CSC10004: Data Structure and Algorithms

Lecture 5: Sorting

Lecturer: Bùi Văn Thạch
TA: Ngô Đình Hy/Lê Thị Thu Hiền
{bvthach,ndhy}@fit.hcmus.edu.vn, lththien@hcmus.edu.vn

Goals

1. To help students able to understand basic algorithms with run time $O(n^2)$ to faster algorithms with $O(n \log n)$ run time, where n is the number of elements in an array.

Last time: Complexity (Running time)

	Best	Worst	Average	Extra requirements
Linear search	$O(1)$	$O(n)$	$O(n)$	NO
Linear search with sentinel	$O(1)$	$O(n)$	$O(n)$	an extra slot at the end of the array for the sentinel
Binary search	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$	Input array needs to be SORTED beforehand
Randomized search	$O(1)$	$O(n^{0.9})$	$O\left(\frac{n}{m}\right)$	1. m is the number of keys in the array 2. Probabilistic method

Course topics

1. Introduction
2. Algorithm complexity analysis
3. Recurrences
4. Search
5. Sorting
6. Stack and Queue
7. Linked list
8. Priority queue
9. Tree
    1. Binary search tree (BST)
    2. AVL tree
11. Graph
    1. Graph representation
    2. Graph search
12. Hashing
13. Algorithm designs
    1. Greedy algorithm
    2. Divide-and-Conquer
    3. Dynamic programming

Complexity in **average**

		Design	**Run time**				**Space**	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	?	?	$O(\log n)$	$O(n)$		?	?
	Insertion sort	?	?				?	?
	Heap sort	?	?				?	?
	Merge sort	?	?				?	?
	Quick sort	?	?				?	?
Non-comparison-based	Radix sort	?	?				?	?

What are deterministic and randomized designs?

- Deterministic (Det.) design: Given the same input, a deterministic design produces the same result.

- Randomized (Rnd.) design: a randomized one might not produce the same result because it includes a degree of randomness in it.

What is “In-place”?

- An in-place sorting algorithm sorts an array without using extra space (or using only a small, constant amount of extra memory). It modifies the original array directly rather than creating a separate data structure.

- Uses only $O(1)$ or O(logn) extra space
- Modifies the original array instead of using extra storage
- Efficient for memory-constrained applications
- Often uses swapping or shifting to sort elements

What is "stable"?

- Stability in sorting means that equal elements maintain their relative order after sorting.

Example:

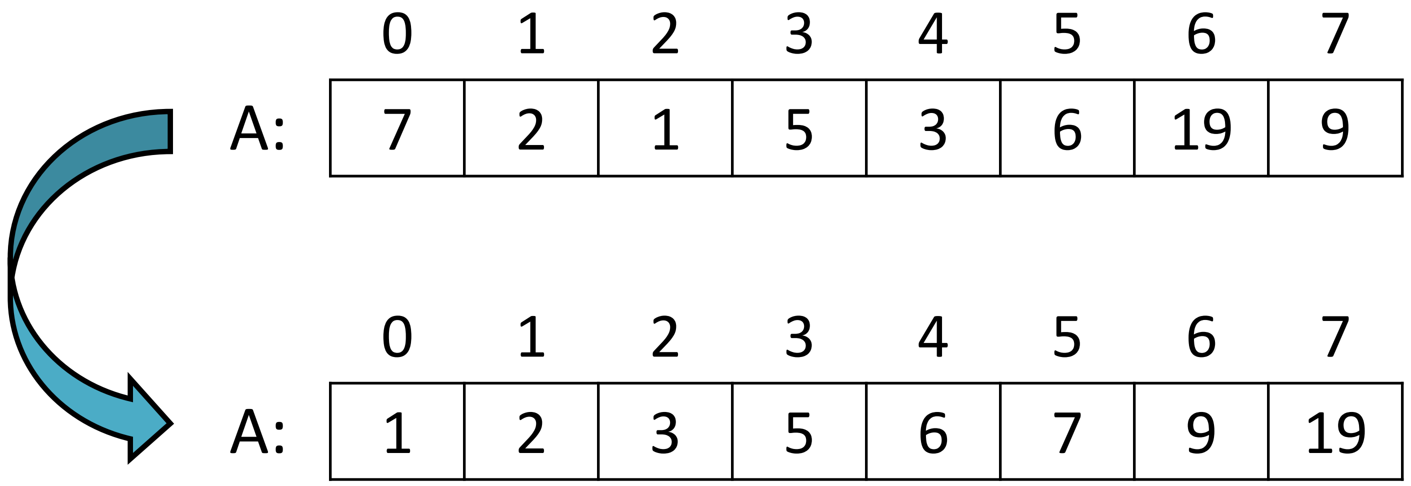
- Original Array (with indices for clarity)
- [(3, A), (1, B), (4, C), (3, D), (2, E)]
- (Key, Original Index)

After Sorting

- [(1, B), (2, E), (3, A), (3, D), (4, C)]
- The two (3, A) and (3, D) maintain their original order.

Sorting an Array

- Reorder the elements to put them in increasing order
  - Duplicate elements are allowed



	0	1	2	3	4	5	6	7
A:	7	2	1	5	3	6	19	9

	0	1	2	3	4	5	6	7
A:	1	2	3	5	6	7	9	19

- There are many algorithms to sort arrays
  - Let us start with a simple one, namely, ***selection sort***

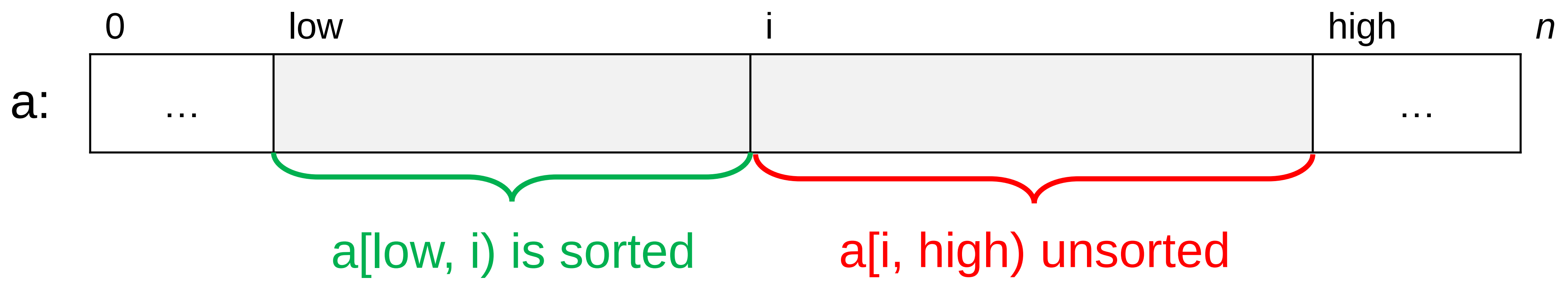
Outline

1. Selection Sort
2. Insertion Sort
3. Heap Sort
4. Merge Sort
5. Quick Sort
6. Radix Sort

Outline

1. Selection Sort
2. Insertion Sort
3. Heap Sort
4. Merge Sort
5. Quick Sort
6. Radix Sort

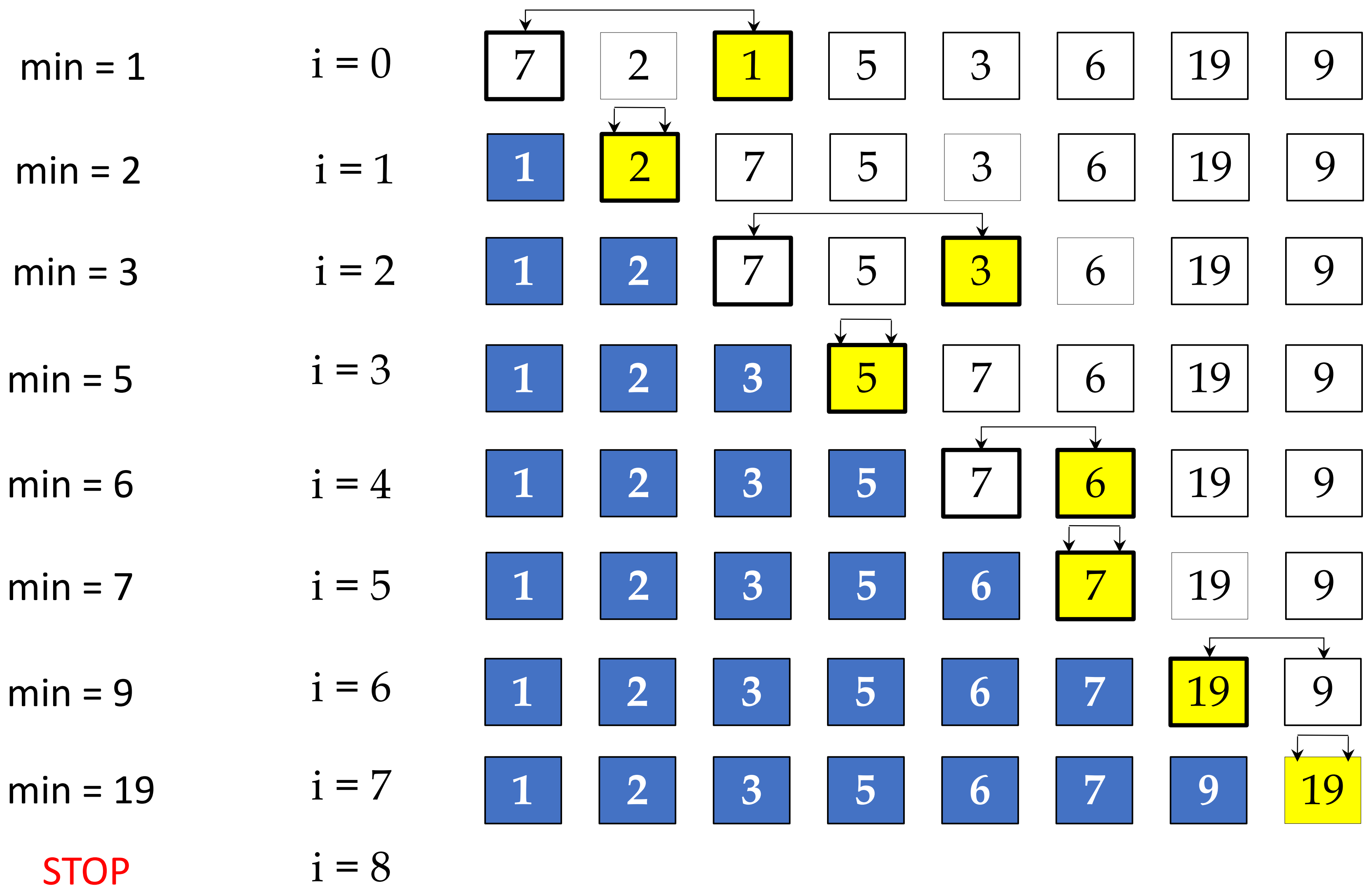
Idea



The main idea is to repeatedly find the smallest (or largest) element from the unsorted part of the array and swap it with the first unsorted element:

1. Find the smallest element in the unsorted part of the array.
2. Swap it with the first unsorted element.
3. Move the boundary between the sorted and unsorted parts one step forward.
4. Repeat until the array is fully sorted.

