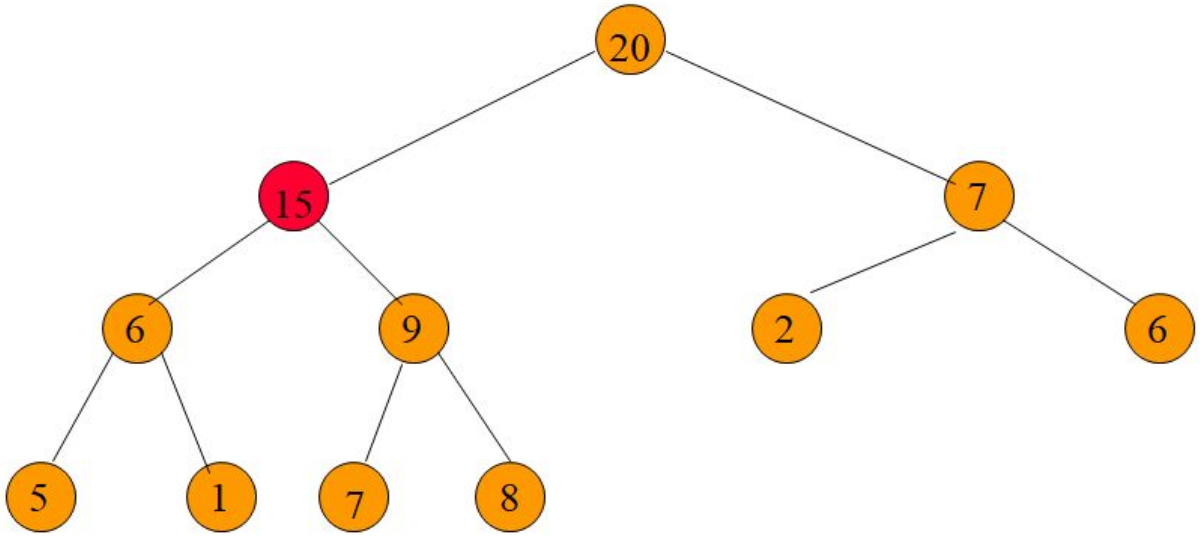
New element is 15.

Putting An Element Into A Max Heap



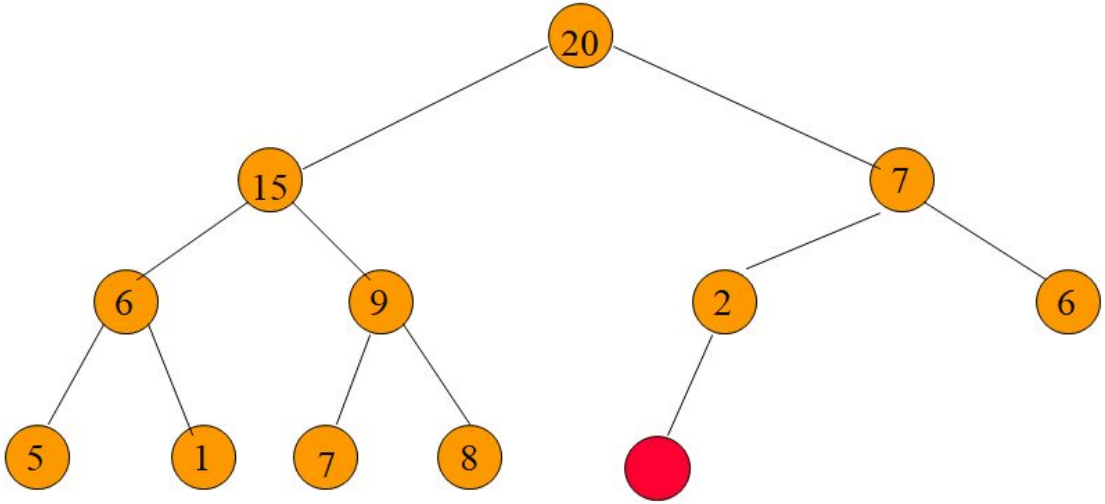
New element is 15.

Putting An Element Into A Max Heap



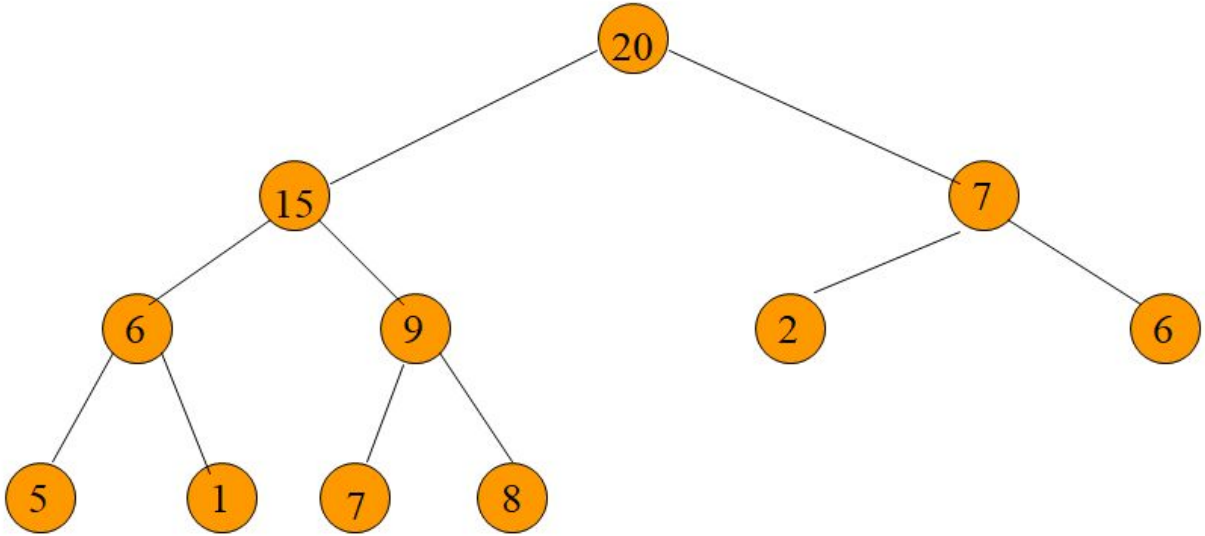
New element is 15.

Complexity Of Insert



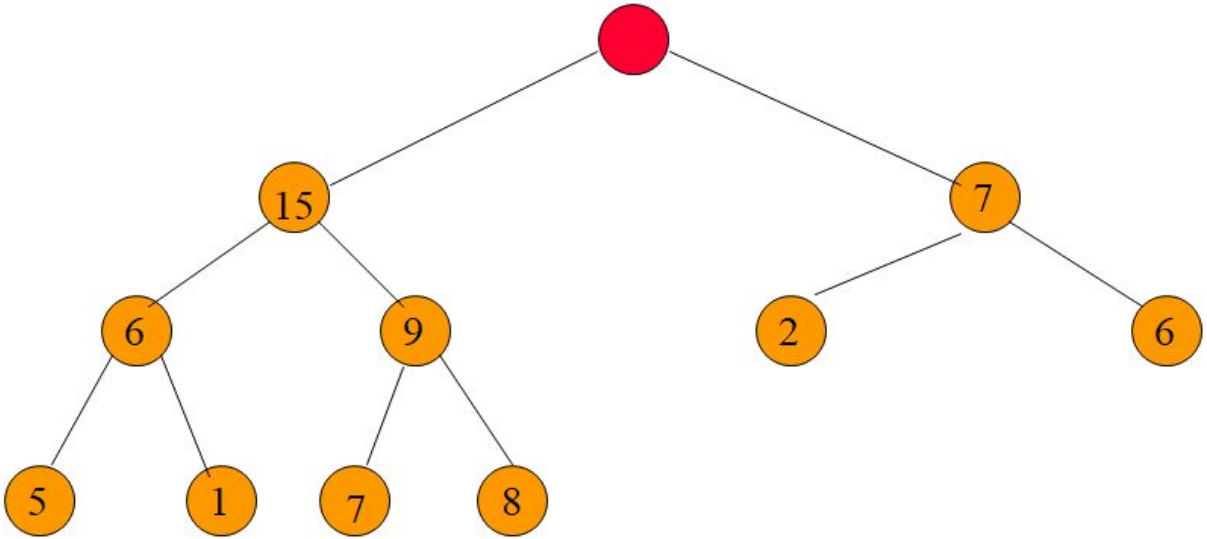
Complexity is O(log n), where n is heap size.

Removing The Max Element



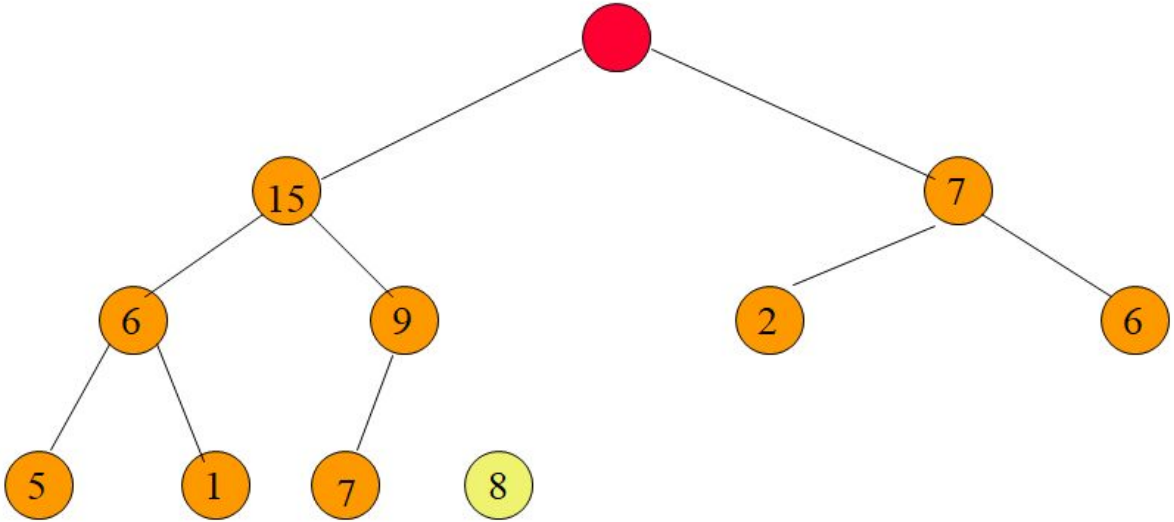
Max element is in the root.

Removing The Max Element



After max element is removed.

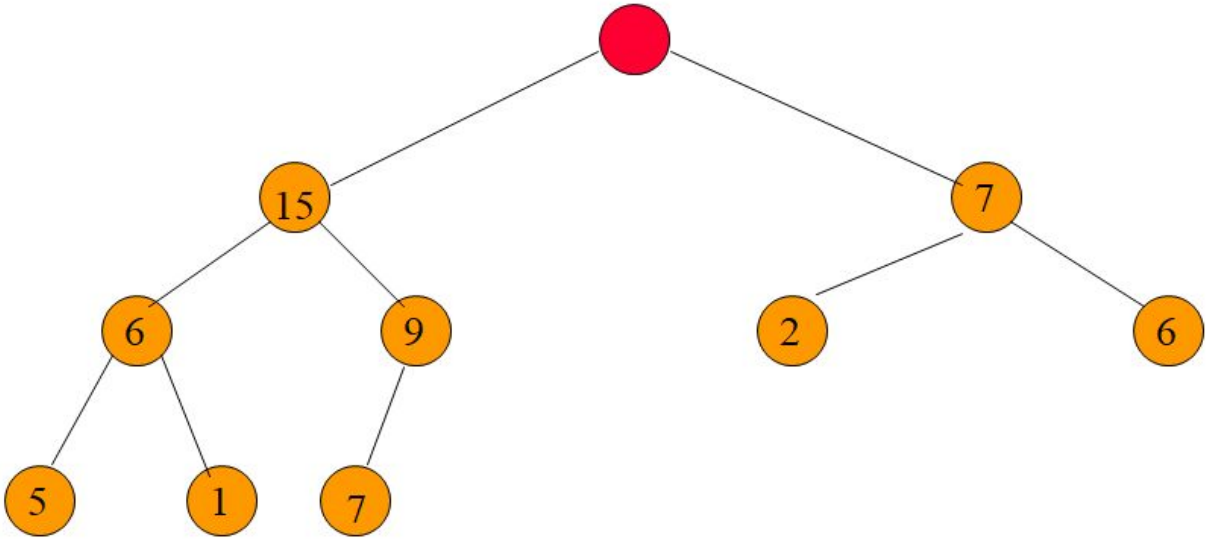
Removing The Max Element



Heap with 10 nodes.

