# Lecture #6
# Sort (3)

Algorithm

JBNU Spring 2021

Jinhong Jung

# In Previous Lecture

## Desired properties of a sorting algorithm

| Name | Stable | In-place : Extra memory | # of comparisons | | # of swaps | | Adaptive |
|---|---|---|---|---|---|---|---|
| | | | Best | Worst | Best | Worst | |
| Selection | No | Yes: $O(1)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | $O(n)$ | No |
| Bubble | Yes | Yes: $O(1)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | No |
| Opt. bubble | Yes | Yes: $O(1)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | Yes |
| Insertion | Yes | Yes: $O(1)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | Yes |
| Merge | Yes | No: $O(n)$ | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | $O(n\log n)$ | No |

## Merge sort

- Divide the array in half & sort them recursively

## Quick sort

- Divide it based on a pivot & sort them recursively

# In This Lecture

## Advanced sorting algorithms

- Quick Sort – Part II
  - How to partition an array based on the selected pivot
  - Analyze the properties of quick sort

- Heap Sort
  - Concept of heap sort
  - Efficient way to build a heap

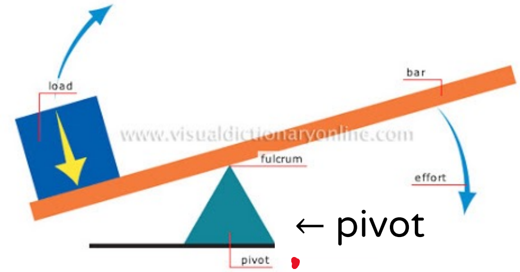
# Outline

## Advanced sorting algorithms

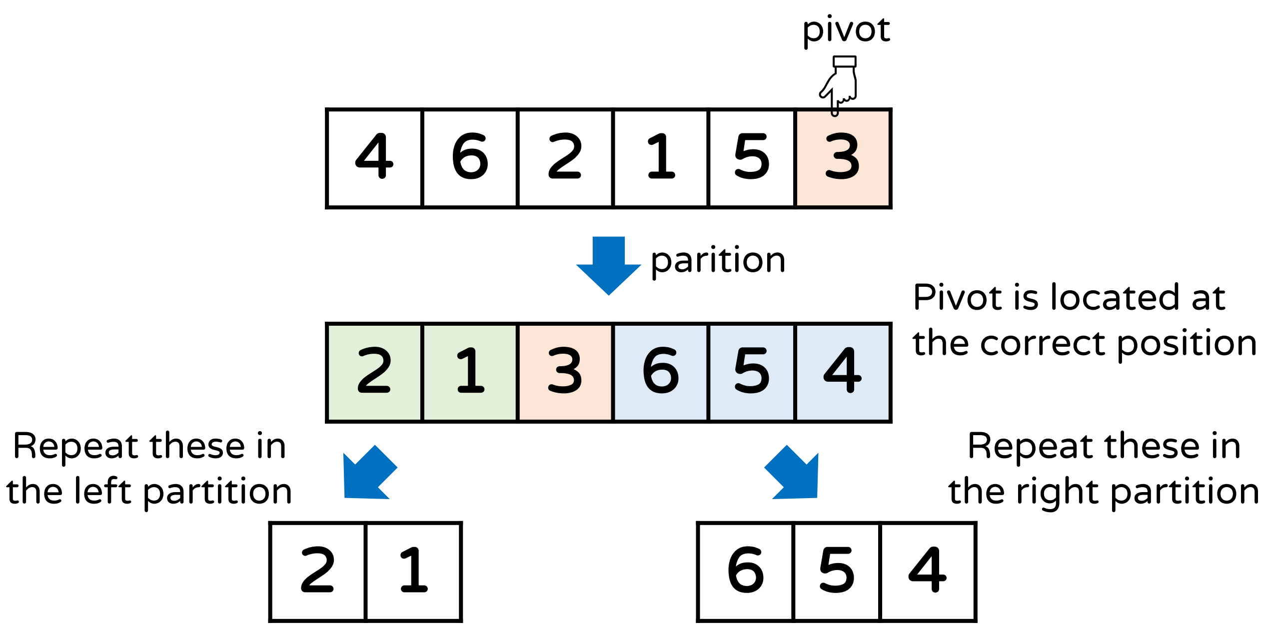
- **Quick Sort – Part II**
- Heap Sort

# Quick Sort (1)

## Idea (based on divide & conquer)




- [**Divide**] select a pivot & partition the input array based on the pivot satisfying the following condition
  - Elements before pivot ≤ pivot ≤ elements after pivot
- [**Conquer**] recursively sort left & right partitions, resp.