Example


min = 1
i = 0
7 2 1 5 3 6 19 9
min = 2
i = 1
1 2 7 5 3 6 19 9
min = 3
i = 2
1 2 7 5 3 6 19 9
min = 5
i = 3
1 2 3 5 7 6 19 9
min = 6
i = 4
1 2 3 5 7 6 19 9
min = 7
i = 5
1 2 3 5 6 7 19 9
min = 9
i = 6
1 2 3 5 6 7 19 9
min = 19
i = 7
1 2 3 5 6 7 9 19
STOP
i = 8

Source code

```
void SelectionSort(int[] a, int low, int n)
//require: 0 <= lo <= n and n <= len(a)
//ensures: a is sorted
{
  for (int i = lo; i < n; i++) {
    int min = find_min(a, i, n);
    swap(a, i, min);
  }
}
```

Stop when traversing the whole array

1. Find the **smallest value** a[min] in the list with index from *i* to *n*-1
2. Swap a[min] and a[i]

```
int find_min(int[] a, int low, int n) {
  int min = low;
  for (int i = low+1; i < n; i++) {
    if (a[i] < a[min])
      min = i;
  }
  return min;
}
```

```
void swap(int[] a, int i, int j) {
  int tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}
```

Analysis: Correctness (1/3)

```
1. void SelectionSort(int[] a, int low, int n)
2. //requires: 0 <= lo <= n and n <=
3. //ensures: a is sorted
4. {
5.   for (int i = lo; i < n; i++) {
6.     int min = find_min(a, i, n);
7.     swap(a, i, min);
8.   }
9. }
```

Stop when traversing the whole array

```
1. int find_min(int[] a, int low, in
2.   int min = low;
3.   for (int i = low+1; i < n; i++)
4.     if (a[i] < a[min])
5.       min = i;
6.   }
7.   return min;
8. }
```

1. Line 2: min = low.
2. Line 3: i runs from low+1 to n-1 which is not exceeded the index n-1 of a.
3. When i=low+1, when reaching Line 6, a[min] must be the smallest item compared with two items a[low] and a[low+1].
4. Line 3: when i increases by 1, if Line 4 holds, **a[i] must be the smallest item compared with items a[low], a[low+1], …, a[i].**
5. Since the index of the minimum value at step i is always recorded via parameter min, a[min] is the minimum value in the set {a[low], a[low+1], …, a[i]}.
6. Because the loop at Line 3 stops when i=n-1, i.e., *the whole array is scanned*, a[min] is the minimum value in the set {a[low], a[low+1], …, a[n-1]}.

Analysis: Correctness (2/3)

```
1. void SelectionSort(int[] a, int low, int n)
2. //require: 0 <= lo <= n and n <= len(a)
3. //ensures: a is sorted
4. {
5.   for (int i = lo; i < n; i++) {
6.     int min = find_min(a, i, n);
7.     swap(a, i, min);
8.   }
9. }
```

Stop when traversing the whole array

1. Find the **smallest value** a[min] in the list with index from *i* to *n*-1
2. Swap a[min] and a[i]

1. Line 2 holds the value of a[i] at tmp.
2. Line 3 assigns value of a[j] to a[i].
3. Line 4 assigns value at tmp, which is a[i], to a[j].
4. Therefore, a[i] and a[j] are swapped.

```
7.     return min;
8. }
```

```
1. void swap(int[] a, int i, int j) {
2.   int tmp = a[i];
3.   a[i] = a[j];
4.   a[j] = tmp;
5. }
```

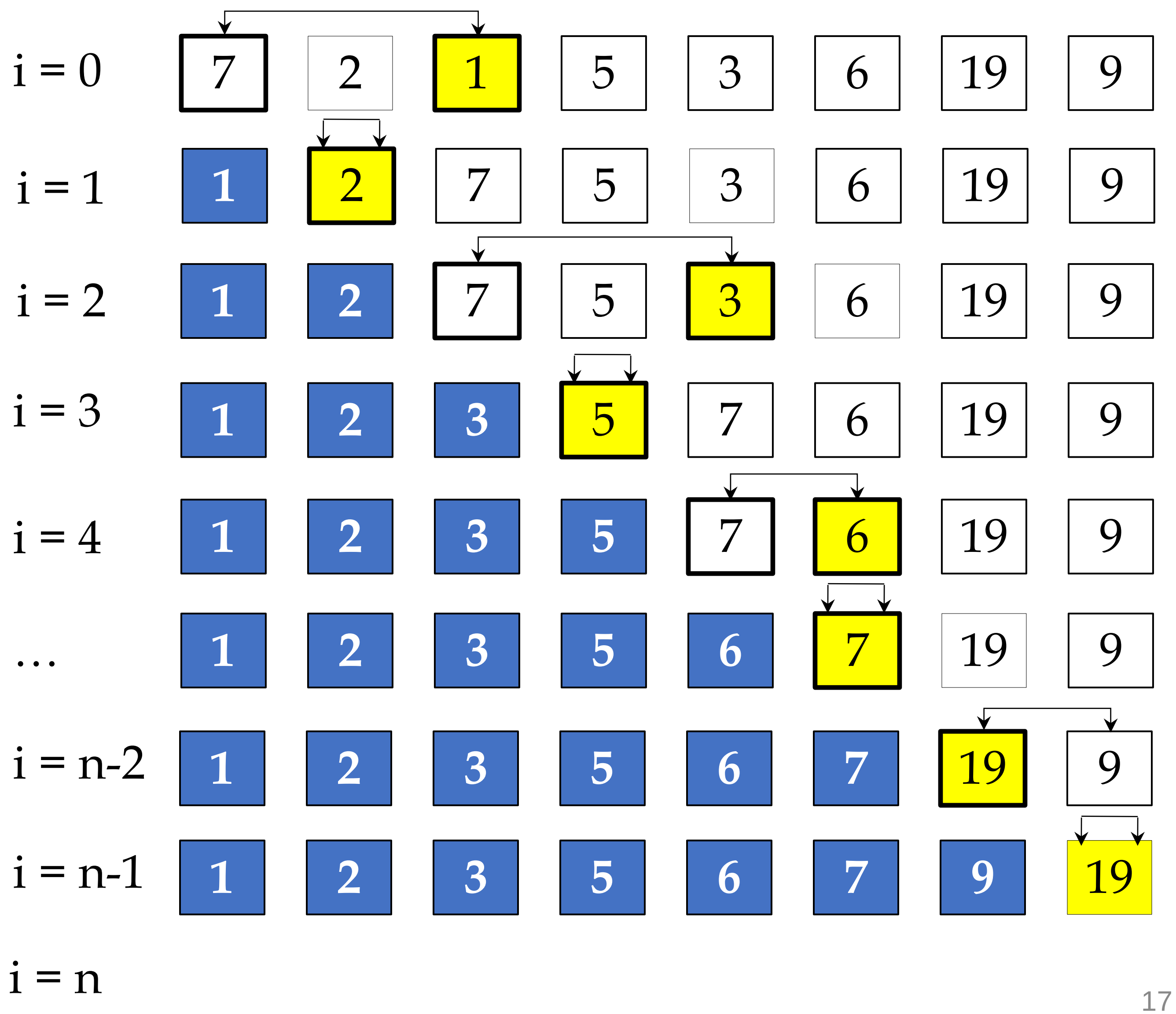
Analysis: Correctness (3/3)

a[min] is minimum in the set {a[0], …, a[n-1]}
Swap a[min] and a[0] ➔ a[0] to a[0] is sorted
a[min] is minimum in the set {a[1], …, a[n-1]}
Swap a[min] and a[1] ➔ a[0] to a[1] is sorted

```
void SelectionSort(int[] a, int low, int n)
//require: 0 <= lo <= n and n <= len(a)
//ensures: a is sorted
{
  for (int i = lo; i < n; i++) {
    int min = find_min(a, i, n);
    swap(a, i, min);
  }
}
```

a[min] is minimum in the set {a[n-2], a[n-1]}
Swap a[min] and a[n-2] ➔ a[0] to a[n-2] is sorted
a[min] is minimum in the set {a[n-1]}
Swap a[min] and a[n-1] ➔ a[0] to a[n-1] is sorted

STOP



Analysis: complexity (1/3)

```
void SelectionSort(int[] a, int low, int n)
//require: 0 <= lo <= n and n <= len(a)
//ensures: a is sorted
{
  for (int i = lo; i < n; i++) {
    int min = find_min(a, i, n);
    swap(a, i, min);
  }
}
```

n loops

? assignments + comparisons?