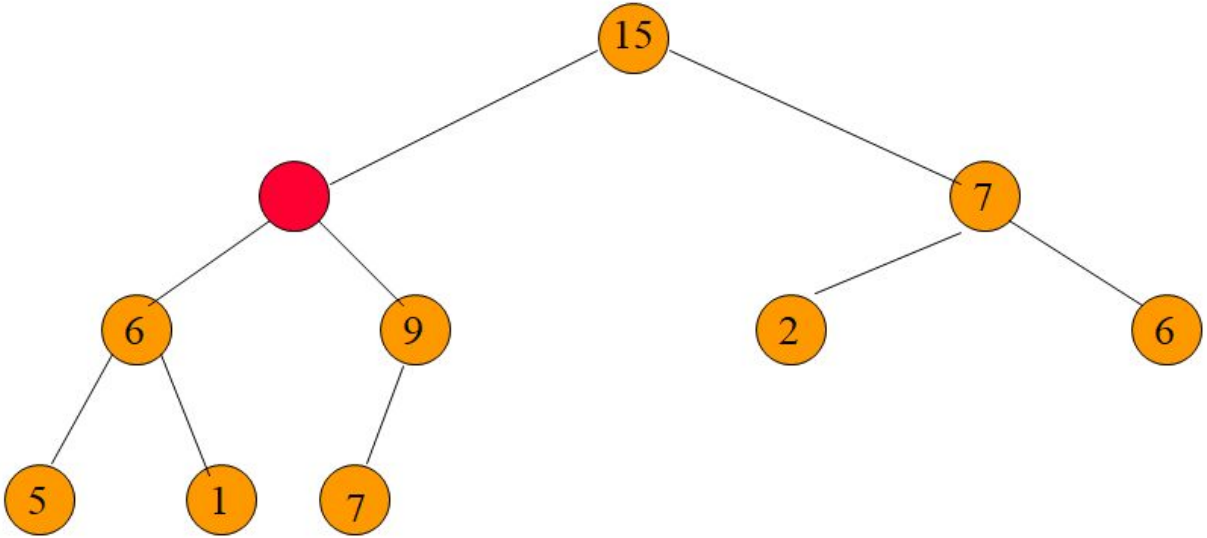
Reinsert 8 into the heap.

Removing The Max Element



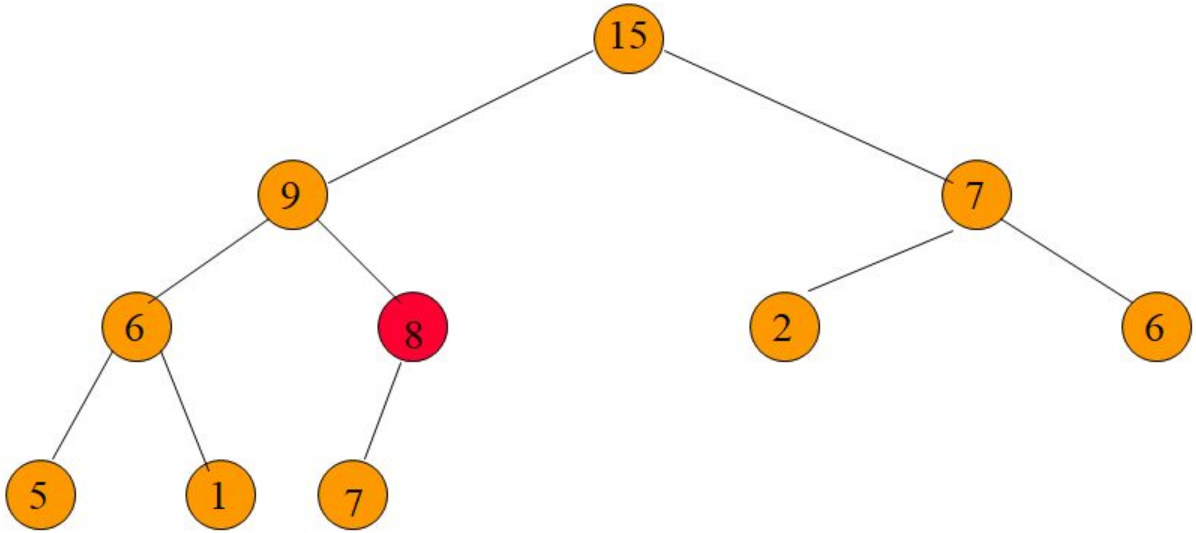
Reinsert 8 into the heap.

Removing The Max Element



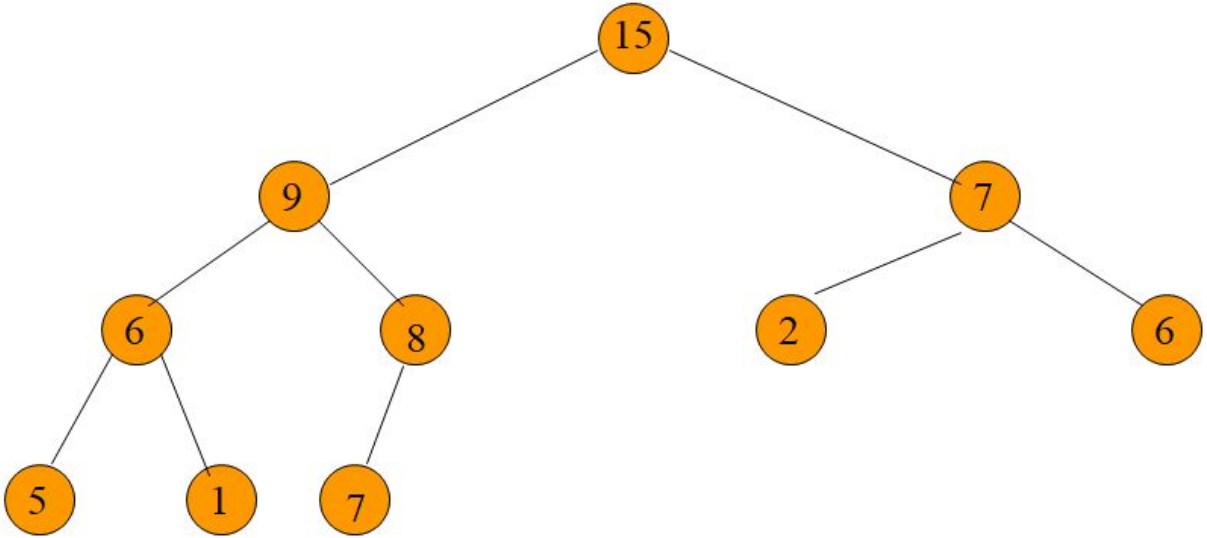
Reinsert 8 into the heap.

Removing The Max Element



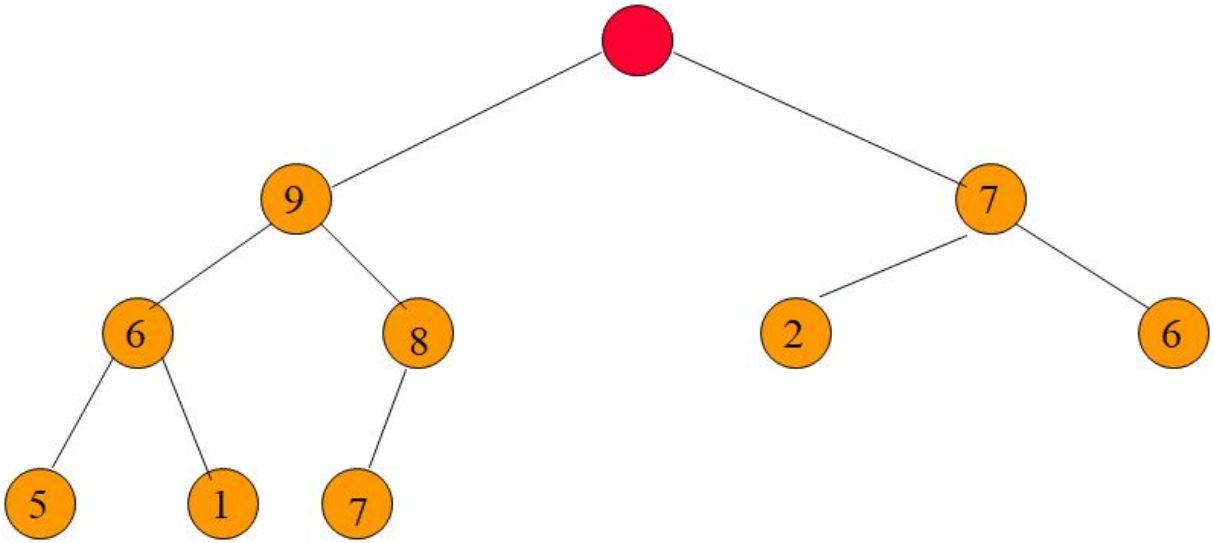
Reinsert 8 into the heap.

Removing The Max Element



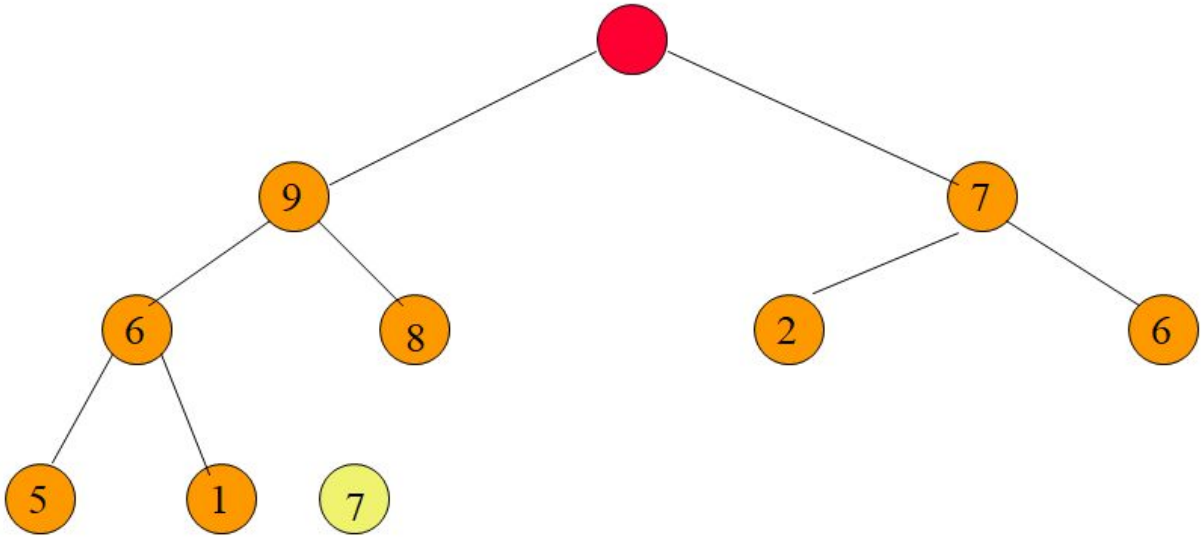
Max element is 15.

Removing The Max Element



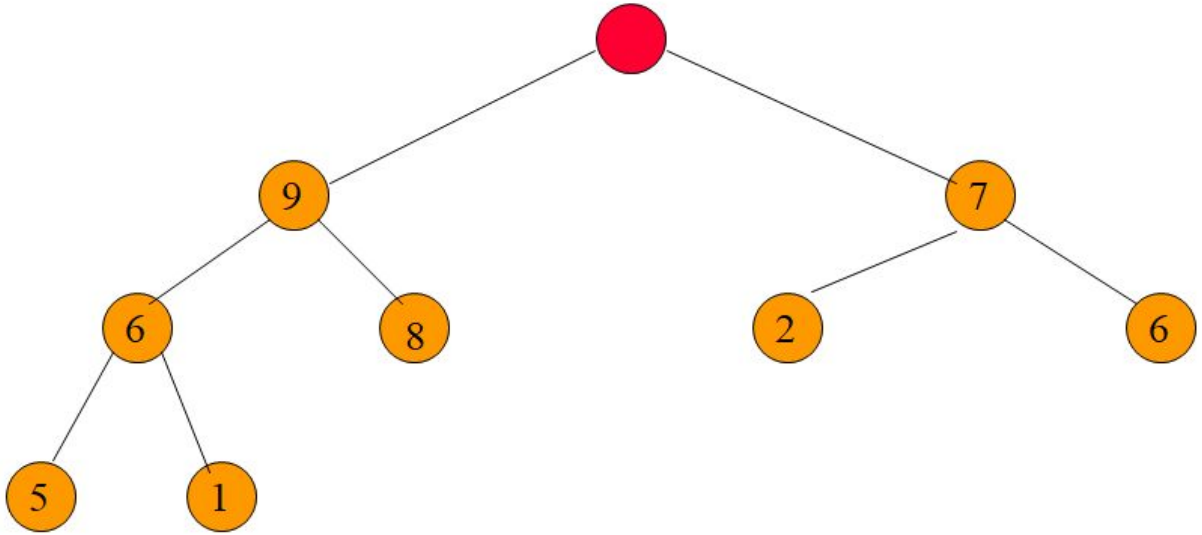
After max element is removed.