# Quick Sort (2)

## Pseudocode of quick sort

```
def quick_sort(A, l, r):
    if l < r:
        # [Divide] pick a pivot and partition A by the pivot
        p ← partition(A, l, r) # p is the pivot's index

        # [Conquer] recursively sort the left and right
        quick_sort(A, l, p - 1)
        quick_sort(A, p + 1, r)
```

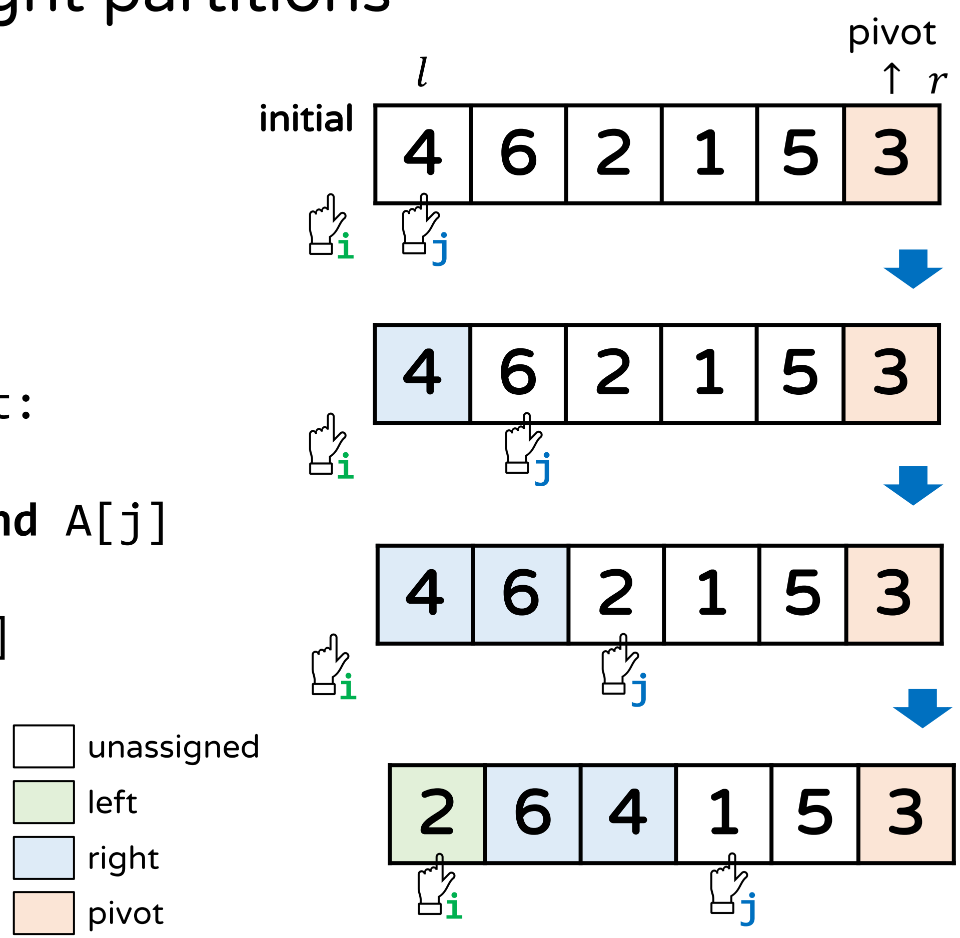
- How to pick the pivot and partition the input?
  - Any method can be possible if it is
    - Efficient (i.e., it should be done in $O(n)$ when partitioning an array of size $n$)
  - There are various methods for this; we'll learn about "**Lomuto partition scheme**" in this lecture

# Quick Sort (3)

## Lomuto partition

- Idea: pick the last element as a pivot & incrementally increase left and right partitions

```
def partition(A, l, r):
    pivot ← A[r]
    i ← l - 1
    for j ← l to r - 1:
        if A[j] <= pivot:
            i ← i + 1
            swap A[i] and A[j]

    swap A[i+1] and A[r]

    return i + 1
```