3 assigments

```
int find_min(int[] a, int low, int n) {
  int min = low;
  for (int i = low+1; i < n; i++) {
    if (a[i] < a[min])
      min = i;
  }
  return min;
}
```

```
void swap(int[] a, int i, int j) {
  int tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}
```

fit@hcmus

Analysis: complexity (3/3): putting all together

```
void SelectionSort(int[] a, int low, int n)
//require: 0 <= lo <= n and n <= len(a)
//ensures a is sorted
{
  for (int i = lo; i < n; i++) {
    int min = find_min(a, i, n);
    swap(a, i, min);
  }
}
```

n loops

3 assigments

```
int find_min(int[] a, int low, int n) {
  int min = low;
  for (int i = low+1; i < n; i++) {
    if (a[i] < a[min])
      min = i;
  }
  return min;
}
```

```
void swap(int[] a, int i, int j) {
  int tmp = a[i];
          ;
```

$2(n-low)$ comparisons
$n-low$ assignments
$\leq n-low$ assignments

$\geq 3(n-low)$ and $\leq 4(n-low)$, i.e.,
$\Theta(n-low)$ comparisons + assignments

fit@hcmus

Analysis: complexity (3/3): putting all together

```
void SelectionSort(int[] a, int low, int n)
//require: 0 <= lo <= n and n <= len(a)
//ensures a is sorted
{
  for (int i = lo; i < n; i++) {
    int min = find_min(a, i, n);
    swap(a, i,
  }
}
```

n loops

$$\leq \sum_{i=0}^{n-1}(4(n-i)+3) = \sum_{j=1}^{n}(4j+3) = 3n + 4\sum_{j=1}^{n} j = O(n^2)$$

```
int find_min(int[] a, int low, int n) {
  int min = lo
  for (int i =
    if (a[i] <
      min = i;
  }
  return min;
}
```

```
void swap(int[] a, int i, int j) {
```

$$\geq \sum_{i=0}^{n-1}(3(n-i)+3) = \sum_{j=1}^{n}(3j+3) = 3n + 3\sum_{j=1}^{n} j = \Omega(n^2)$$

$$\Theta(n^2)$$

$\geq 3(n-low)$ and $\leq 4(n-low)$, i.e.,
$\Theta(n-low)$ comparisons + assignments

Analysis: Pros and Cons

- Cons: If sorting a very large array, selection sort algorithm is inefficient to use.
- Pros: The behavior of selection sort algorithm does not depend on the initial order of data.

Complexity in **average**

		Design	**Run time**				**Space**	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$	$O(\log n)$	$O(n)$		Yes	No
	Insertion sort	?	?				?	?
	Heap sort	?	?				?	?
	Merge sort	?	?				?	?
	Quick sort	?	?				?	?
Non-comparison-based	Radix sort	?	?				?	?

Exercises

Given the following list of integers:

a) 13, 4, 2, 7, 34, 1.

b) 1, 56, 3, 6, 9, 8, 30, 24, 17, 7.

1. Using the Selection Sort and Insertion Sort, demonstrate the steps to sort ASCENDENLY.
2. Using the Selection Sort and Insertion Sort, demonstrate the steps to sort DECENDENLY.
3. Applying the Heap Construction (heapify) algorithm, demonstrate the steps to create a max-heap from the above list.
4. Using the Merge Sort, demonstrate the steps to sort the following list of integers ASCENDENLY.

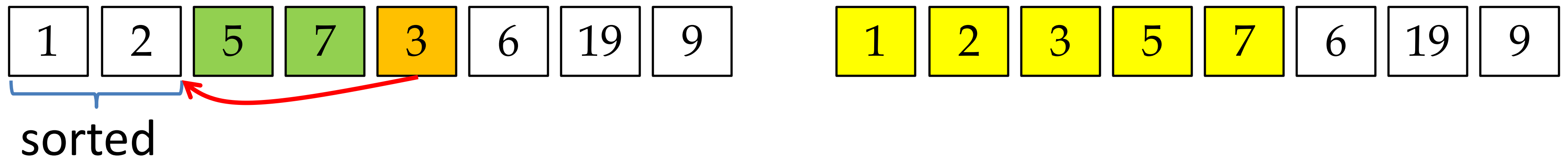
Outline

1. Selection Sort
2. Insertion Sort
3. Heap Sort
4. Merge Sort
5. Quick Sort
6. Radix Sort

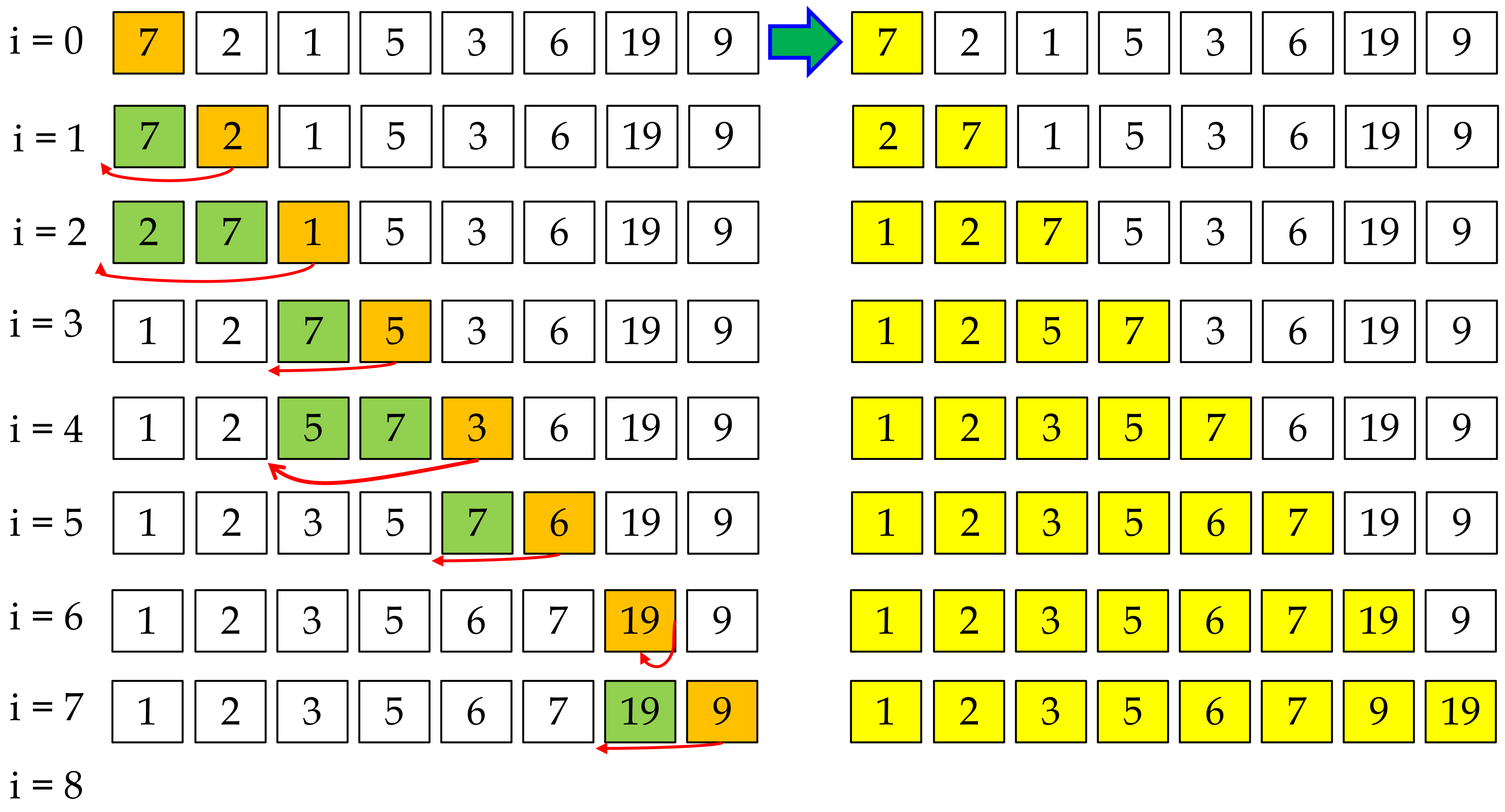
Idea

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time (it works similarly to how we sort playing cards in our hands).

1. Start with the second element (consider the first as sorted).
2. Compare it with elements in the sorted part (left side).
3. Shift larger elements to the right to create space.
4. Insert the element in the correct position.
5. Repeat for all elements until the array is sorted.



Example



	i = 0	7	2	1	5	3	6	19	9	7	2	1	5	3	6	19	9
	i = 1	7	2	1	5	3	6	19	9	2	7	1	5	3	6	19	9
	i = 2	2	7	1	5	3	6	19	9	1	2	7	5	3	6	19	9
	i = 3	1	2	7	5	3	6	19	9	1	2	5	7	3	6	19	9
	i = 4	1	2	5	7	3	6	19	9	1	2	3	5	7	6	19	9
	i = 5	1	2	3	5	7	6	19	9	1	2	3	5	6	7	19	9
	i = 6	1	2	3	5	6	7	19	9	1	2	3	5	6	7	19	9
	i = 7	1	2	3	5	6	7	19	9	1	2	3	5	6	7	9	19
STOP	i = 8																

Source code

```
void InsertionSort(int a[], int n)
{
    for (int i = 1; i < n; i++) {
        int key = a[i];
        int j = i - 1;
// Move elements of a[0..i-1] that are greater than key, to one position ahead of their current position
        while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = key;
    }
}
```

Analysis: Correctness

```
1. void InsertionSort(int a[], int n)
2. {
3.     for (int i = 1; i < n; i++) {
4.         int key = a[i];
5.         int j = i - 1;
6.         while (j >= 0 && a[j] > key) {
7.             a[j + 1] = a[j];
8.             j = j - 1;
9.         }
10.        a[j + 1] = key;
11.    }
12.}
```

Claim: At start of iteration i (Line 4): array a[0…i-1] is sorted.

- Suppose that the while loop (Lines 6-9) stop at j^*. Then:

1. Any element in a[0…j^*] is smaller or equal to a[i].
2. Any element in a[$j^* + 1$…i-1] is larger than a[i].

- The loop moves each element in a[$j^* + 1$…i-1] to one position ahead.
  - a[$j^* + 2$…i] is sorted and each element int it is greater than key=a[i] (note that a[i] is now replaced by a[i-1]).
- Line 10 assigns a[$j^* + 1$] = a[i];
- Therefore, array a[0…j^*] a[$j^* + 1$] a[$j^* + 2$…i] is sorted.
- When $i = n - 1$, array a is sorted.

Analysis: Complexity (1/2)

- Best case: $O(n)$, If the list is already sorted.
- Worst case: $O(n^2)$, if the list is in reverse order.