Removing The Max Element



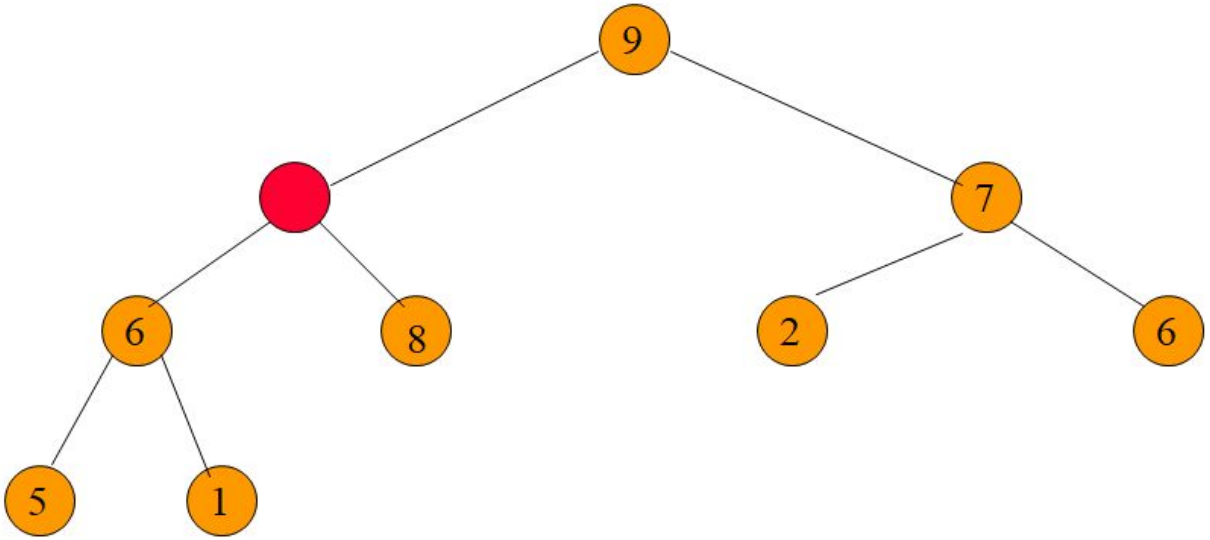
Heap with 9 nodes.

Removing The Max Element



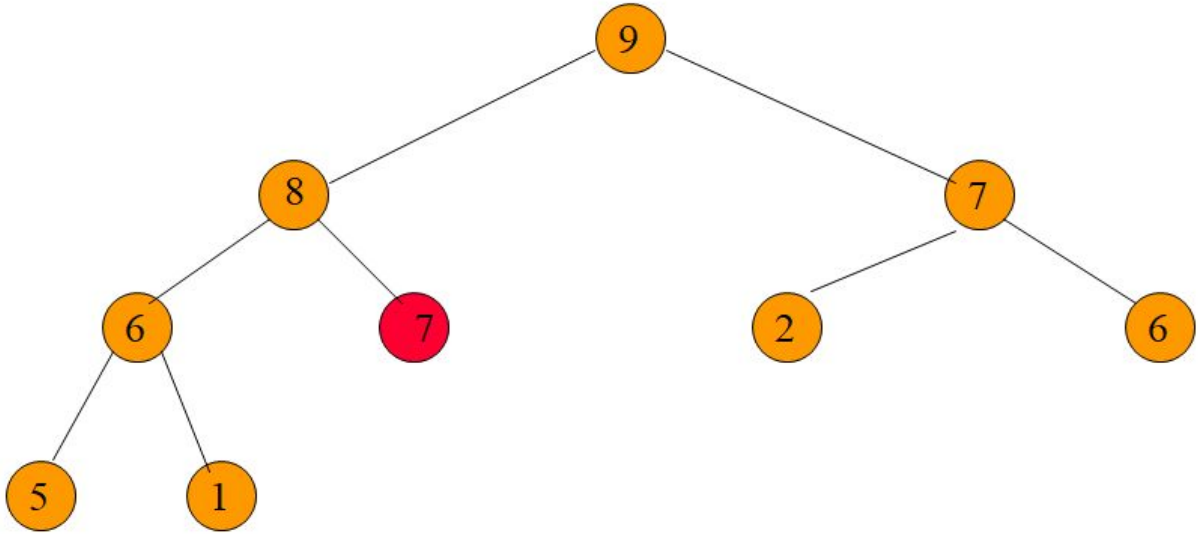
Reinsert 7.

Removing The Max Element



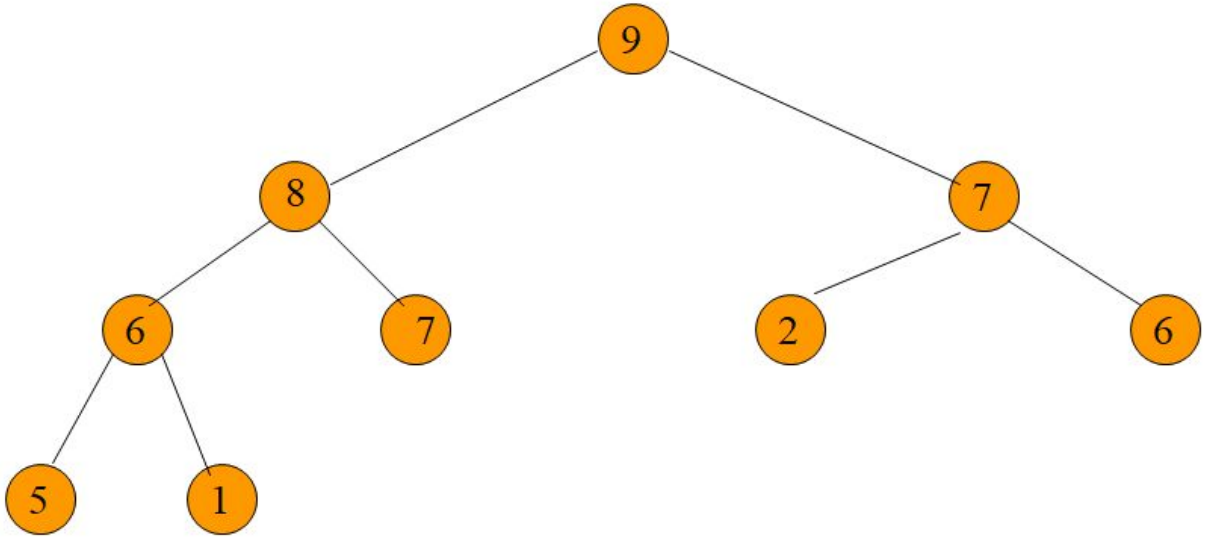
Reinsert 7.

Removing The Max Element

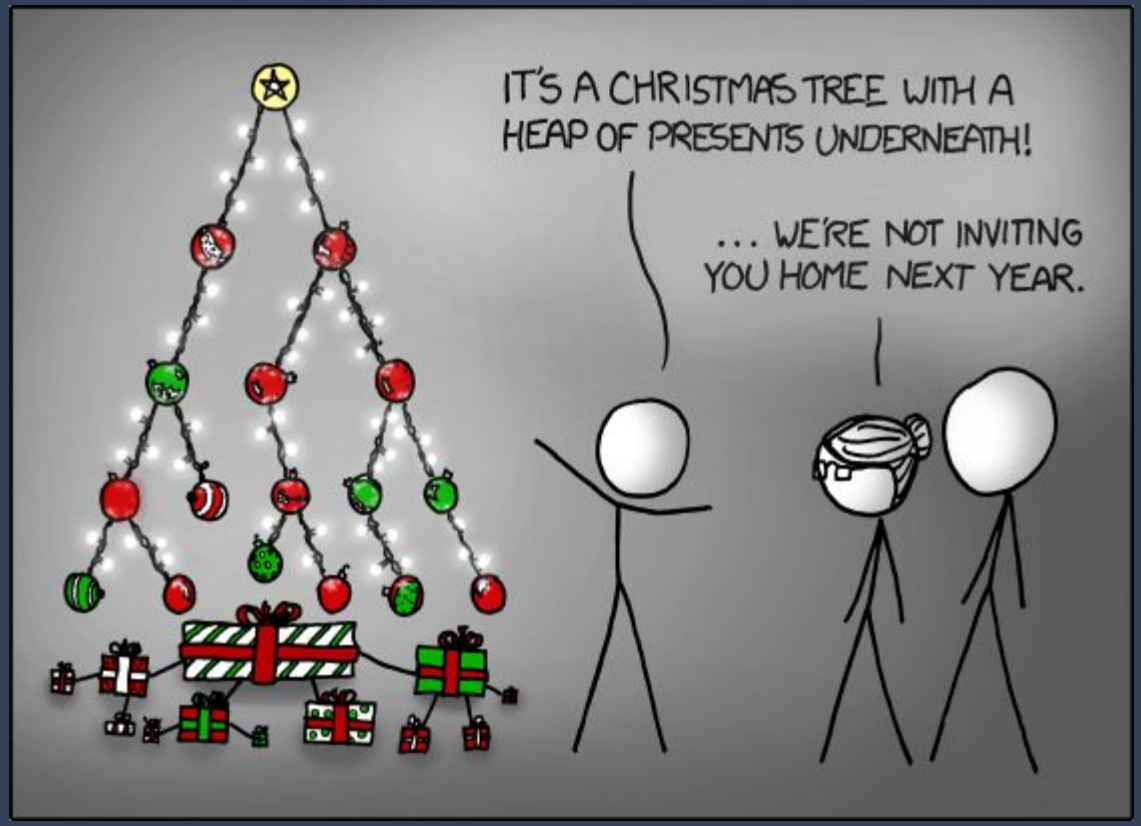


Reinsert 7.

Complexity Of Remove Max Element

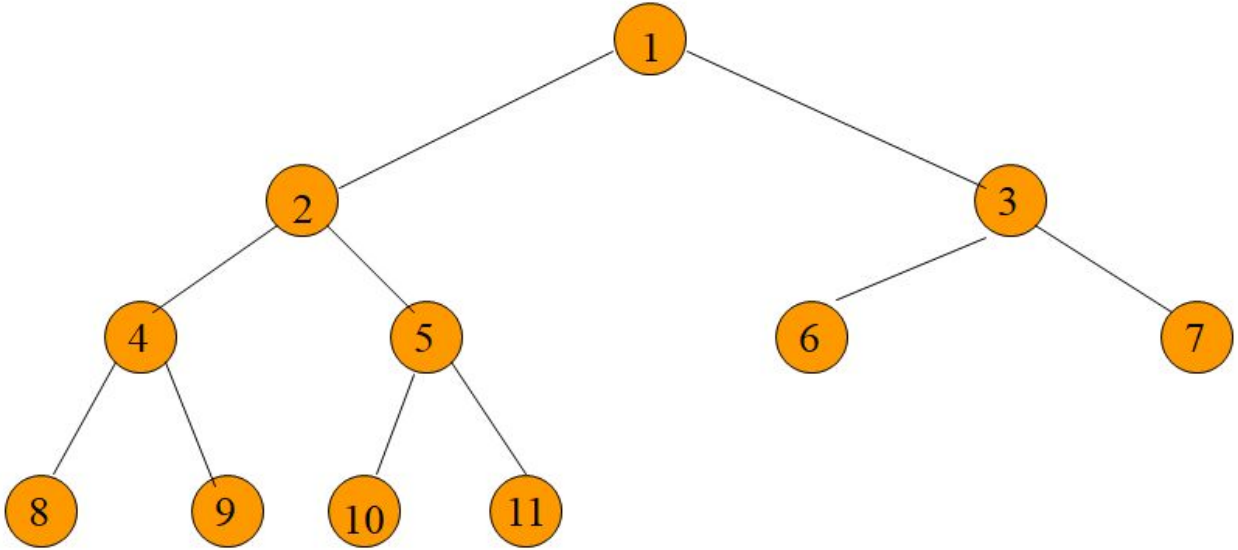


Complexity is O(log n).



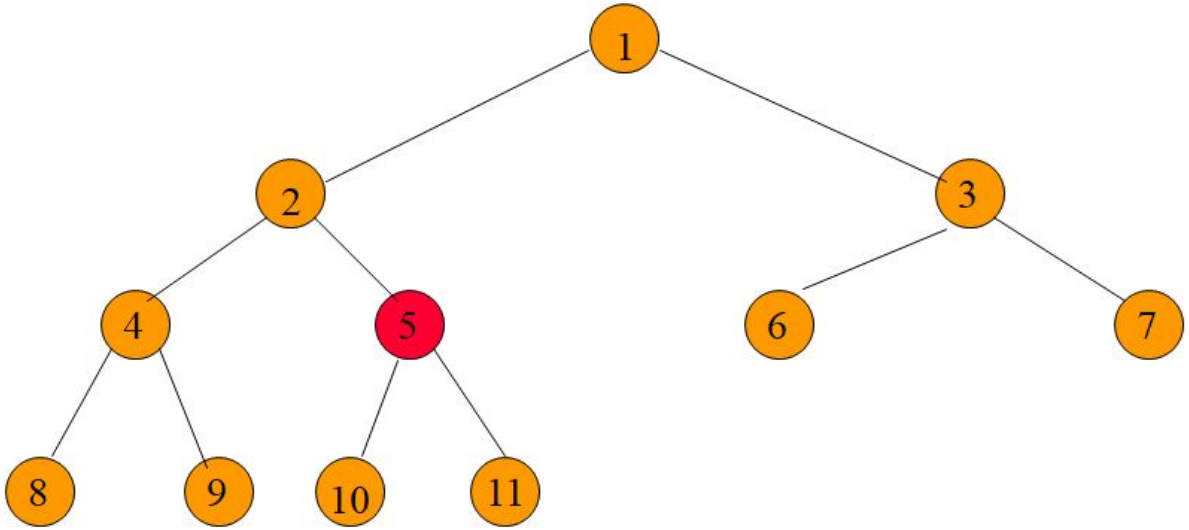
How do we build a heap in time better than O(n log n)?