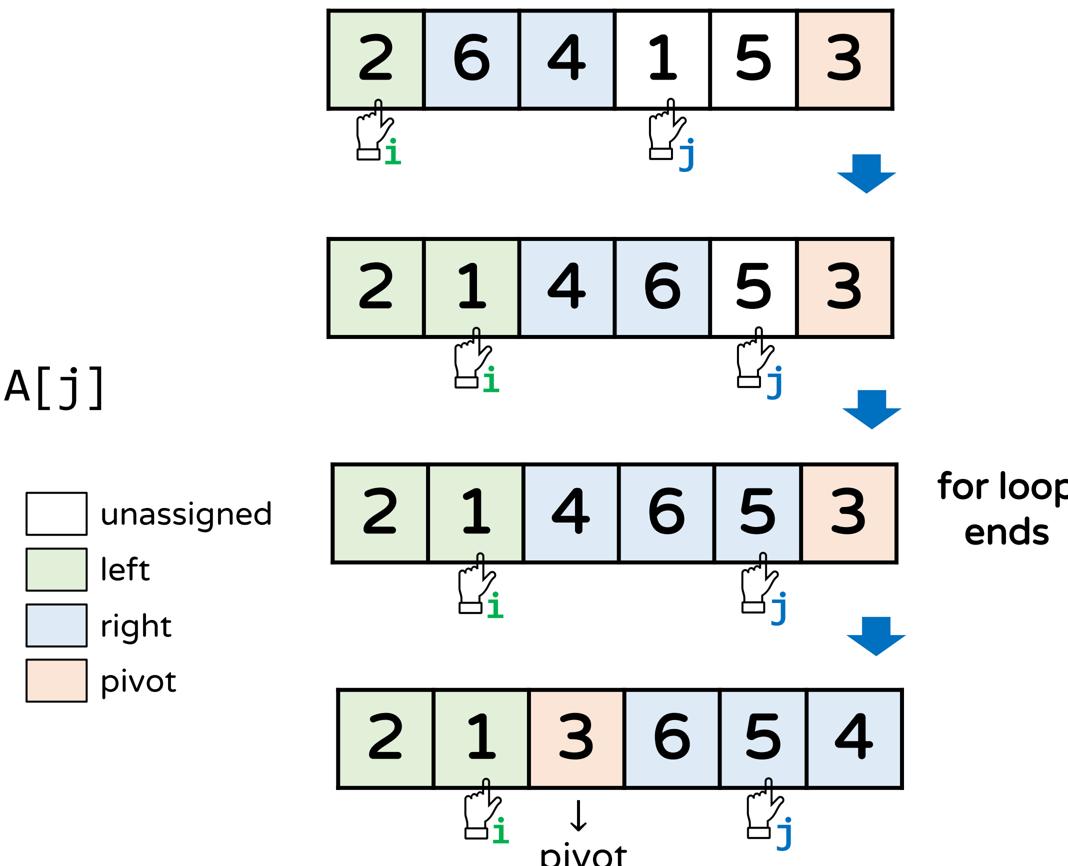


# Quick Sort (4)

## Lomuto partition

- Idea: pick the last element as a pivot & incrementally increase left and right partitions

```
def partition(A, l, r):
    pivot ← A[r]
    i ← l - 1
    for j ← l to r - 1:
        if A[j] <= pivot:
            i ← i + 1
            swap A[i] and A[j]

    swap A[i+1] and A[r]

    return i + 1
```




Let $n$ be $r - l + 1$.
⇒ # of comparisons is $n - 1$
⇒ # of swaps is $n$ at most

# Quick Sort (5)

## Correctness of quick sort

```
def quick_sort(A, l, r):
    if l < r:
        p ← partition(A, l, r) # Lomuto
        quick_sort(A, l, p - 1)
        quick_sort(A, p + 1, r)
```

## Proof by induction

- **Base case**: when the size is $1$, it is sorted by itself
- **Inductive step**
  - Assume quick_sort correctly sorts an array of any size less than $k$
  - Q. Does quick_sort sort an array of size $k$ correctly?
    - The partition() will correctly partition the input into two left and right partitions
    - The size of each partition is less than $k$
    - By the hypothesis, each partitions will be correctly sorted
    - Thus, all entries are sorted correctly

# Quick Sort (6)

## Time complexity of quick sort

- Let $T(n)$ be the time complexity of `quick_sort(A, 1, n)`
  - partition() takes $O(n)$ time when its input size is $n$
- **Best case** for partitioning
  - When each selected pivot splits its input in half

[rough] $$T(n) = 2T\left(\frac{n}{2}\right) + Cn \Rightarrow \Theta(n \log n)$$

- **Worst case** for partitioning
  - When each selected pivot makes one of partition empty
    - Consider an input array is already sorted in the ascending order

[rough] $$T(n) = T(n-1) + Cn \Rightarrow \Theta(n^2)$$

    - However, this is very unlikely to happen in real-world (suppose an array is randomly sorted)

# Quick Sort (7)

## Average time complexity of quick sort

- Suppose the pivot's index is $i$
  - The size of left partition: $(i-1)-1+1=i-1$
  - The size of right partition: $n-(i+1)+1=n-i$

$$T_i(n)=T(i-1)+T(n-i)+Cn$$

- Take the average of $T_i(n)$ over all $i\in[1,n]$
  - The probability that the pivot's index is $i$ is uniform