Analysis: Complexity (2/2): average case

```
1.  void InsertionSort(int a[], int n)
2.  {
3.      for (int i = 1; i < n; i++) {
4.          int key = a[i];
5.          int j = i - 1;
// Move elements of a[0..i-1] that are greater than key, to one
position ahead of their current position
6.          while (j >= 0 && a[j] > key) {
7.              a[j + 1] = a[j];
8.              j = j - 1;
9.          }
10.         a[j + 1] = key;
11.     }
12. }
```

1.
2. 0
3. $n-1$ loops with 1 comparison and 1 assignment each loop
4. 1 assignment
5. 1 assignment
6. $\leq i-1$ loops and 2 comparisons
7. 1 assignment
8. 1 assignment
9.
10. 1 assignment

$$\leq \sum_{i=1}^{n-1}(1+1+(i-1)(2+1+1)+1) = \sum_{i=1}^{n-1}(4i-1) = O(n^2)$$

Complexity in **average**

		Design	**Run time**				**Space**	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$	$O(\log n)$	$O(n)$		Yes	No
	Insertion sort	Det.	$O(n^2)$				Yes	Yes
	Heap sort	?	?				?	?
	Merge sort	?	?				?	?
	Quick sort	?	?				?	?
Non-comparison-based	Radix sort	?	?				?	?

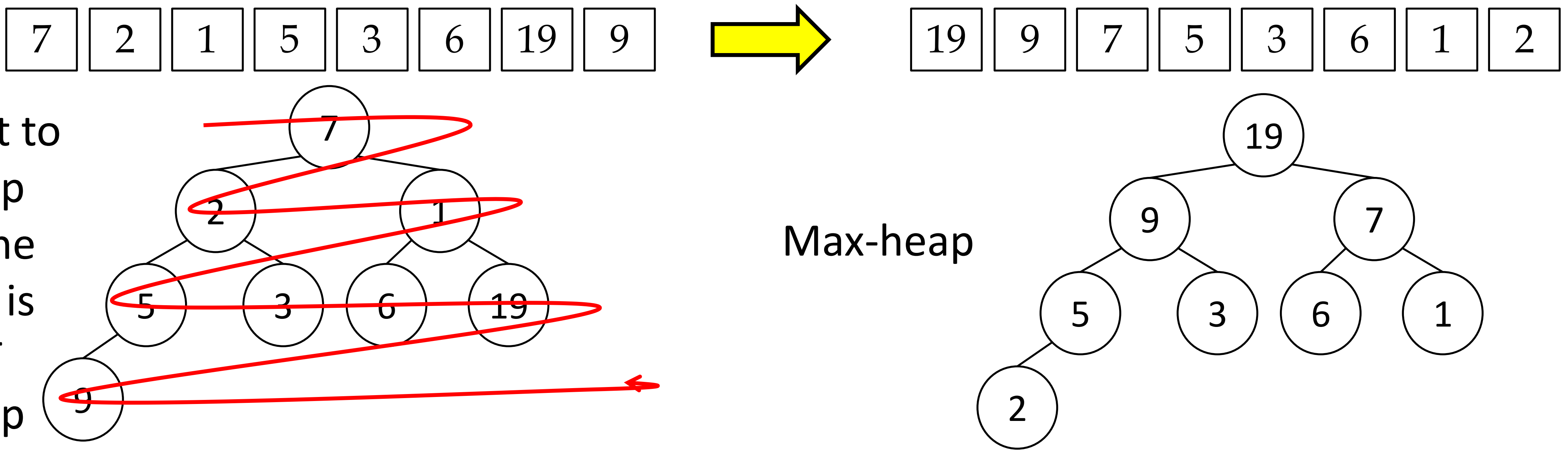
Outline

1. Selection Sort
2. Insertion Sort
3. Heap Sort
4. Merge Sort
5. Quick Sort
6. Radix Sort

Heap: definition

- Heap is a collection of *n* elements (a_0, a_1, …, a_{n-1}) in which
  - For every i ($0 \leq i \leq n/2 - 1$): $a_i \geq a_{2i+1}$ and $a_i \geq a_{2i+2}$
  - If $2i + 2 \geq n$, just $a_i \geq a_{2i+1}$ needs to hold
  - Condition does not apply to the second half as $2i + 1$ and $2i + 2$ are out of array.
- Heap in above definition is called **max-heap** *(we also have **min-heap**).*

Convert to a heap but the result is NOT a heap



Heap: properties

- Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest).
- Map from array elements to tree nodes and vice versa.
  - Root – a[0]
  - Left[i] – a[2i+1]
  - Right[i] – a[2i+2]
  - Parent[i] – a[(i-1)/2] if i is odd and a[i/2-1] if i is even.

Idea

HeapSort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements efficiently:

1. **Convert the array into a Max Heap** (a complete binary tree where each parent node is greater than its children).
2. **Extract the maximum element** (root) and place it at the end of the array.
3. **Heapify the remaining elements** to maintain the heap property.
4. **Repeat until the array is sorted**.

Source code (1/2): Function to perform Heap Sort

```
void heapSort(int arr[], int n) {
    // Step 1: Build a Max Heap
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Step 2: Extract elements one by one from the heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move the largest element to the end
        heapify(arr, i, 0);   // Heapify the reduced heap
    }
}
```

Source code (2/2): Function to heapify a subtree rooted at index i in an array of size n

```
void heapify(int arr[], int n, int i) {
    int largest = i;       // Assume root (i) is the largest
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than the largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        swap(arr[i], arr[largest]); // Swap root with the largest child
        heapify(arr, n, largest);  // Recursively heapify the affected subtree
    }
}
```

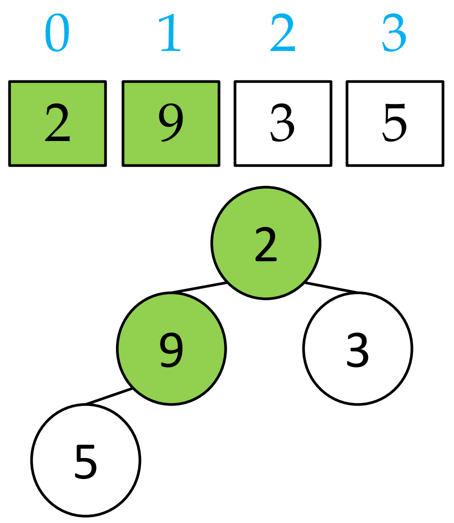
$\text{left} = 1$
$\text{right} = 2$

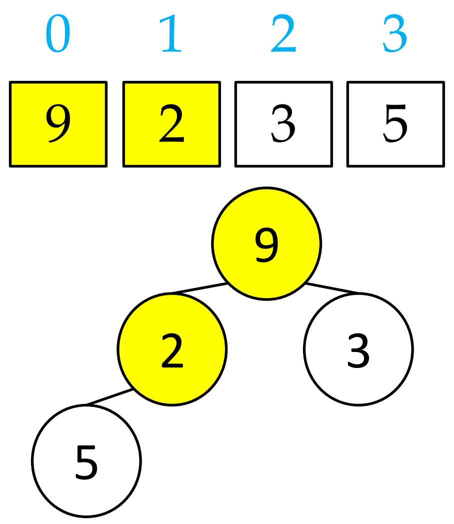
$n = 4, i = 0$

$\text{largest} = 0$

$\downarrow$

$\text{largest} = 1$

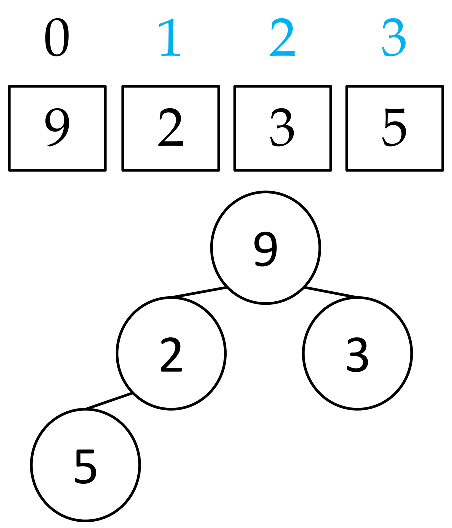




Source code (2/2): example of heapify (2/)

```
void heapify(int arr[], int n, int i) {
    int largest = i;      // Assume root (i) is the largest
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than the largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;
    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        swap(arr[i], arr[largest]); // Swap root with the largest child
        heapify(arr, n, largest);  // Recursively heapify the affected subtree
    }
}
```

$n = 4, i = 1$



Source code (2/2): example of heapify (3/)

```
1.  void heapify(int arr[], int n, int i) {
2.      int largest = i;      // Assume root (i) is the largest
3.      int left = 2 * i + 1; // Left child
4.      int right = 2 * i + 2; // Right child

5.      // If left child is larger than root
6.      if (left < n && arr[left] > arr[largest])
7.          largest = left;

8.      // If right child is larger than the largest so far
9.      if (right < n && arr[right] > arr[largest])
10.         largest = right;
11.     // If largest is not root, swap and continue heapifying
12.     if (largest != i) {
13.         swap(arr[i], arr[largest]); // Swap root with the largest child
14.         heapify(arr, n, largest);  // Recursively heapify the affected subtree
15.     }
16. }
```

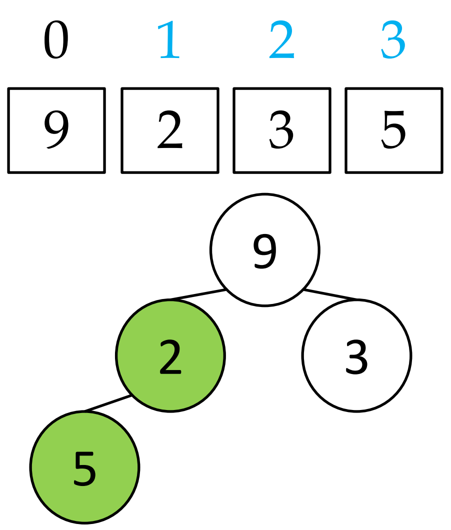
$\text{left} = 3$
$\text{right} = 4$

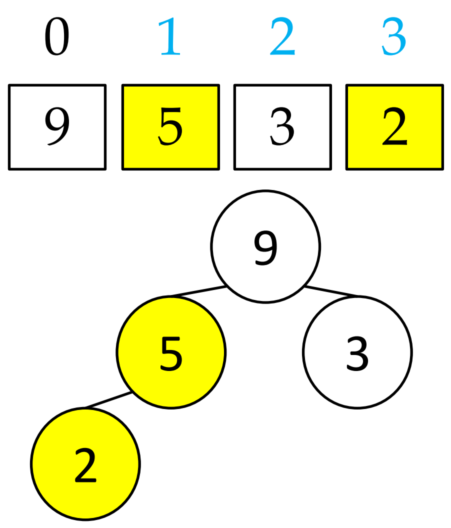
$n = 4, i = 1$

$\text{largest} = 1$

$\downarrow$

$\text{largest} = 3$

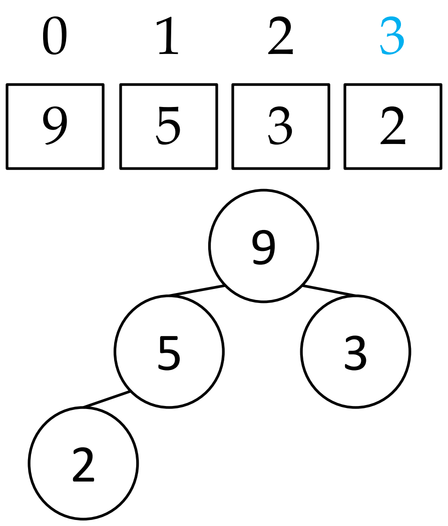




Source code (2/2): example of heapify (3/)

```
void heapify(int arr[], int n, int i) {
    int largest = i;      // Assume root (i) is the largest
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than the largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;
    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        swap(arr[i], arr[largest]); // Swap root with the largest child
        heapify(arr, n, largest);  // Recursively heapify the affected subtree
    }
}
```