Initializing A Max Heap



input array = [-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

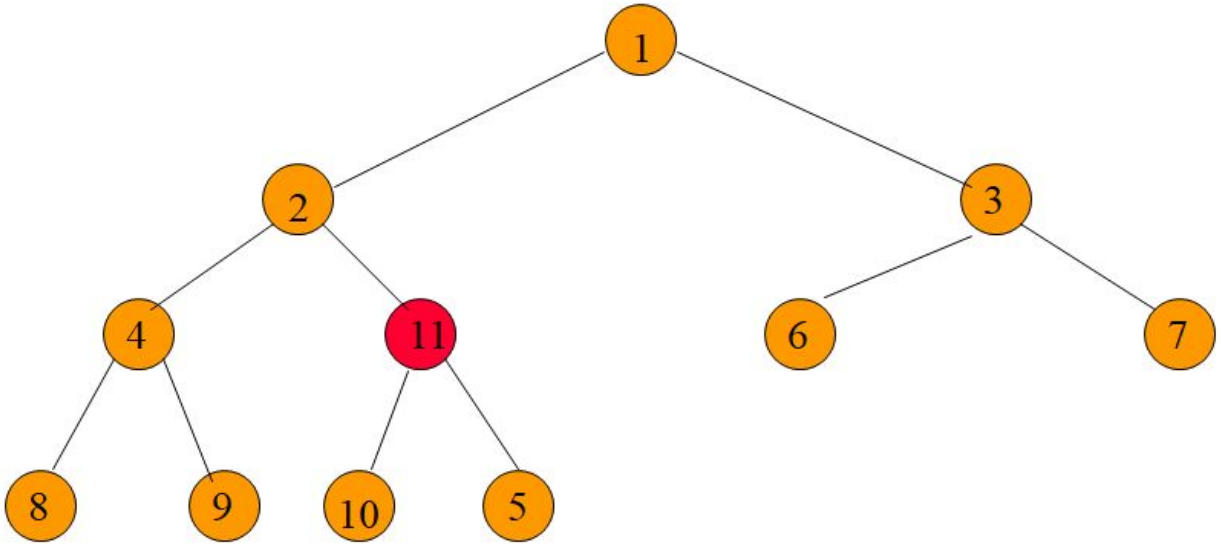
Initializing A Max Heap



Start at rightmost array position that has a child.

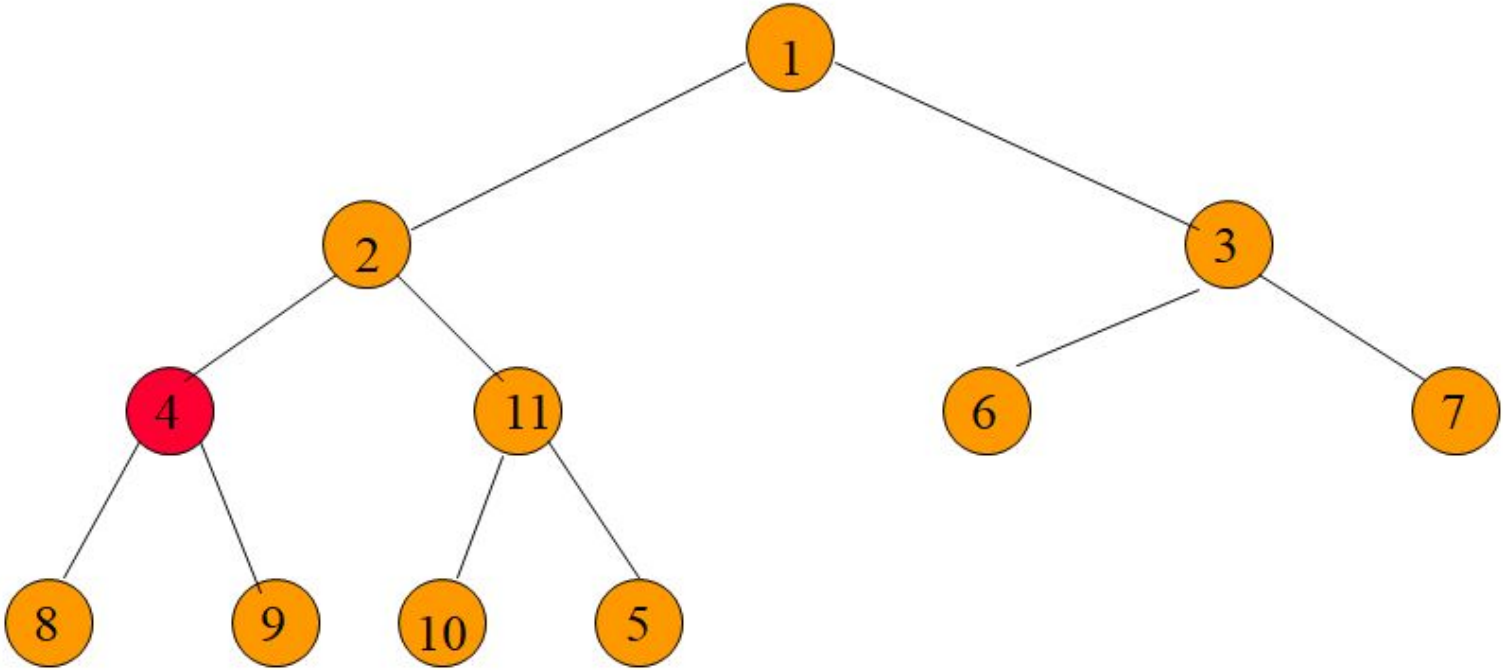
Index is n/2.

Initializing A Max Heap

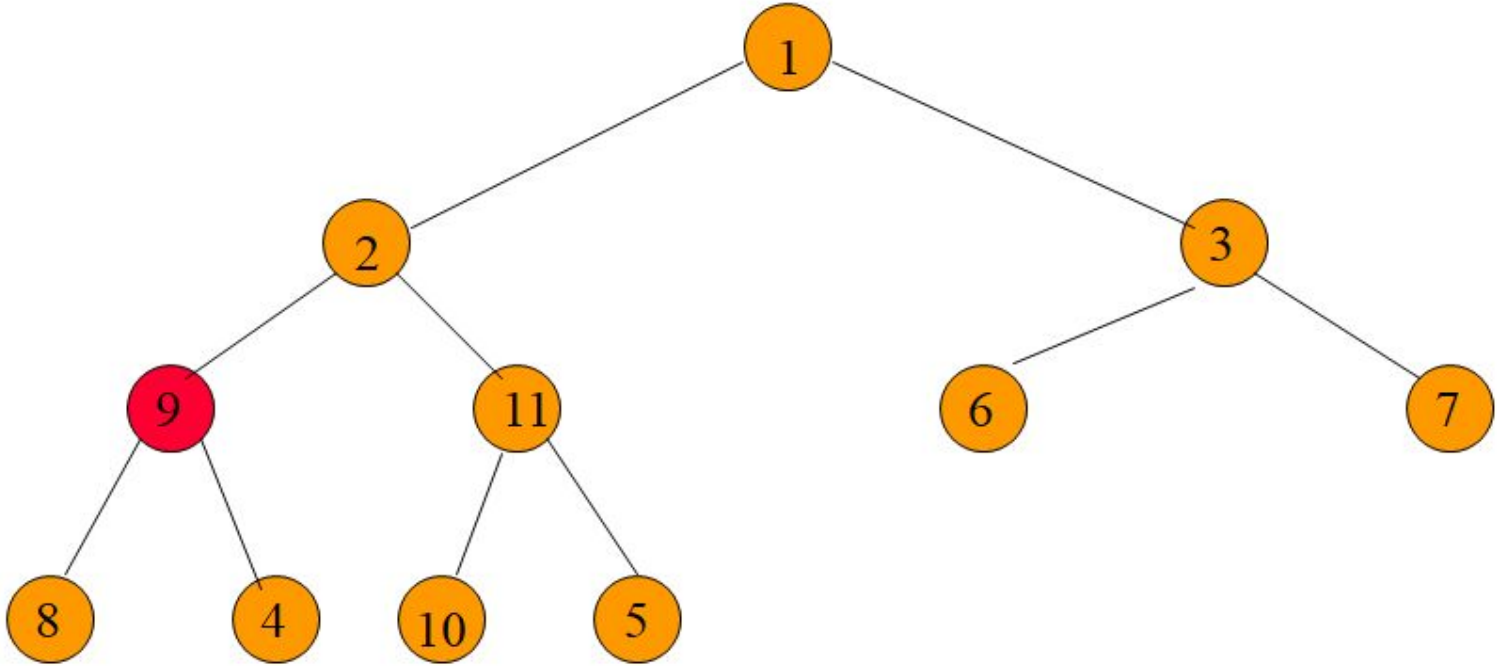


Move to next lower array position.