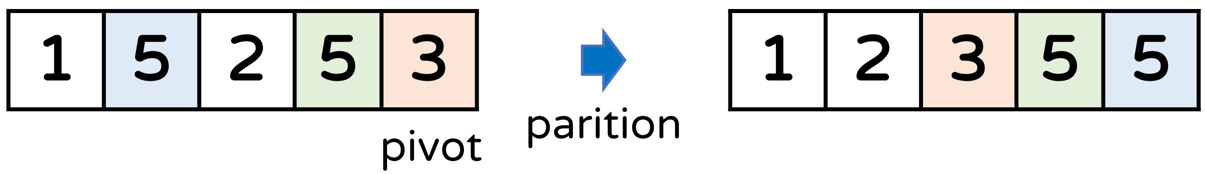
$$T(n)=\frac{1}{n}\sum_{i=1}^{n}T_i(n)=\frac{1}{n}\left(\sum_{i=1}^{n}T(i-1)+T(n-i)\right)+Cn$$

$$=\frac{2}{n}\sum_{k=0}^{n-1}T(k)+Cn\in O(n\log n)$$ ← proved by induction (108p)

# Quick Sort (8)

## Stability of merge sort

- Quick sort is not stable (see the below counter-example)
  - Lomuto partition is not stable




## Adaptivity of quick sort

- Quick sort is not adaptive due to the partition function
  - i.e., its comparisons are always performed

# Quick Sort (9)

```
def quick_sort(A, l, r):
    if l < r:
        p ← partition(A, l, r)
        quick_sort(A, l, p - 1)
        quick_sort(A, p + 1, r)
```

## Space complexity of quick sort

- $S(n) = \Theta(n)$
  - $O(n)$ is required to store $n$ input data
  - $O(log\, n)$ is required for extra space
- **Extra space:** $\Theta(n)$ for worst and $\Theta(\log n)$ for best cases
  - Required for maintaining recursive calls in a system stack
    - Note that the Lomuto partition is operated in-place, i.e., $\Theta(1)$
  - **Best case**: When each selected pivot splits its input in half
    - The recursion tree is fully binarized $\Rightarrow$ its maximum height is $O(\log n)$
  - **Worst case**: When each selected pivot makes one of partition empty
    - The recursion tree is degenerate $\Rightarrow$ its maximum height is $O(n)$
    - Can be optimized to $O(\log n)$ (see Appendix)

# Quick Sort (10)

## Is quick sort in-place?

- The answer depends on a criteria since it requires $O(\log n)$ extra space

## Yes [Wikipedia, CLRS] << including this lecture

- Because it sorts the elements within the array with a most a nearly constant amount of them outside the array
  - If $n = 10^{30}$, $\log 10^{30} = 30$ is very small compared to $n$
  - Thus, it can be considered as a constant; it's negligible for $n$

## No [By the strict definition of in-place alg.]

- Because an in-place algorithm should have $O(1)$ space

# Summary

| Name | Stable | In-place : Extra memory | # of comparisons | | # of swaps | | Adaptive |
|---|---|---|---|---|---|---|---|
| | | | Best | Worst | Best | Worst | |
| Selection | No | Yes: $O(1)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | $O(n)$ | No |
| Bubble | Yes | Yes: $O(1)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | No |
| Opt. bubble | Yes | Yes: $O(1)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | Yes |
| Insertion | Yes | Yes: $O(1)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | Yes |
| Merge | Yes | No: $O(n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | No |
| Quick | No | Yes: $O(\log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ | $O(n^2)$ | No |

## Remarks

- A swap operation is replaced with
  - A backward movement in insertion sort
  - A movement to temp array in merge sort
- Extra memory of quick sort is estimated by its optimized version
- No ideal answer in the above algorithms

# Outline

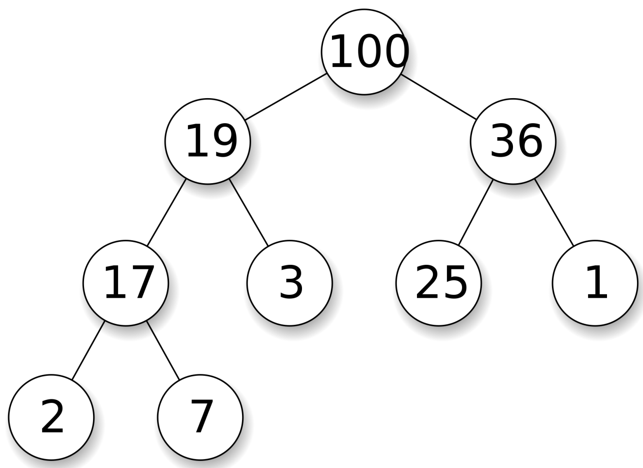
## Advanced sorting algorithms

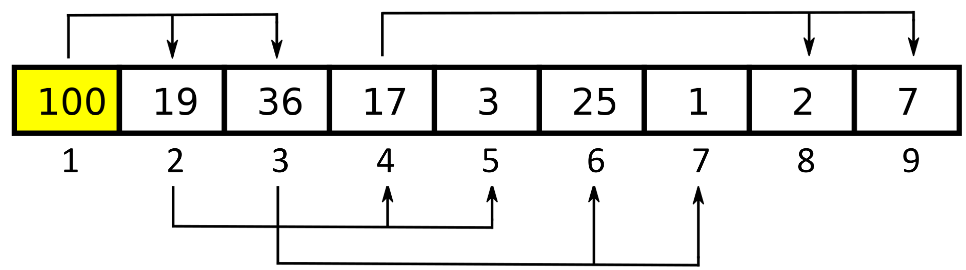
- Quick Sort – Part II
- **Heap Sort**

# Heap Sort (1)

## Sort an input using a heap!

- **Heap is a complete binary tree satisfying heap property**
  - Max-heap property: $\text{key(parent node)} \geq \text{key(children node)}$