$n = 4, i = 3$



fit@hcmus

Source code (2/2): example of heapify (4/)

```
1.  void heapify(int arr[], int n, int i) {
2.      int largest = i;      // Assume root (i) is the largest
3.      int left = 2 * i + 1; // Left child
4.      int right = 2 * i + 2; // Right child

5.      // If left child is larger than root
6.      if (left < n && arr[left] > arr[largest])
7.          largest = left;

8.      // If right child is larger than the largest so far
9.      if (right < n && arr[right] > arr[largest])
10.         largest = right;
11.     // If largest is not root, swap and continue heapifying
12.     if (largest != i) {
13.         swap(arr[i], arr[largest]); // Swap root with the largest child
14.         heapify(arr, n, largest);  // Recursively heapify the affected subtree
15.     }
16. }
```

$\text{left} = 7$
$\text{right} = 8$

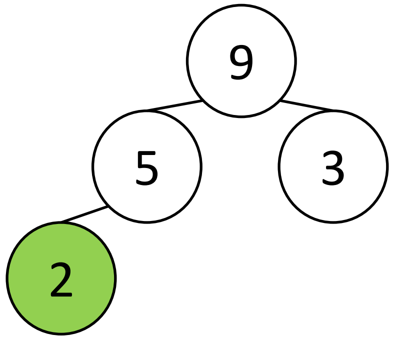
$n = 4, i = 3$

$\text{largest} = 3$

$\downarrow$

$\text{largest} = i$

0	1	2	3
9	5	3	2



STOP

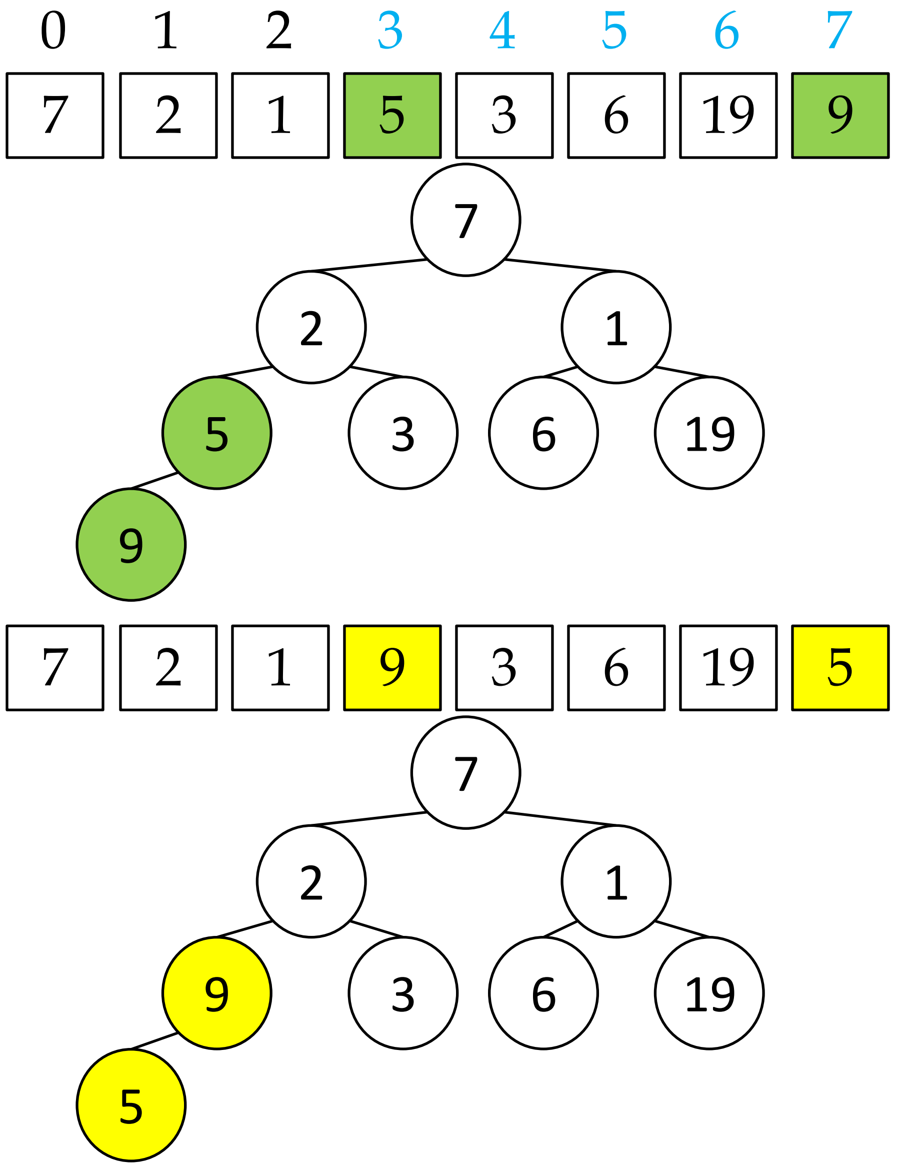
fit@hcmus

Example (1/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

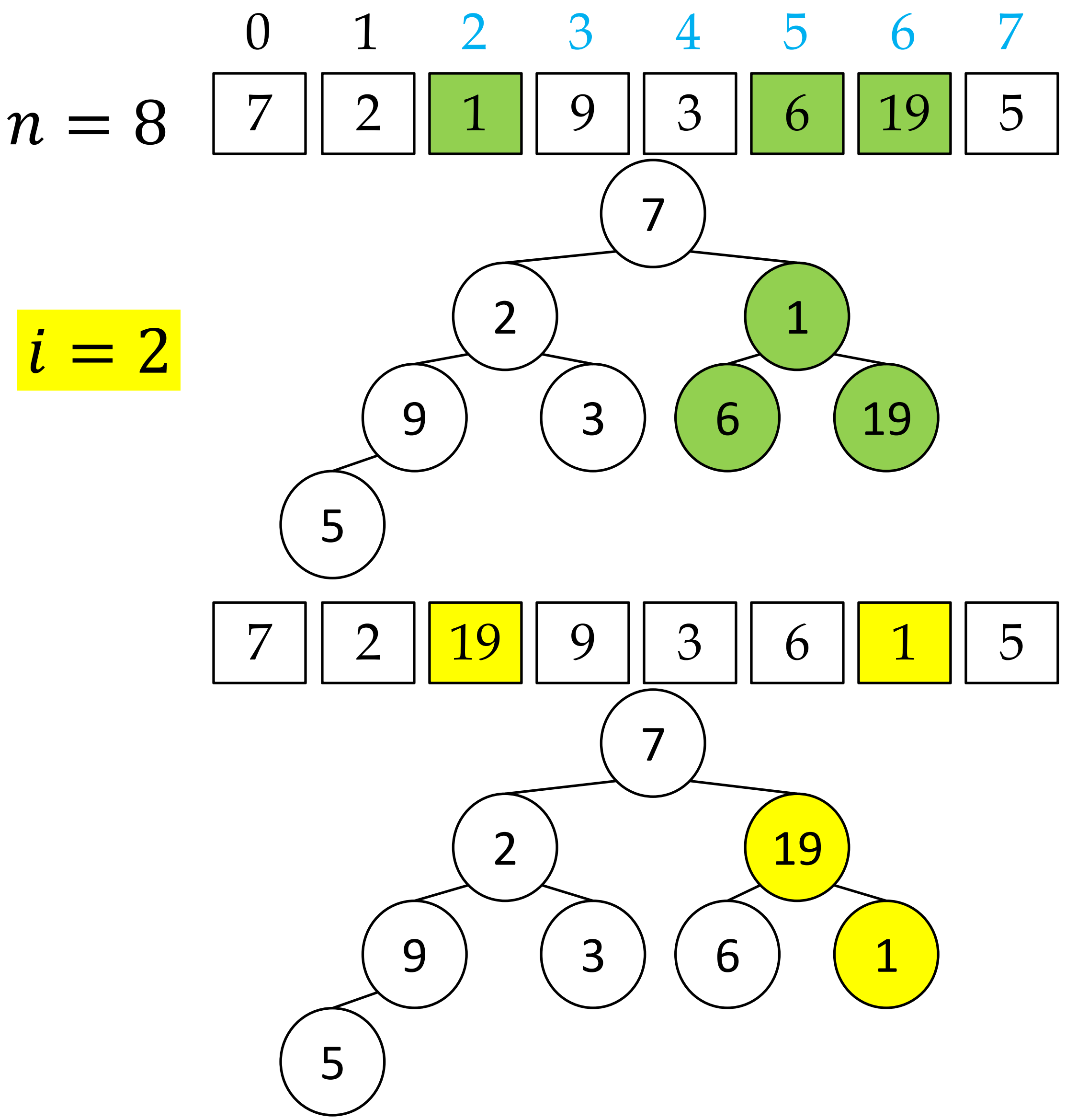
$n = 8$