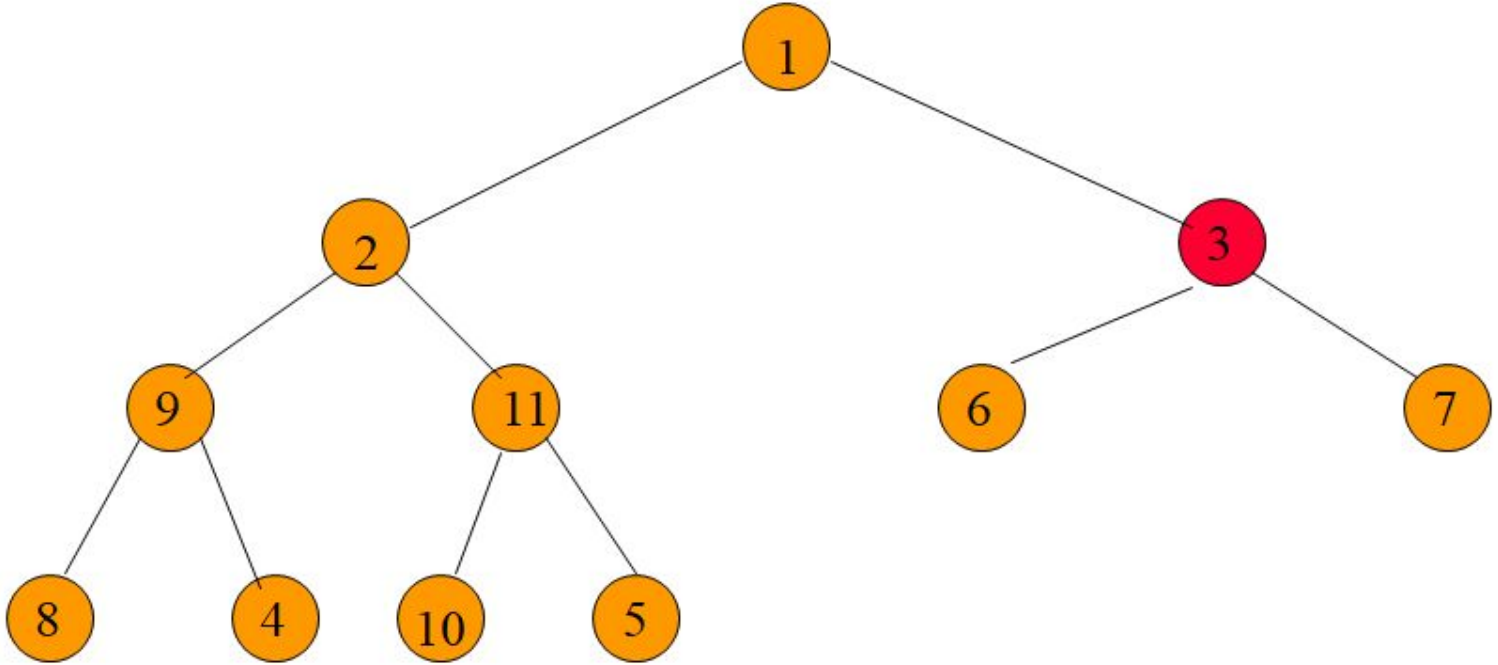
Initializing A Max Heap



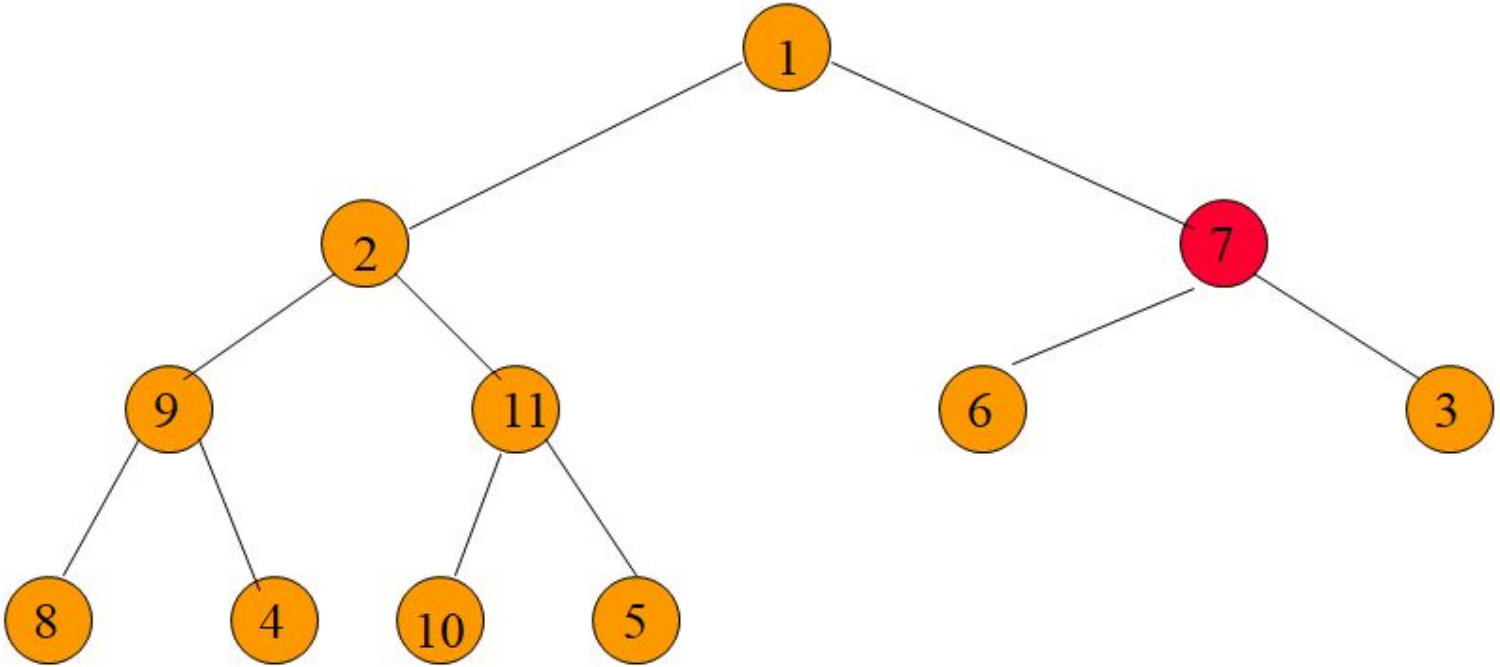
Initializing A Max Heap



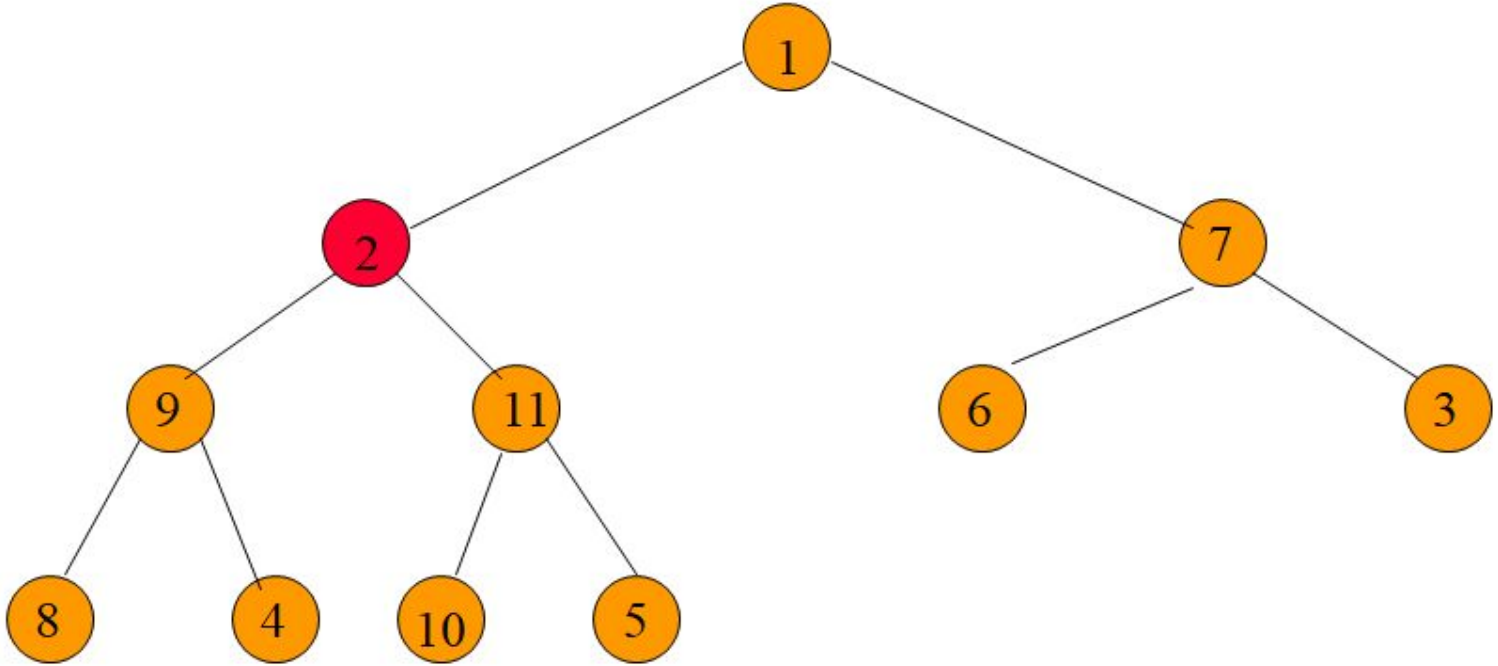
Initializing A Max Heap



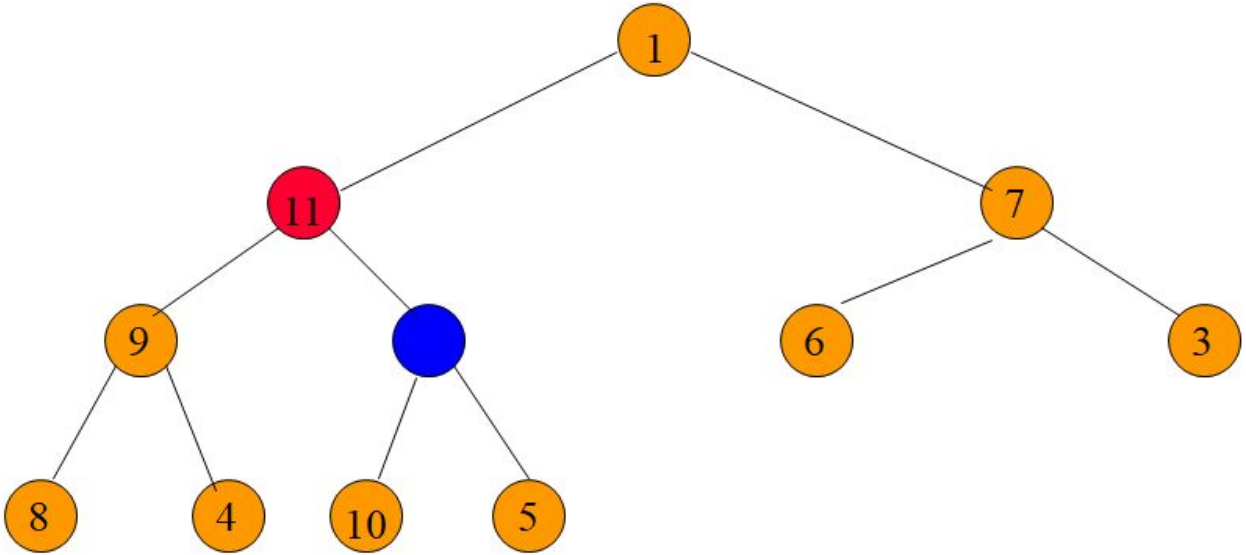
Initializing A Max Heap



Initializing A Max Heap

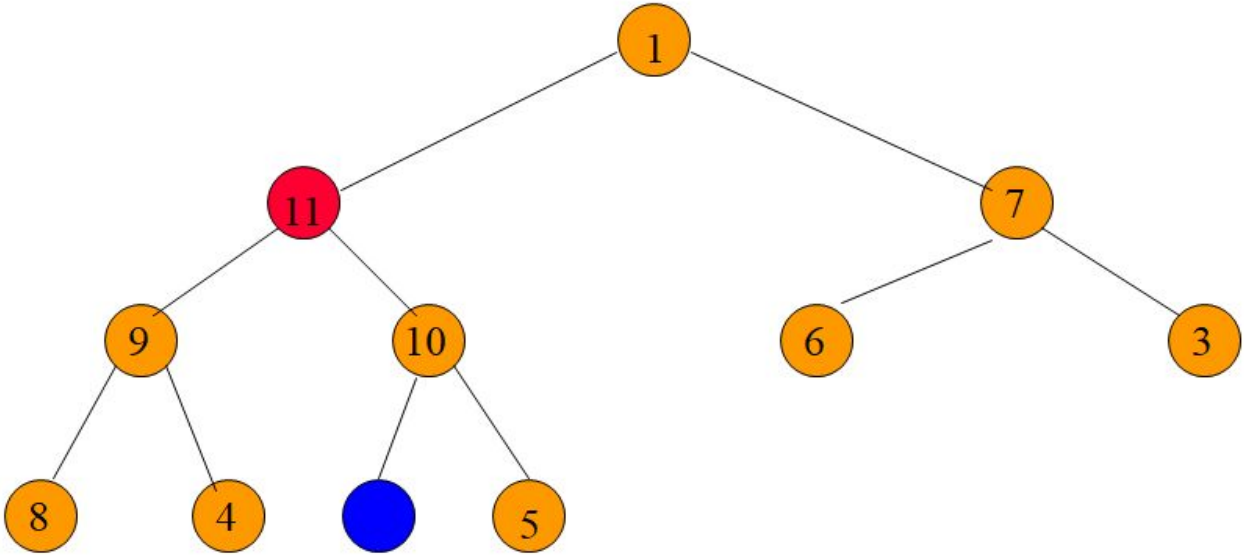


Initializing A Max Heap



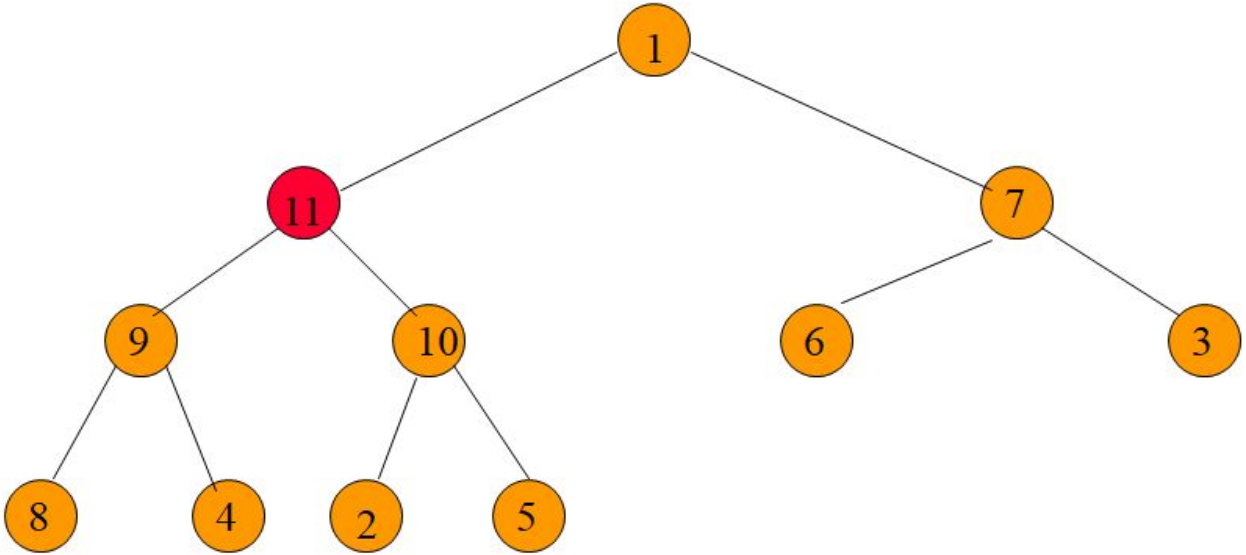
Find a home for 2.

Initializing A Max Heap



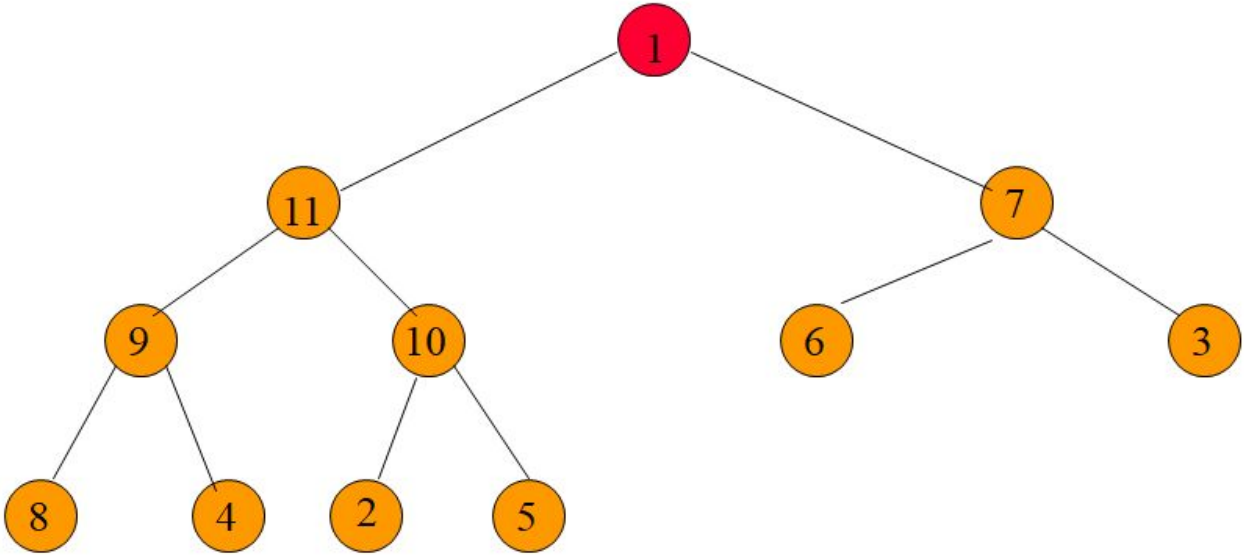
Find a home for 2.