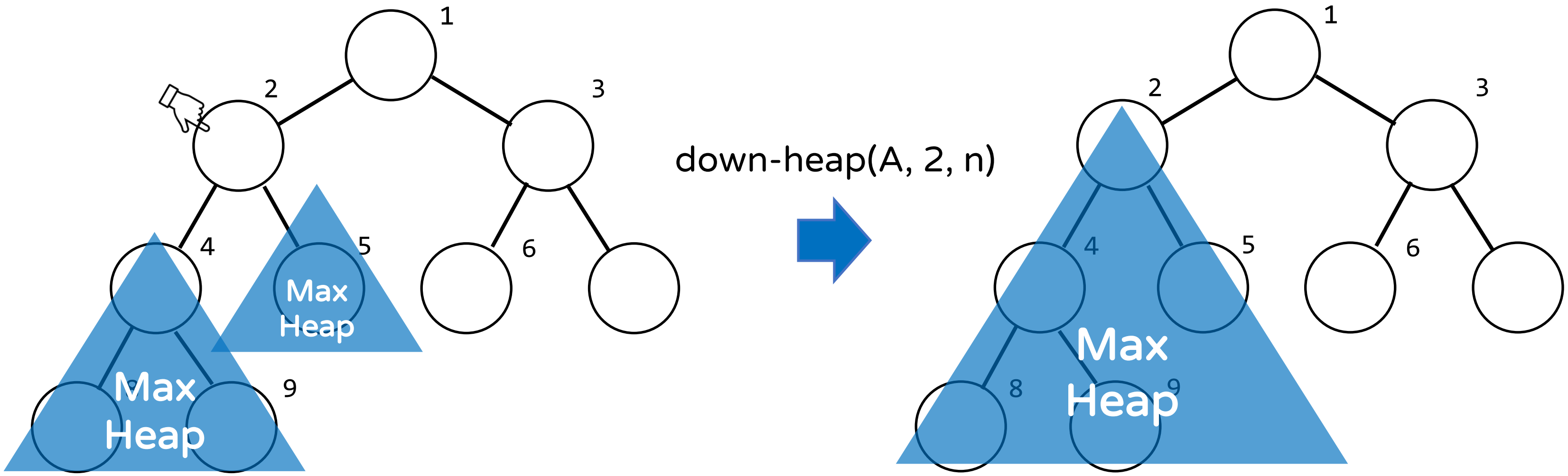




- **Main idea of heap sort**
  - **Step 1)** Build a heap from the input
  - **Step 2)** For each loop, extract the max from the heap & place it into the front of sorted area

# Heap Sort (2)

## down-heap (or heapify) operation

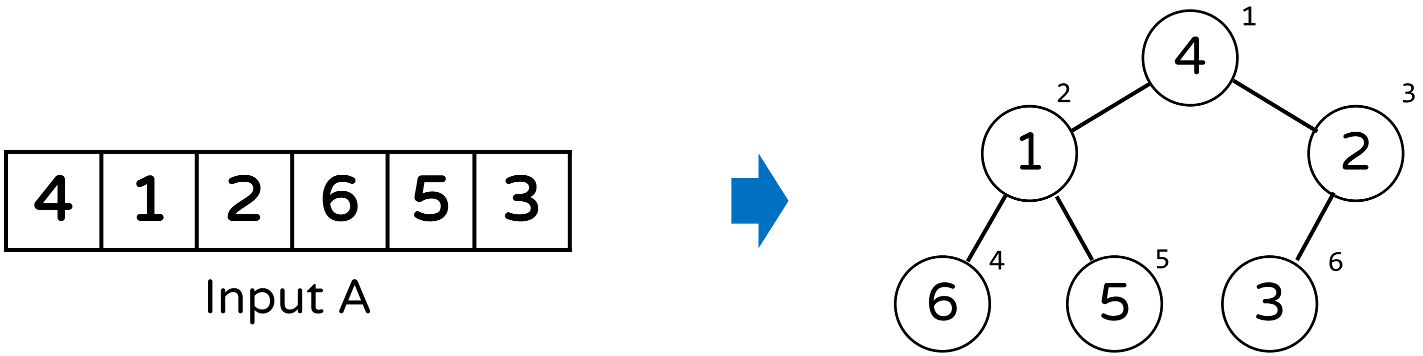
- Given index $i$, the **down-heap** operation heapifies the tree rooted at the index $i$
  - When the sub-trees of the root are heaps
  - Where $n$ is the size of the input array $A$



# Heap Sort (3)

## Step 1) Build a heap from the input

- Using down-heap() operation!