$i = 3$



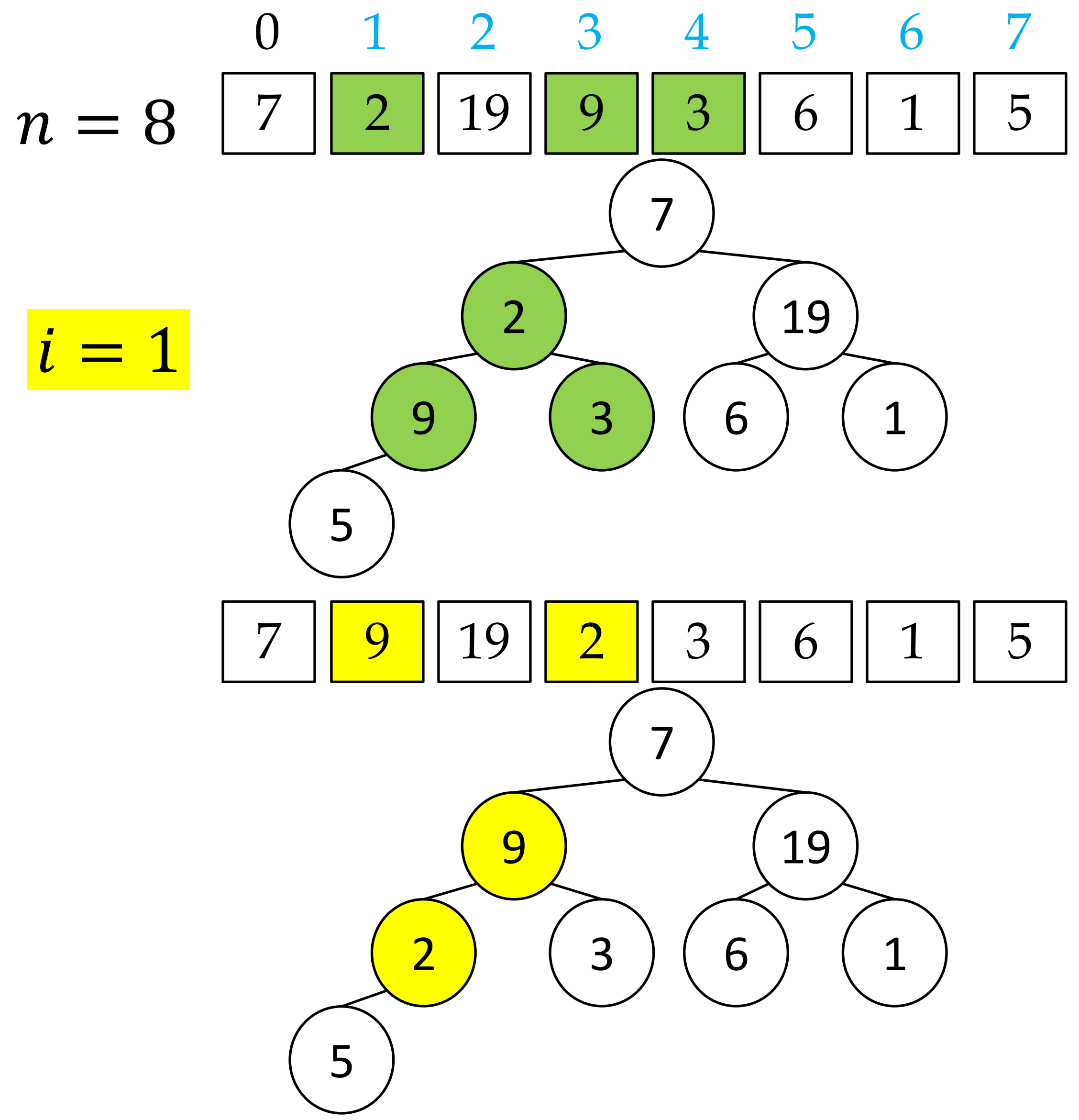
Example (2/19)

```
void heapSort(int arr[], int n) {
    // Step 1: Build a Max Heap
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Step 2: Extract elements one by one from the heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move the largest element to the end
        heapify(arr, i, 0);   // Heapify the reduced heap
    }
}
```



Example (3/19)

```
void heapSort(int arr[], int n) {
    // Step 1: Build a Max Heap
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Step 2: Extract elements one by one from the heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move the largest element to the end
        heapify(arr, i, 0);   // Heapify the reduced heap
    }
}
```



Example (4/19)

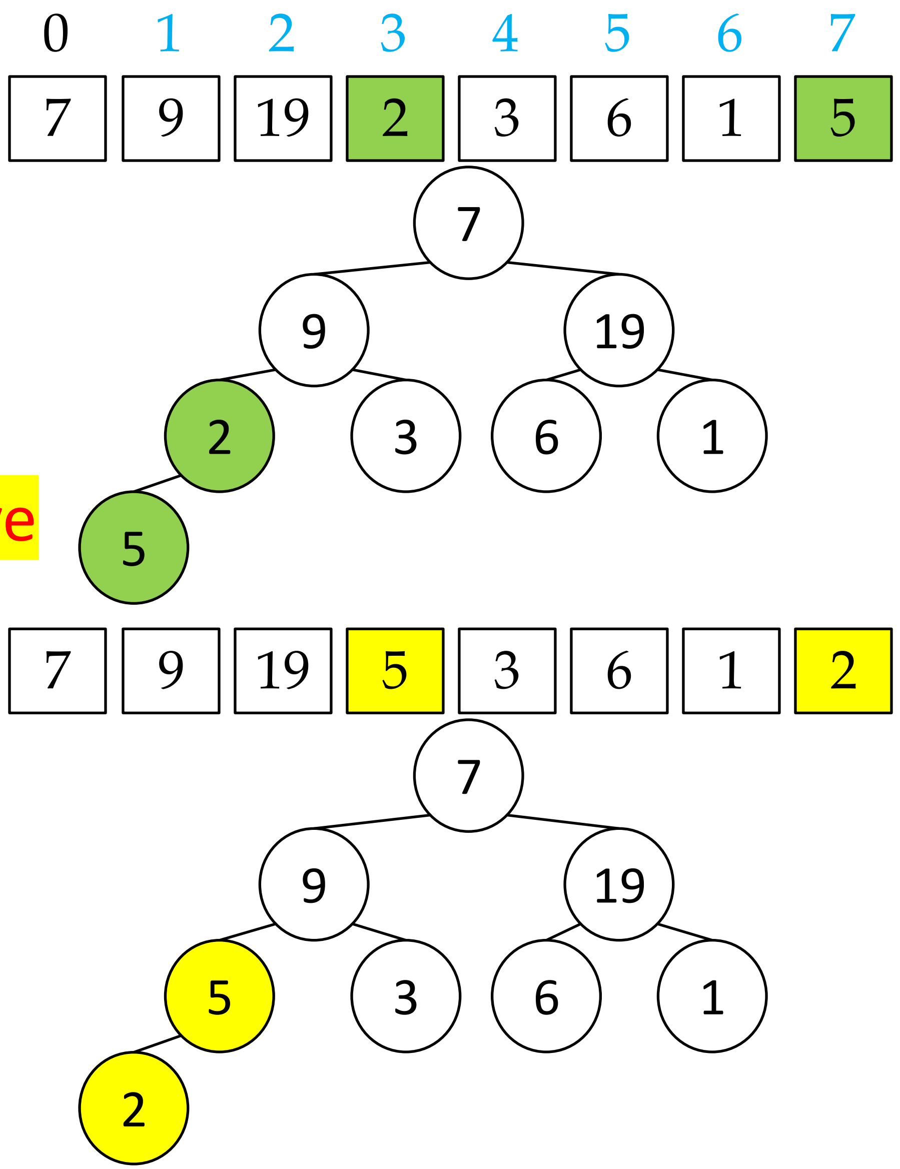
1. void heapSort(int arr[], int n) {
2. // Step 1: Build a Max Heap
3. for (int i = n/2 - 1; i >= 0; i--)
4. heapify(arr, n, i);

5. // Step 2: Extract elements one by one from the heap
6. for (int i = n - 1; i > 0; i--) {
7. swap(arr[0], arr[i]); // Move the largest element to the end
8. heapify(arr, i, 0); // Heapify the reduced heap
9. }
10. }