Initializing A Max Heap



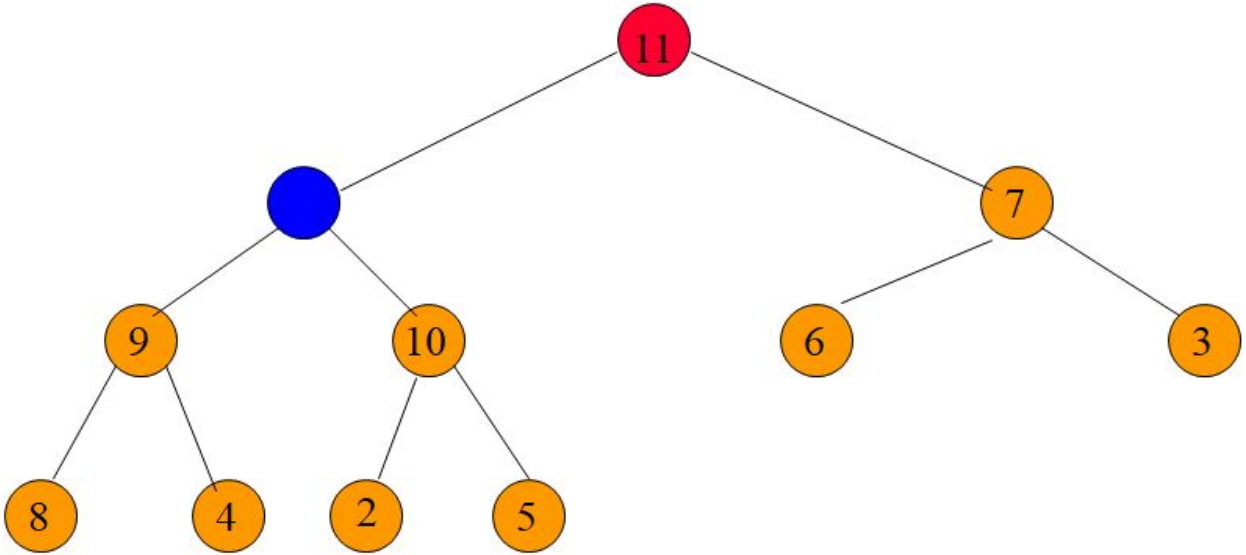
Done, move to next lower array position.

Initializing A Max Heap



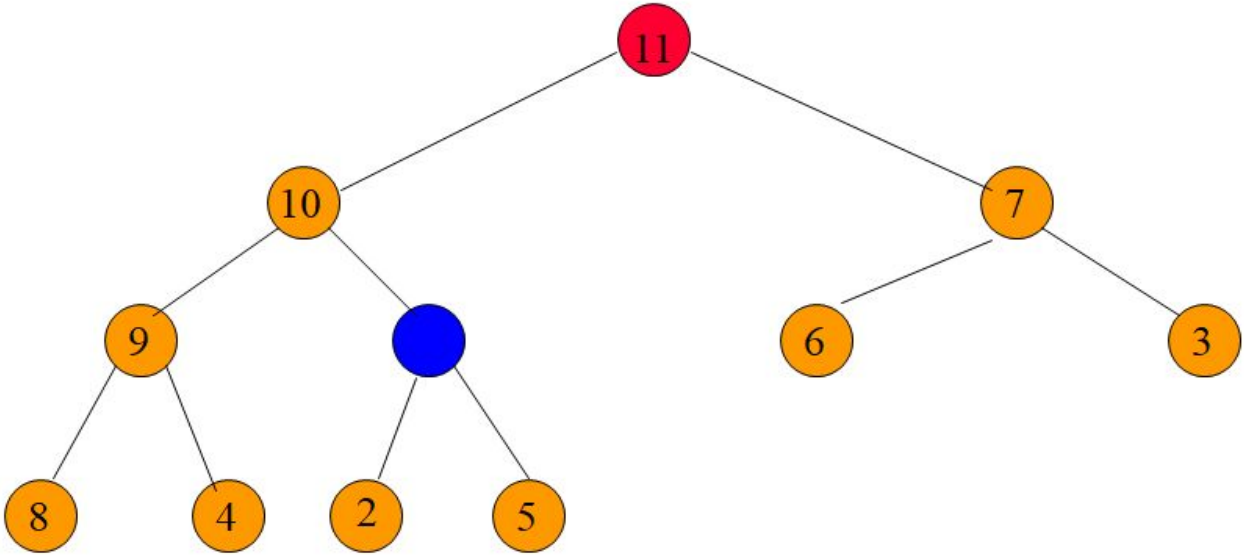
Find home for 1.

Initializing A Max Heap



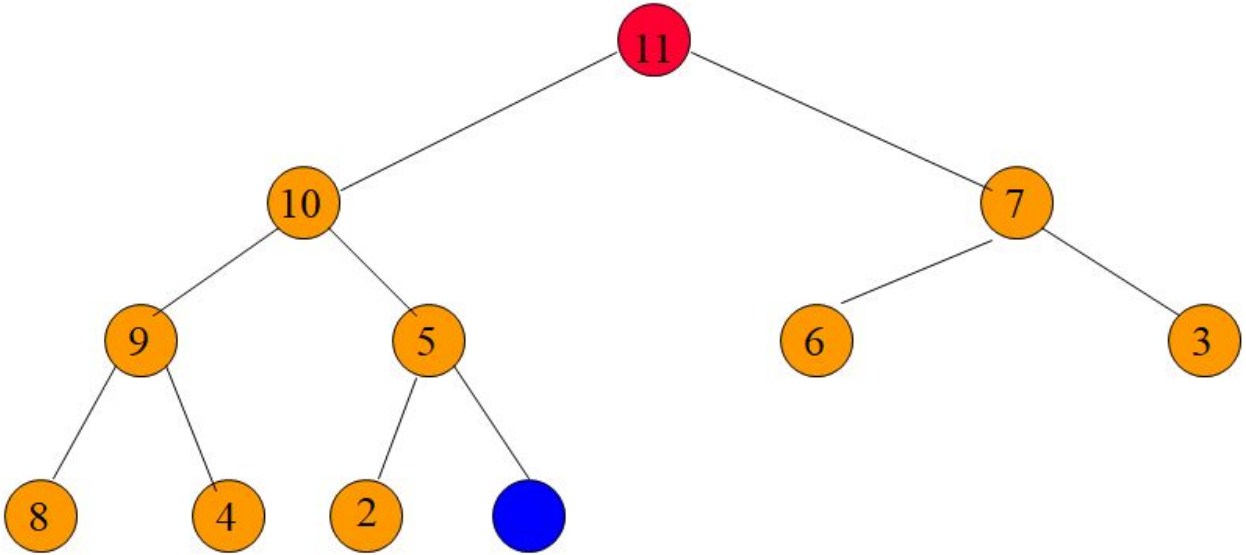
Find home for 1.

Initializing A Max Heap



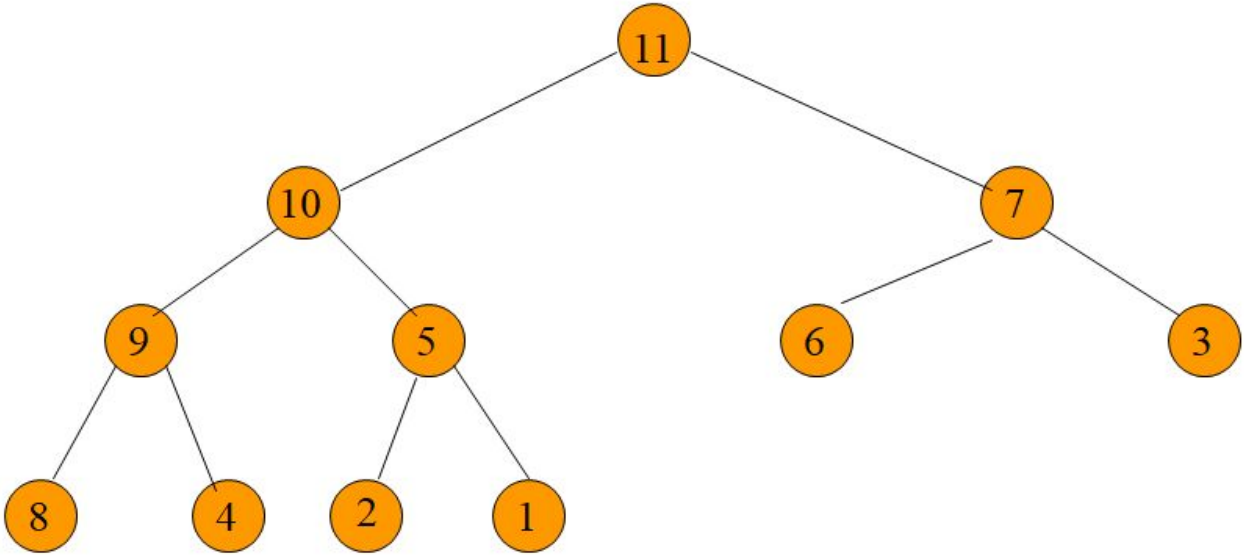
Find home for 1.

Initializing A Max Heap



Find home for 1.

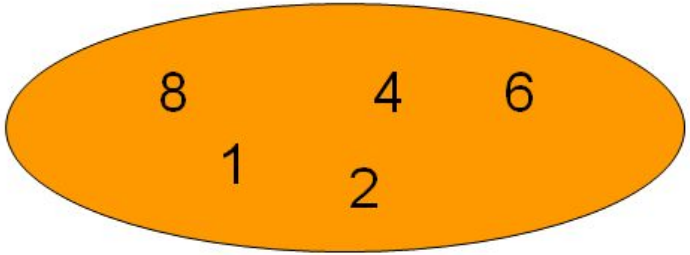
Initializing A Max Heap



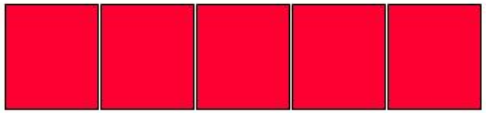
Time complexity = ?

Done.

After Putting Into Max Priority Queue

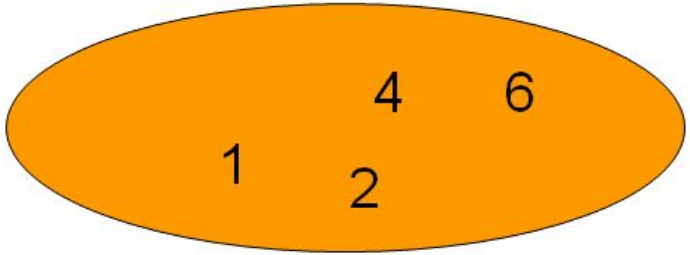


Max Priority Queue

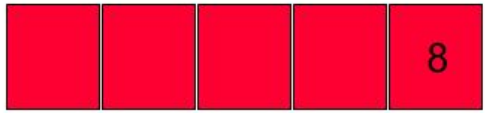


Sorted Array

After First Remove Max Operation

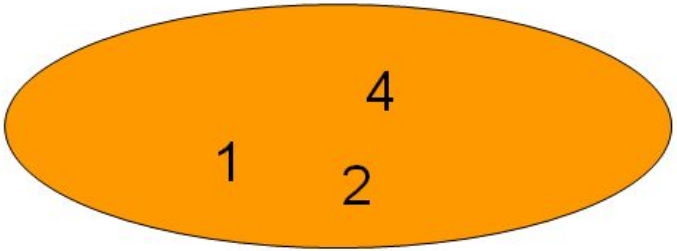


Max Priority Queue

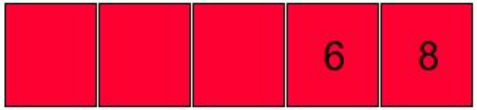


Sorted Array

After Second Remove Max Operation

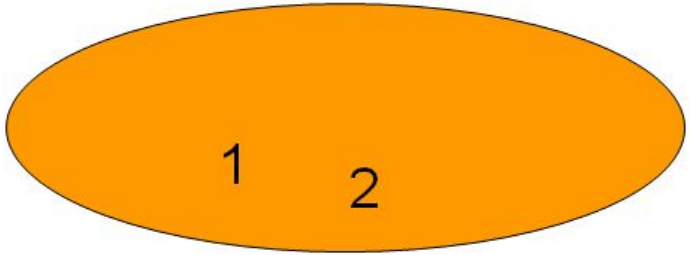


Max Priority Queue



Sorted Array

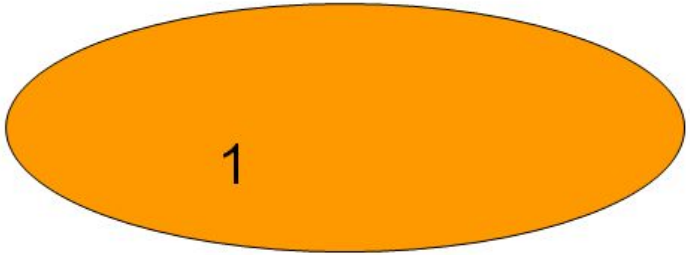
After Third Remove Max Operation



Max Priority Queue

Sorted Array

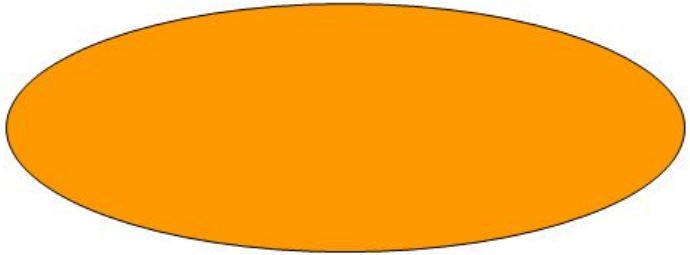
After Fourth Remove Max Operation



Max Priority Queue

Sorted Array

After Fifth Remove Max Operation



Max Priority Queue

Sorted Array

Heap Sort can be done in place.