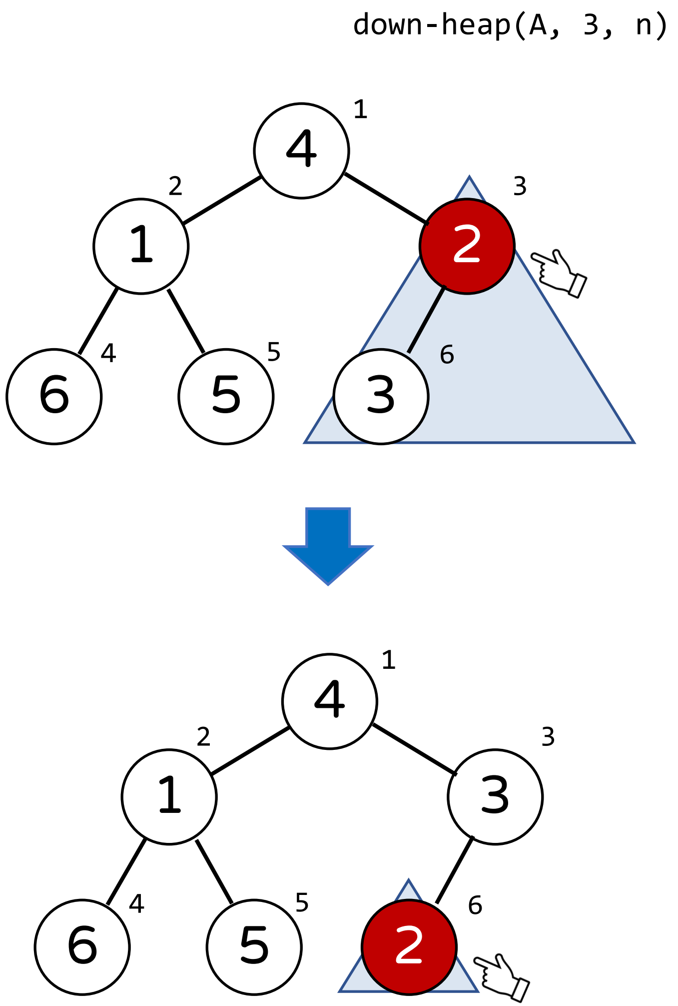

  - At the initial step, some of sub-trees do not satisfy the max-heap property
  - To push up a greater value to its upper level, performs down-heap() at each non-leaf nodes
    - From the bottom to the top & From the right to the left
    - Note that we don't need to do down-heap() for leaf nodes (i.e., start from index 3!)

# Heap Sort (4)

## Step 1) Build a heap from the input

- Using down-heap() operation!

```
def down-heap(A, i, n):
    left ← LEFT(i);
    right ← RIGHT(i);
    largest ← i;

    if left <= n and A[left] > A[largest]:
        largest ← left

    if right <= n and A[right] > A[largest]:
        largest ← right

    if largest != i:
        swap A[i] and A[largest];
        down-heap(largest);
```

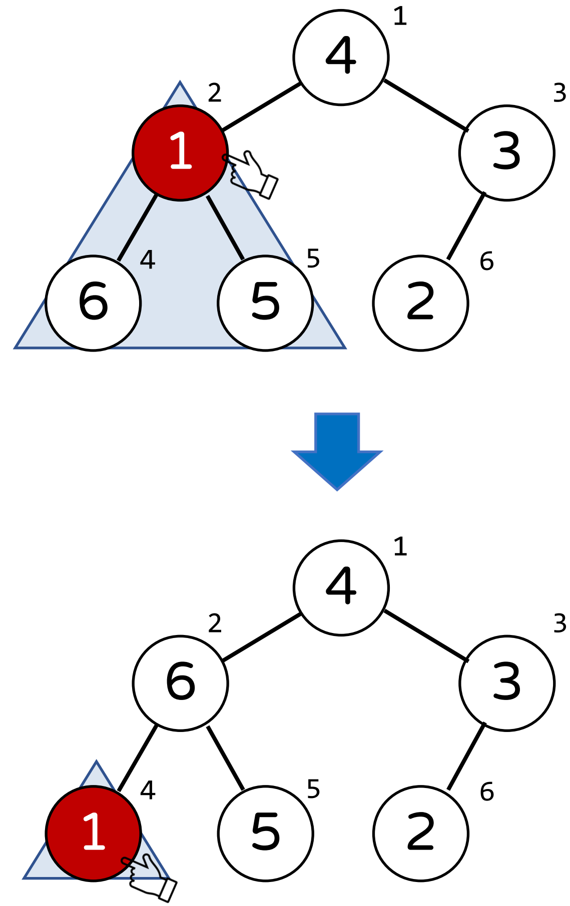


# Heap Sort (5)

## Step 1) Build a heap from the input

- Using down-heap() operation!