$n = 8$

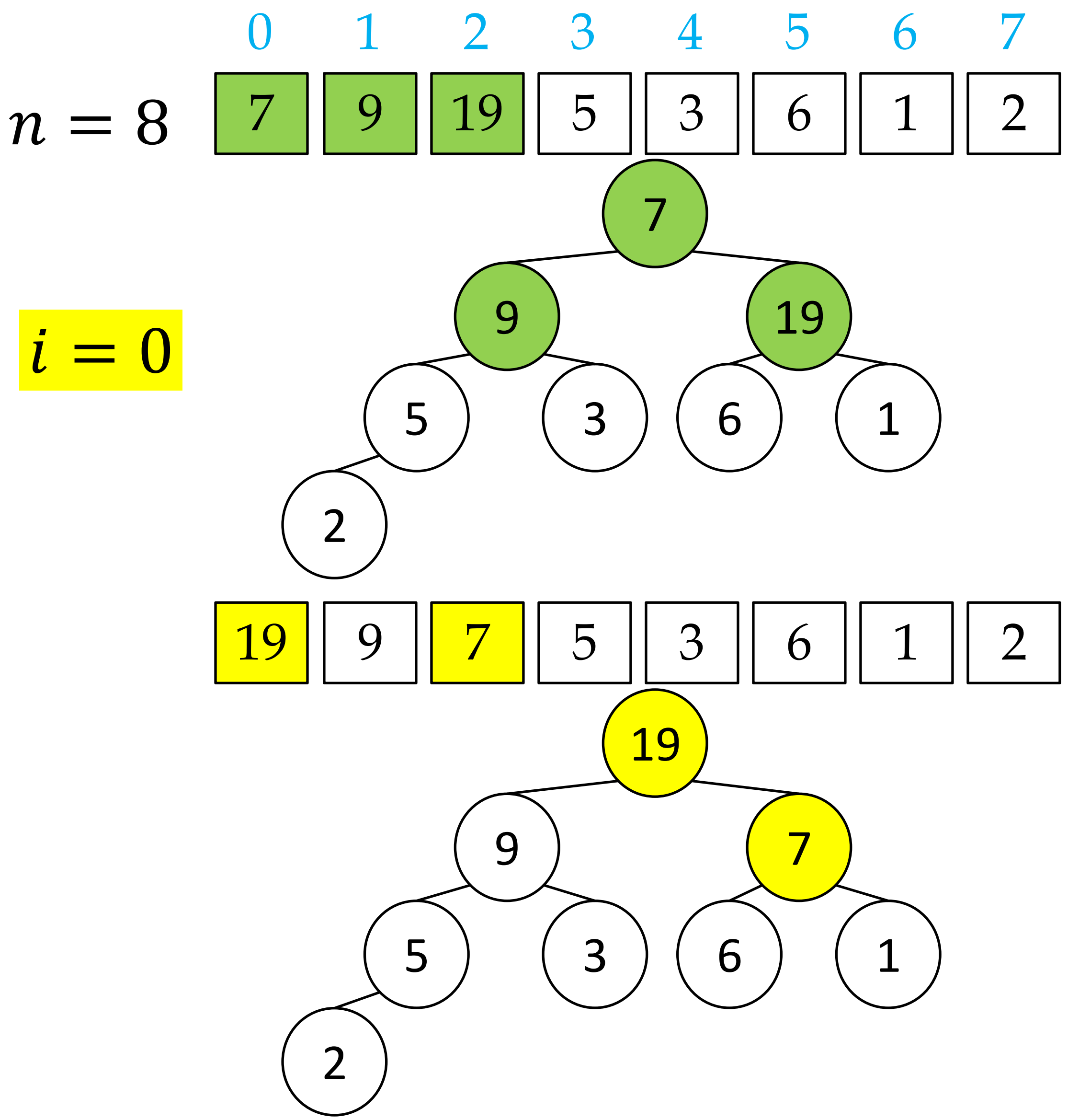
$i = 1$

Recursive



Example (5/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```



fit@hcmus

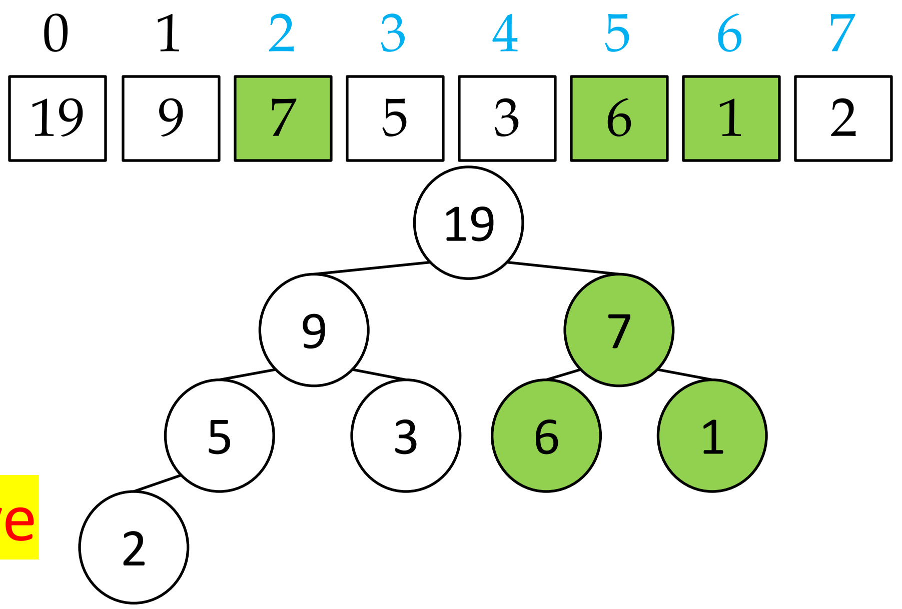
Example (5/19)

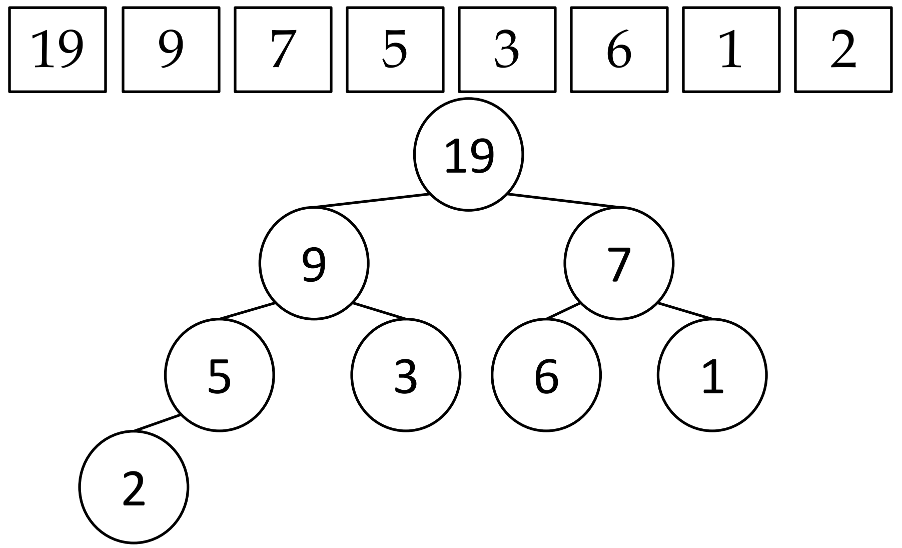
```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

$n = 8$

$i = 0$

Recursive





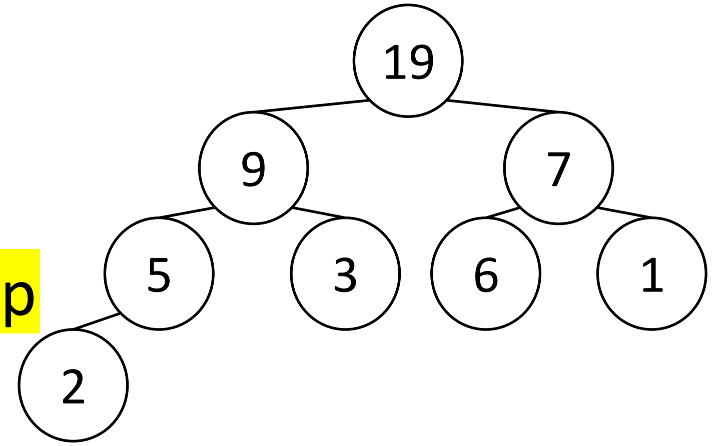
Example (5/19)