In O(n log n) worst-case time.

Heap Sort can be done in place.
In O(n log n) worst-case time.
But it is not stable. Why?

Heap Sort can be done in place.
In O(n log n) worst-case time.
But it is not stable.
Cache performance also not good.
Why?

```
#MAXHEAP
class PriorityQueue:

  def __init__(self):
    self.A = ['SENTINEL']
    self.size = 0

  def parent(self, i): #Index of parent for node i
    if i >= 1 and i <= self.size:
      return i / 2
    else:
      return -1 #There is no parent

  def leftchild(self, i):
    if i >= 1 and 2*i <= self.size:
      return 2*i
    else:
      return -1

  def rightchild(self, i):
    if i >= 1 and 2*i+1 <= self.size:
      return 2*i + 1
    else:
      return -1
```

```
def insert(self, key):
  self.A.append(key)
  self.size += 1
  curr = self.size #index of the element right now as it is being bubbled up
  while self.parent(curr) != -1 and self.A[self.parent(curr)] < self.A[curr]:
    (self.A[self.parent(curr)],self.A[curr]) = (self.A[curr],self.A[self.parent(curr)])
    curr = self.parent(curr)

 def sink(self,i):
  if i < 1 or i > self.size:
    return
  left = self.leftchild(i)
  right = self.rightchild(i)
  if left == -1:
    return
  if right == -1:
    largerchild = left
  else:
     if self.A[left] > self.A[right]:
       largerchild = left
     else:
        largerchild = right
  if self.A[i] < self.A[largerchild]:
    #swap the two to sink the heavier element down
    (self.A[i],self.A[largerchild]) = (self.A[largerchild],self.A[i])
    self.sink(largerchild)
```

```
def extractmax(self):
    if self.size == 0:
      return None
    #Swap the top element with the last element
    (self.A[1],self.A[self.size]) = (self.A[self.size],self.A[1])
    #Heapify-down the new first element
    maxvalue = self.A[self.size]
    self.size -= 1
    self.sink(1)
    return maxvalue

  def buildheap(self,array):
    if self.size != 0:
      return
    self.A.extend(array)
    print len(self.A)
    self.size = len(self.A) - 1
    for i in range(self.size,0,-1):
      self.sink(i)

  def heapsort(self):
    if self.size == 0:
      return
    numelements = self.size
    for i in range(numelements): #Do as many deletions as number of elements
      self.extractmax()
    return self.A[1:numelements+1]
```

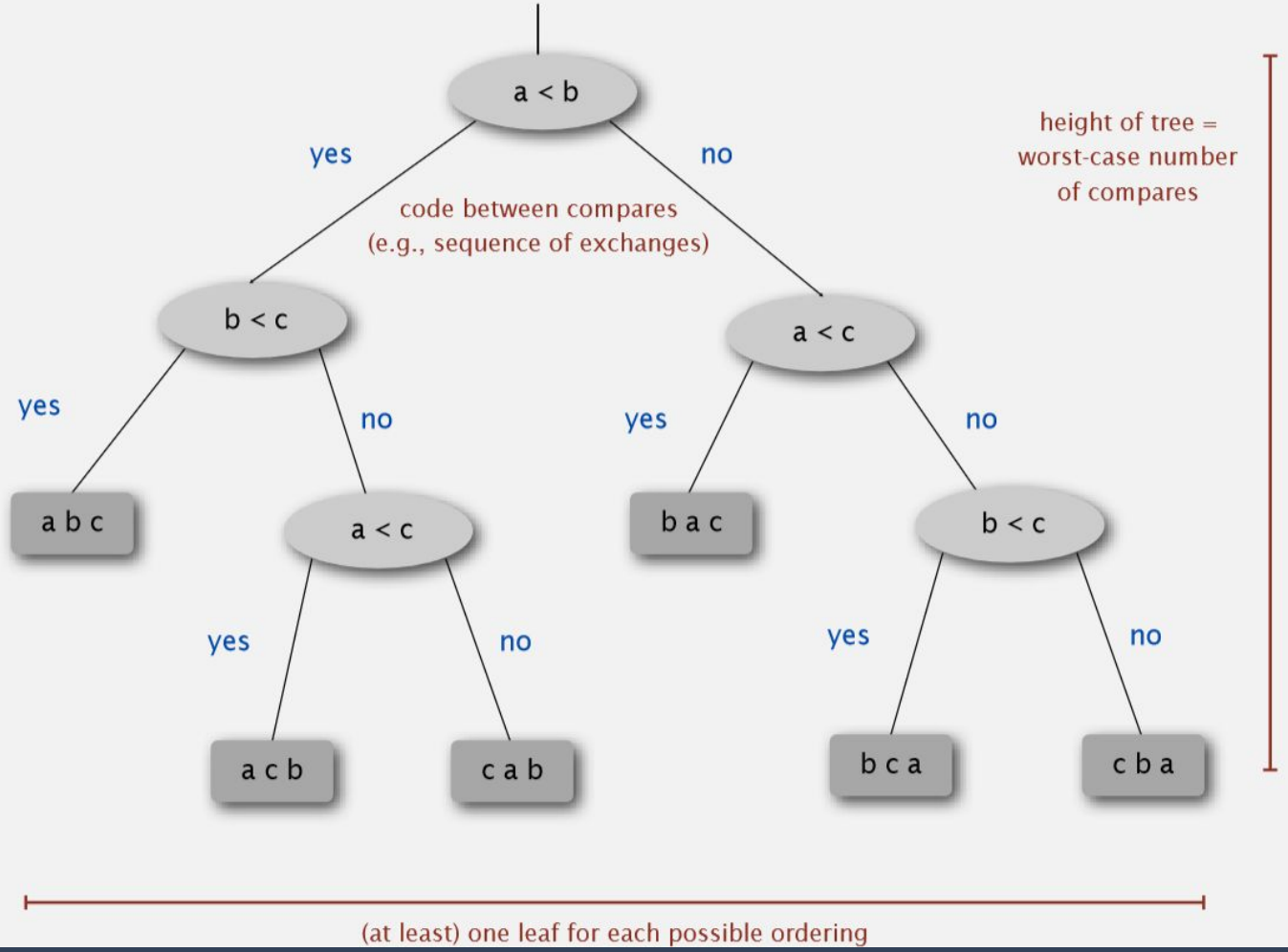
What if we want to search for an element in a binary heap?

Lower bound on worst-case complexity of sorting

All the algorithms we have seen use comparison operations.

If we can get a lower bound on just comparison operations, that would also be a lower bound on the entire algorithm.

Construct a decision tree for visualising the execution of a sorting algorithm.
- Internal nodes are comparisons, which yield a binary yes/no decision
- Leaf nodes represent the found answers
- The execution of an algorithm represented by a path from root to leaf
- The path length is the running time.
- Height of the tree is the worst-case running time

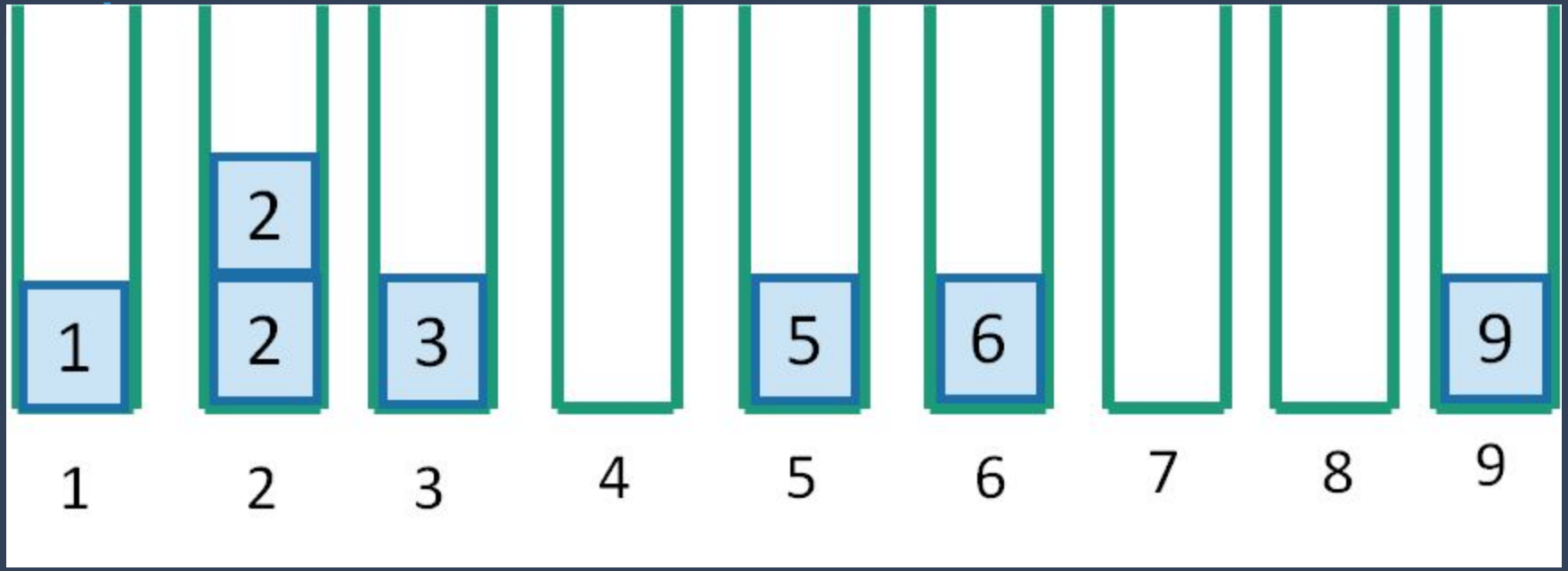


Comparison-based sorting: Worst-case Ω(n log n)

Sorting an array with many entries of ‘small’ values

You are given an array of student objects. Each student has an integer-valued age field that is to be treated as a key. Rearrange the elements of the array so that the ages appear in sorted order.

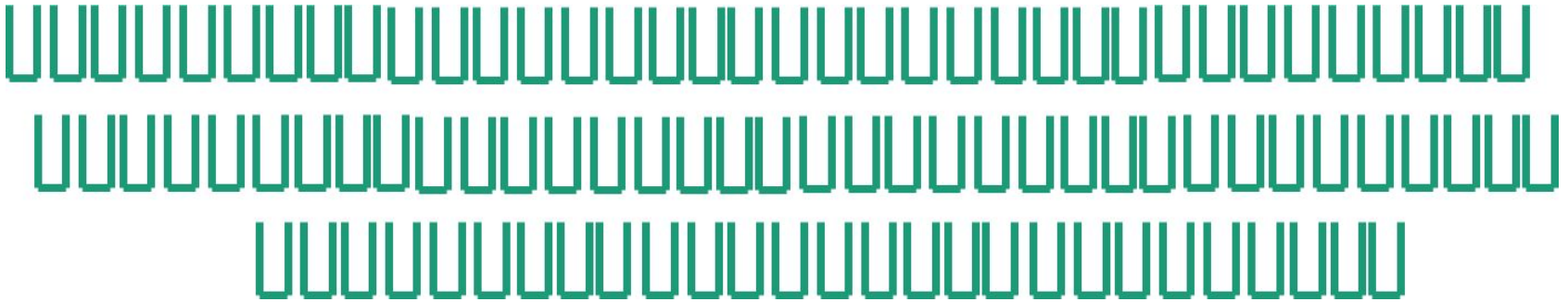
Just count how many students of each age there



- Need to be able to know what bucket to put something in.
  - We assume we can evaluate the items directly, not just by comparison
- Need to know what values might show up ahead of time.

2	12345	13	2^{1000}	50	100000000	1

- Need to assume there are not too many such values.



Time complexity = ?