down-heap(A, 2, n)

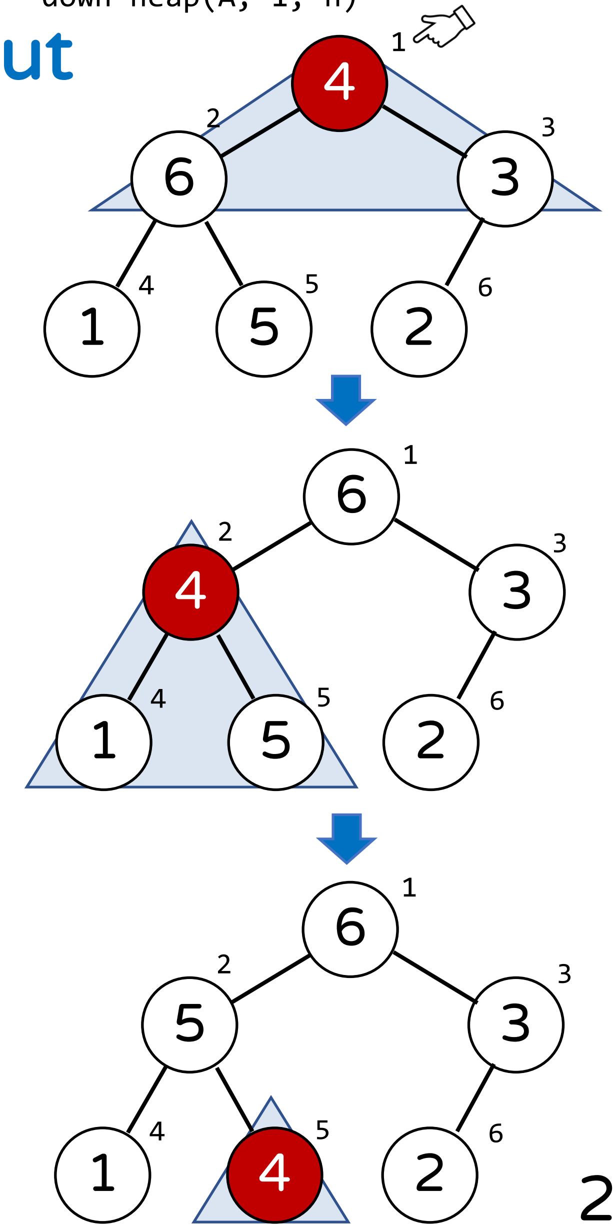



```
def down-heap(A, i, n):
    left ← LEFT(i);
    right ← RIGHT(i);
    largest ← i;

    if left <= n and A[left] > A[largest]:
        largest ← left

    if right <= n and A[right] > A[largest]:
        largest ← right

    if largest != i:
        swap A[i] and A[largest];
        down-heap(largest);
```

# Heap Sort (6)

## Step 1) Build a heap from the input

- Using down-heap() operation!




```
def down-heap(A, i, n):
    left ← LEFT(i);
    right ← RIGHT(i);
    largest ← i;

    if left <= n and A[left] > A[largest]:
        largest ← left

    if right <= n and A[right] > A[largest]:
        largest ← right

    if largest != i:
        swap A[i] and A[largest];
        down-heap(largest);
```

# Heap Sort (7)

## Step 1) Build a heap from the input

- **Using down-heap() operation!**

```
def build-heap(A, n):
    for i ← ⌊n/2⌋ downto 1:
        down-heap(A, i, n)
```

  - $\left\lfloor \frac{n}{2} \right\rfloor$ indicates the index of the last non-leaf (or internal) node

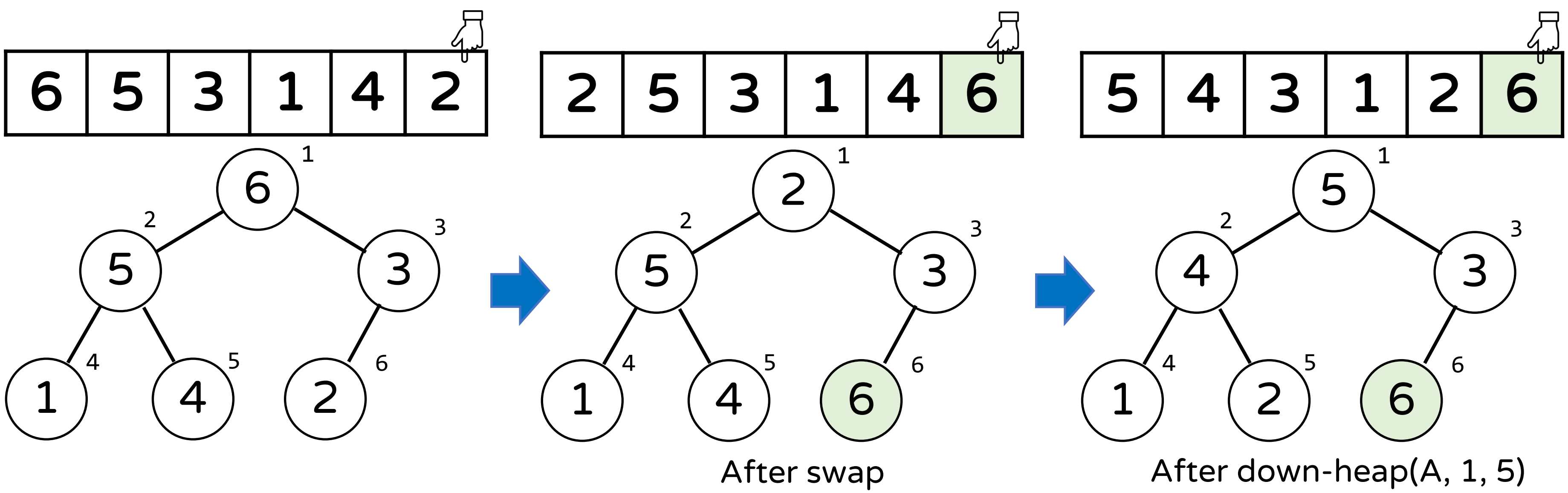
- Fact: $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \cdots, n$ are indices of leaf nodes
  - **Proof by contradiction**
    - Assume the fact is false (i.e., $i$ is the index of an internal node for $\left\lfloor \frac{n}{2} \right\rfloor + 1 \le i \le n$)
    - Then, $2i$ and $2i + 1$ are indices of node $i$'s left & right children
    - Then, $2\left\lfloor \frac{n}{2} \right\rfloor + 2 \le 2i \le 2n$; $2\left\lfloor \frac{n}{2} \right\rfloor + 2 > 2\left(\frac{n}{2} - 1\right) + 2 = n$
    - i.e., $n < 2\left\lfloor \frac{n}{2} \right\rfloor + 2 \le 2i \Rightarrow 2i$ is out-of-array ⇒ "the fact is false" is false

<<<Contradiction>>>

# Heap Sort (8)

**Step 2) For each loop, extract the max from the heap & place it into the front of sorted area**

```
def heap-sort(A, n):
    build-heap(A, n)
    for i ← n downto 2:
        swap A[i] and A[1]
        down-heap(A, 1, i-1)
```



# What You Need To Know

## Desired properties of a sorting algorithm

| Name | Stable | In-place : Extra memory | # of comparisons: Best | # of comparisons: Worst | # of swaps: Best | # of swaps: Worst | Adaptive |
|---|---|---|---|---|---|---|---|
| Selection | No | Yes: $O(1)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | $O(n)$ | No |
| Bubble | Yes | Yes: $O(1)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | No |
| Opt. bubble | Yes | Yes: $O(1)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | Yes |
| Insertion | Yes | Yes: $O(1)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | $O(n^2)$ | Yes |
| Merge | Yes | No: $O(n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | No |
| Quick | No | Yes: $O(\log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ | $O(n^2)$ | No |

## Quick sort

- Divide the input based on a pivot & sort them recursively
- Lomuto partition gives $O(n \log n)$ average time complexity

## Heap sort

- Build a heap from the input & repeatedly extract the max

# In Next Lecture

## Discussion on advanced sorting algorithms

- Including the analysis of heap sort

## Theoretical lower bound of comparison-based sorting algorithm

- Can we make a sorting algorithm faster than $\Omega(n \log n)$?

## Non-comparison based sorting algorithms

- Counting Sort
- Radix Sort

# Thank You

# [Appendx] Optimized Quick Sort

Space optimized version

```
def opt_quick_sort(A, l, r):
    while l < r:
        p ← partition(A, l, r) # Lomuto
        left_size = p – l
        right_size = r – p

        if left_size < right_size:
            opt_quick_sort(A, l, p - 1) # do the smaller first
            l = p + 1 # then repeat it on the right partition
        else:
            opt_quick_sort(A, p + 1, r) # do the smaller first
            r = p – 1 # then repeat it on the left partition
```

Then, the maximum height of the recursion tree is $O(\log n)$ for a worst case

- The smaller parition's size is at most half the size of the input parition