Assume there are n integers in the range {0,1,2...,k-1}

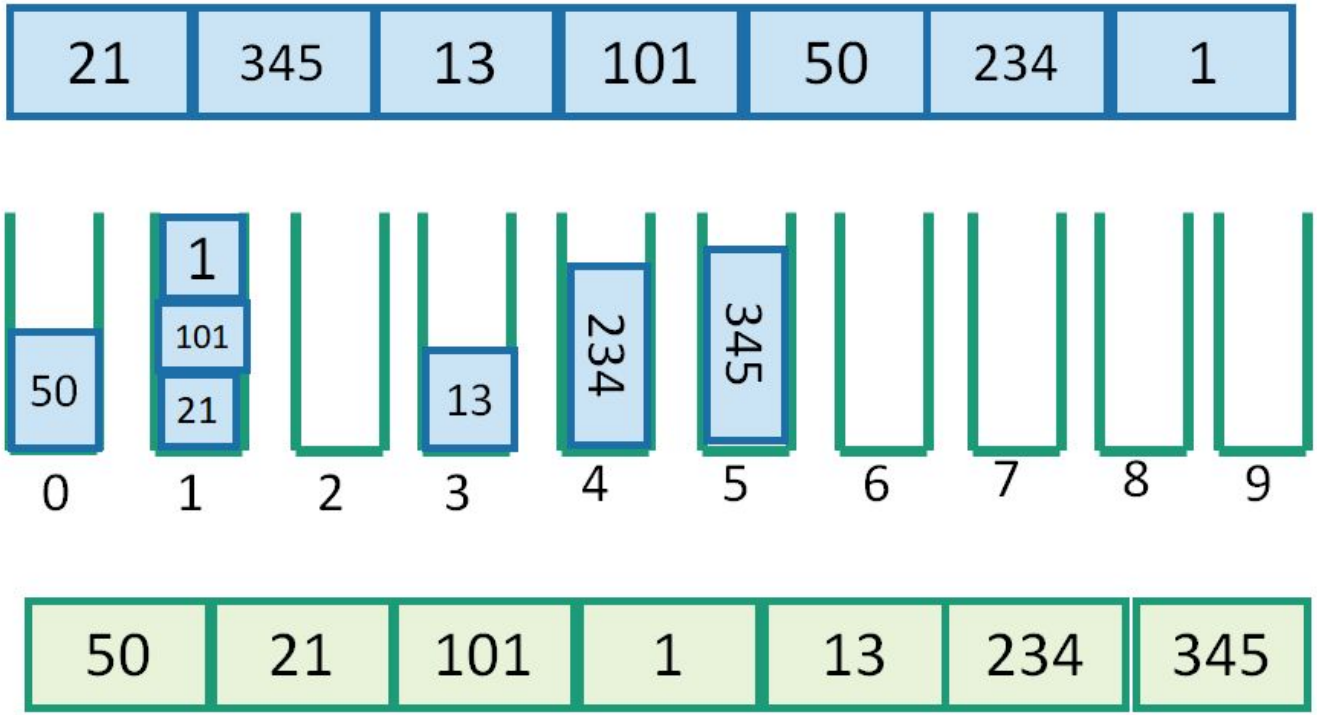
How much extra space is needed?

What if we want to keep the sort stable?

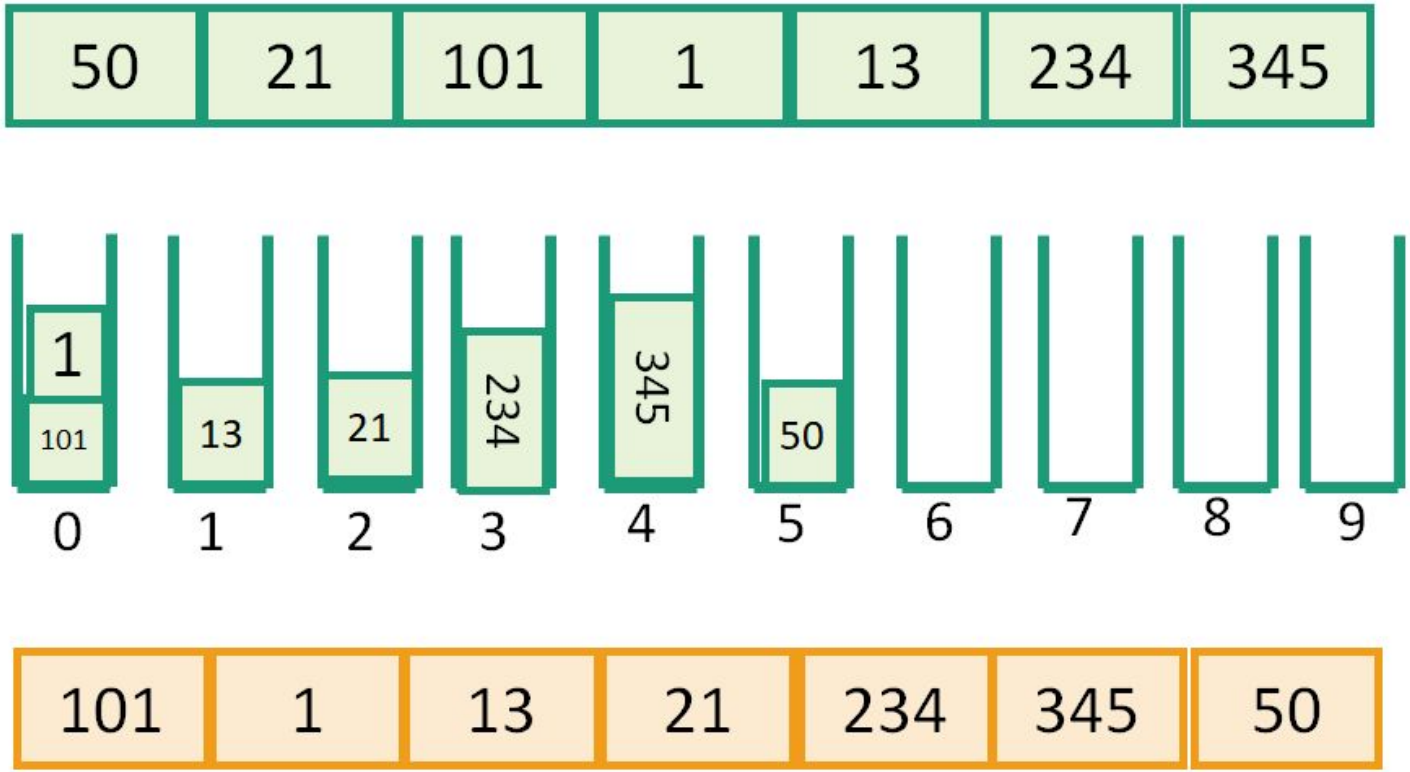
Radix Sort

n integers, d digits, in some base (base-10 in the example)

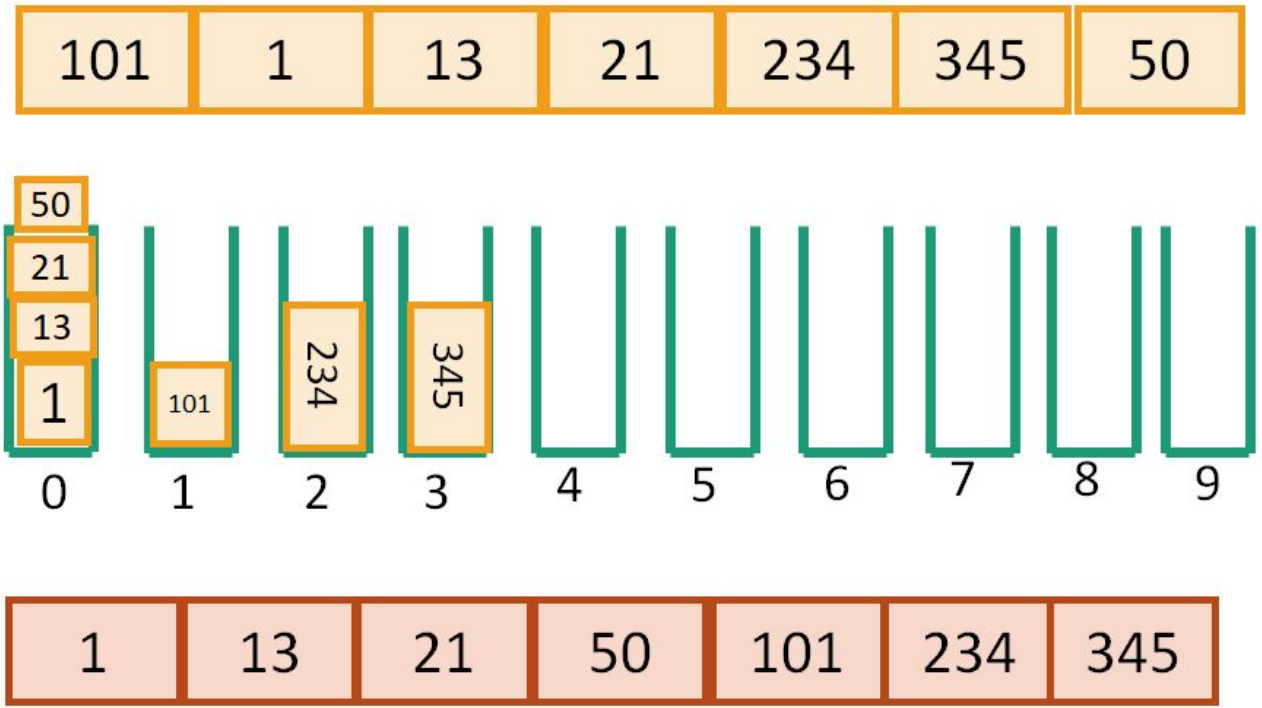
Step 1: CountingSort on least significant digit



Step 2: CountingSort on the 2nd least sig. digit



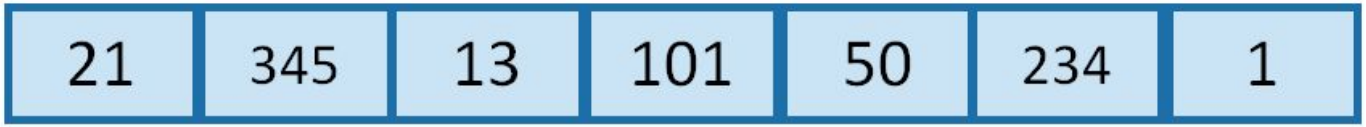
Step 3: CountingSort on the 3rd least sig. digit



It worked!!

Why does this work?

Original array:



Next array is sorted by the first digit.

Next array is sorted by the first two digits.

Next array is sorted by all three digits.

Sorted array

What is the running time?

for RadixSorting numbers base-10.

- Suppose we are sorting n d-digit numbers (in base 10).

e.g., n=7, d=3:

021	345	013	101	050	234	001

1. How many iterations are there?
   - d iterations
2. How long does each iteration take?
   - Time to initialize 10 buckets, plus time to put n numbers in 10 buckets. O(n).
3. What is the total running time?
   - O(nd)

This doesn't seem so great

- To sort n integers, each of which is in {1,2,…,n}…
- d = $\lfloor \log_{10}(n) \rfloor + 1$
  - For example:
    - n = 1234
    - $\lfloor \log_{10}(1234) \rfloor + 1 = 4$
- Time = $O(nd) = O(n \log(n))$.
  - Same as MergeSort!

Can we do better?

- RadixSort base 10 doesn't seem to be such a good idea…
- But what if we change the base? (Let's say base r)
- We will see there's a trade-off:
  - Bigger r means more buckets
  - Bigger r means fewer digits

General running time of RadixSort

- Say we want to sort:
  - n integers,
  - maximum size M,
  - in base r.
- Number of iterations of RadixSort:
  - Same as number of digits, base r, of an integer x of max size M.
  - That is $d = \lfloor \log_r(M) \rfloor + 1$
- Time per iteration:
  - Initialize r buckets, put n items into them
  - $O(n + r)$ total time.
- Total time:
  - $O\big(d \cdot (n + r)\big) = O\big((\lfloor \log_r(M) \rfloor + 1) \cdot (n + r)\big)$

A reasonable choice: r=n

- Running time:

$$O\left(\left(\lfloor \log_r(M) \rfloor + 1\right) \cdot (n + r)\right)$$

Intuition: balance n and r here.

- Choose n=r:

$$O\left(n \cdot \left(\lfloor \log_n(M) \rfloor + 1\right)\right)$$

What have we learned?

You can put any constant here instead of 100.

- RadixSort can sort n integers of size at most n^{100} in time O(n), and needs enough space to store O(n) integers.
- If your integers have size much much bigger than n (like 2^n), maybe you shouldn't use RadixSort.
- It matters how we pick the base.

“I think the bubble sort is the wrong way to go”

Sorting Algorithms Algorithms

What is the most efficient way to sort a million 32-bit integers?

Anders Kaseorg, MIT PhD student in Computer Science
Updated Jun 11, 2014

Since we know we're sorting fixed-size integers, radix sort is much more efficient than any comparison-based sort (quicksort, introsort, merge sort, etc.).

Here are some benchmarks on my computer:

- quicksort (hand-coded): 79 milliseconds
- introsort (`std::sort`): 71 milliseconds
- merge sort (`std::stable_sort`): 62 milliseconds
- merge sort (glibc's `qsort`) 148 milliseconds—over twice as long because of the function pointer overhead
- base-2^8 LSD radix sort (hand-coded, code below): 14 milliseconds
- base-2^8 LSD radix sort with one initial MSD pass (see Gregory Popovitch's comment): 11 milliseconds

The President is correct that bubble sort would be the wrong way to go—that would take about 33 *minutes*!

https://www.quora.com/What-is-the-most-efficient-way-to-sort-a-million-32-bit-integers

COW	BAR
DOG	EAR
SEA	TAR
RUG	DIG
ROW	BIG
MOB	TEA
BOX	NOW
TAB	FOX

Bucket Sort

Divides the entire range of values into a set of buckets.

Generalization of Counting Sort (where bucket size = 1)

Steps -

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