```
void heapSort(int arr[], int n) {
    // Step 1: Build a Max Heap
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Step 2: Extract elements one by one from the heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move the largest element to the end
        heapify(arr, i, 0);   // Heapify the reduced heap
    }
}
```

$n = 8$

0	1	2	3	4	5	6	7
19	9	7	5	3	6	1	2

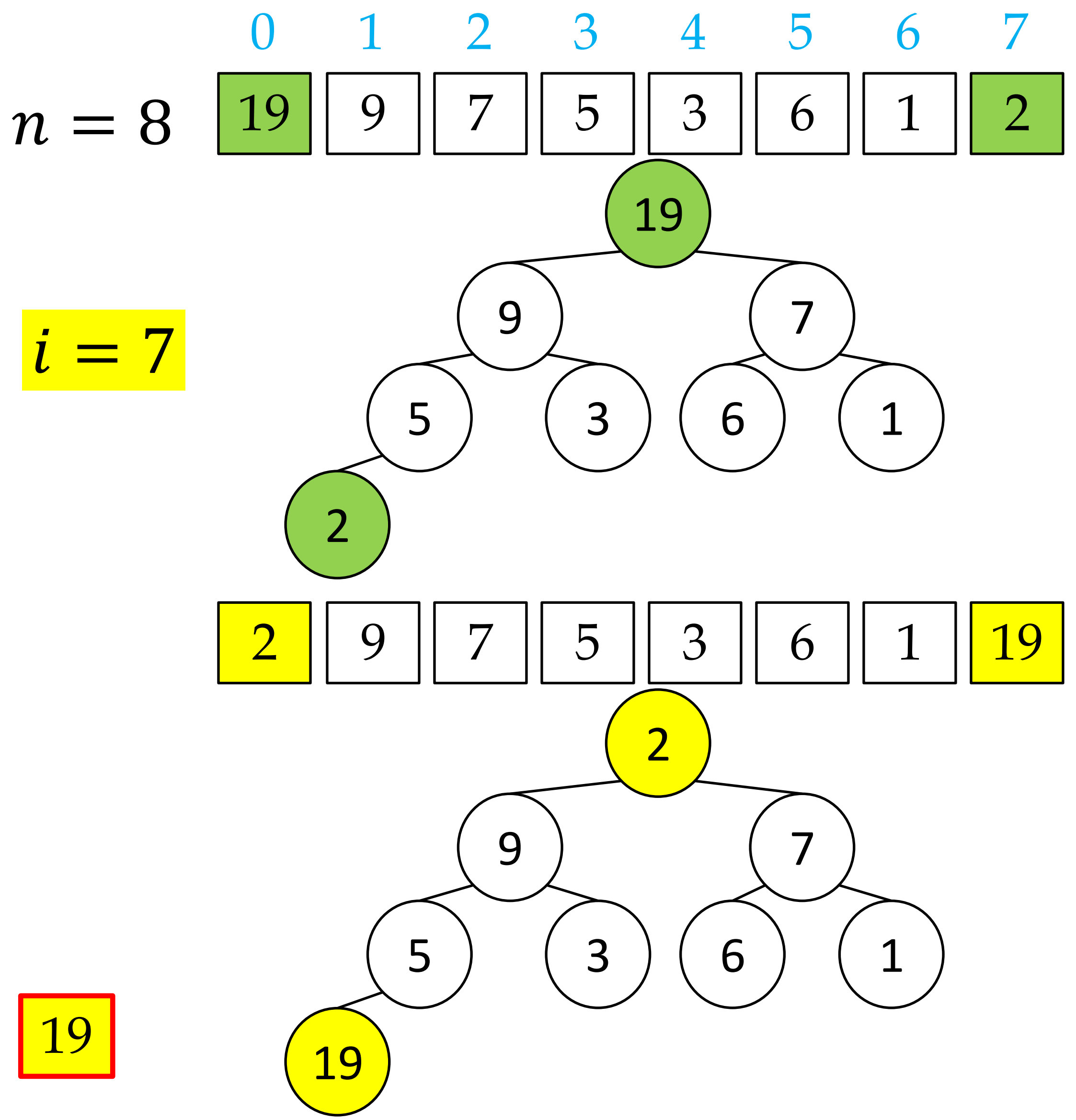
$i = -1$: Stop



Example (6/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

Sorted



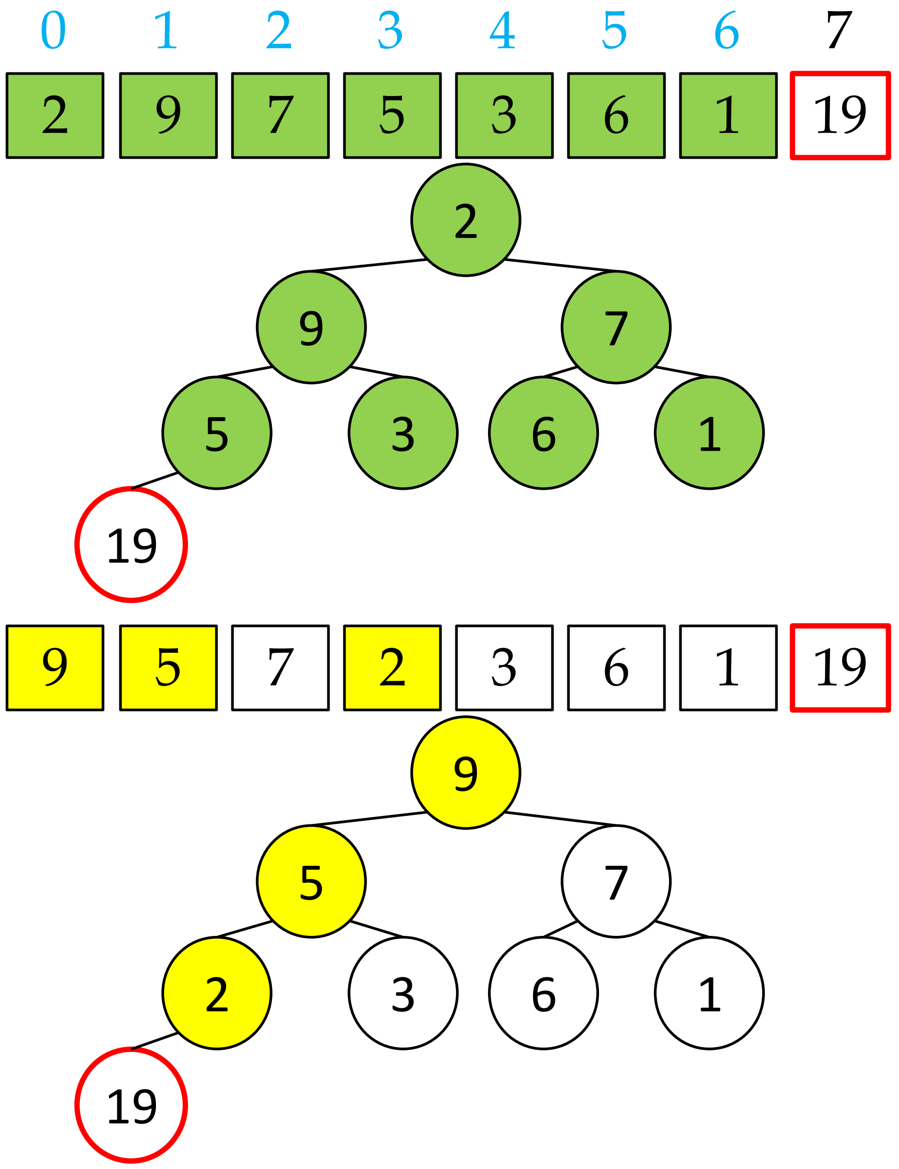
Example (7/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

Sorted

$n = 8$

$i = 7$



19

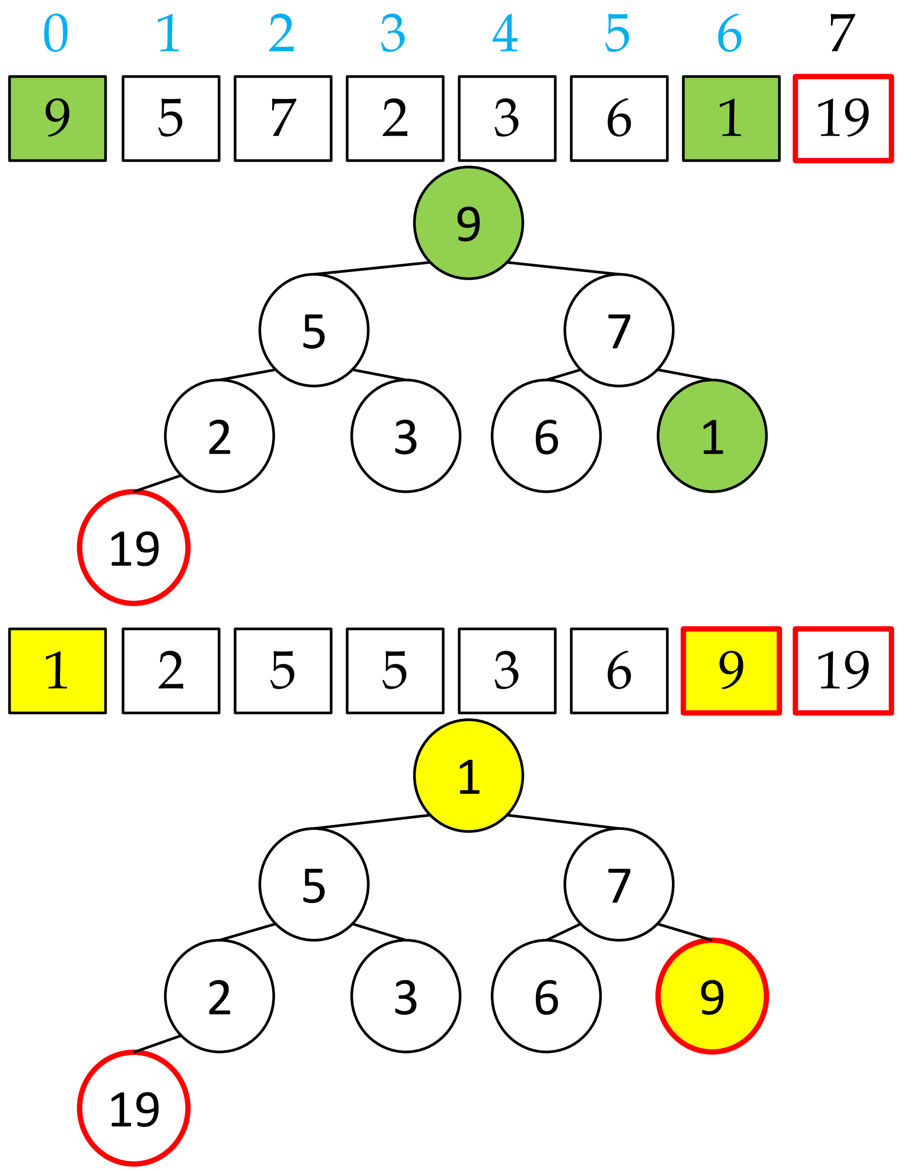
fit@hcmus

Example (8/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

$n = 8$

$i = 6$



Sorted

9	19

Example (9/19)

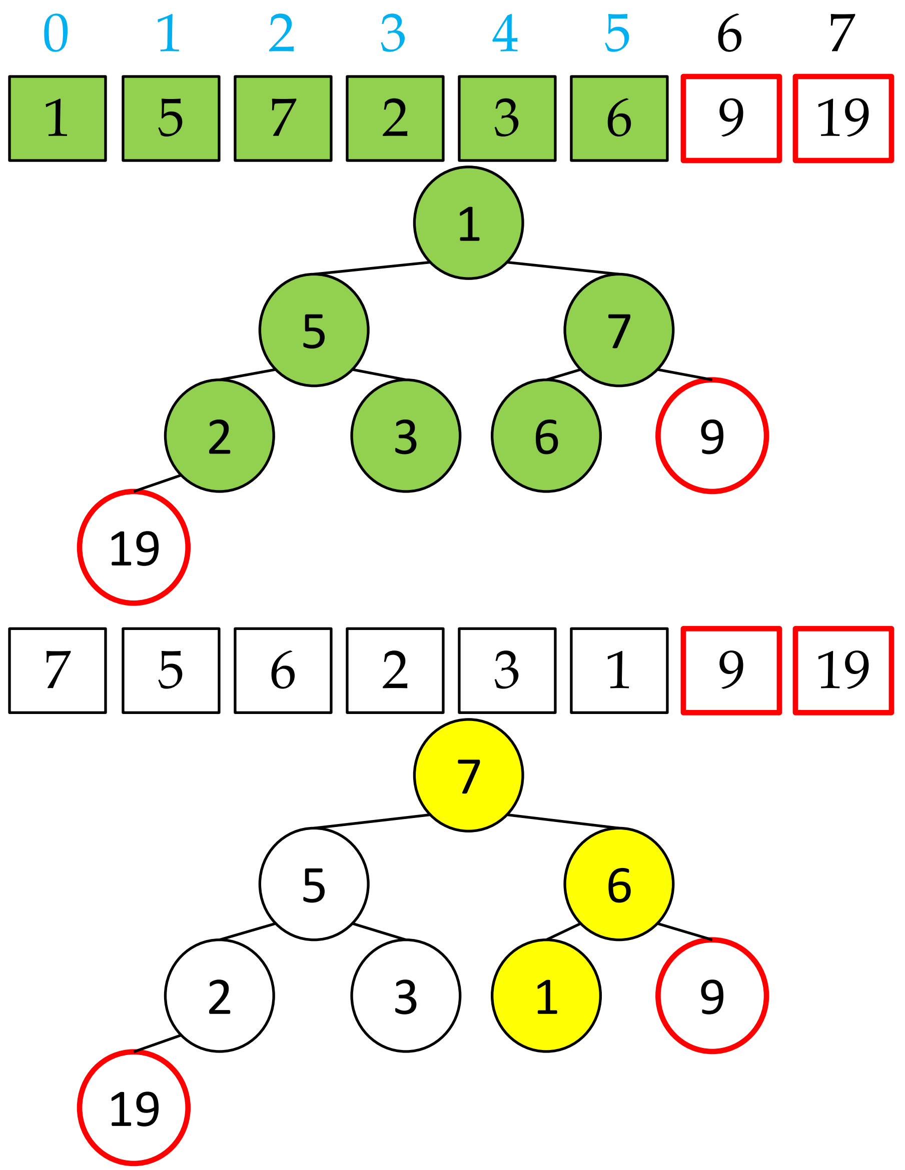
```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

Sorted

9	19

$n = 8$

$i = 6$



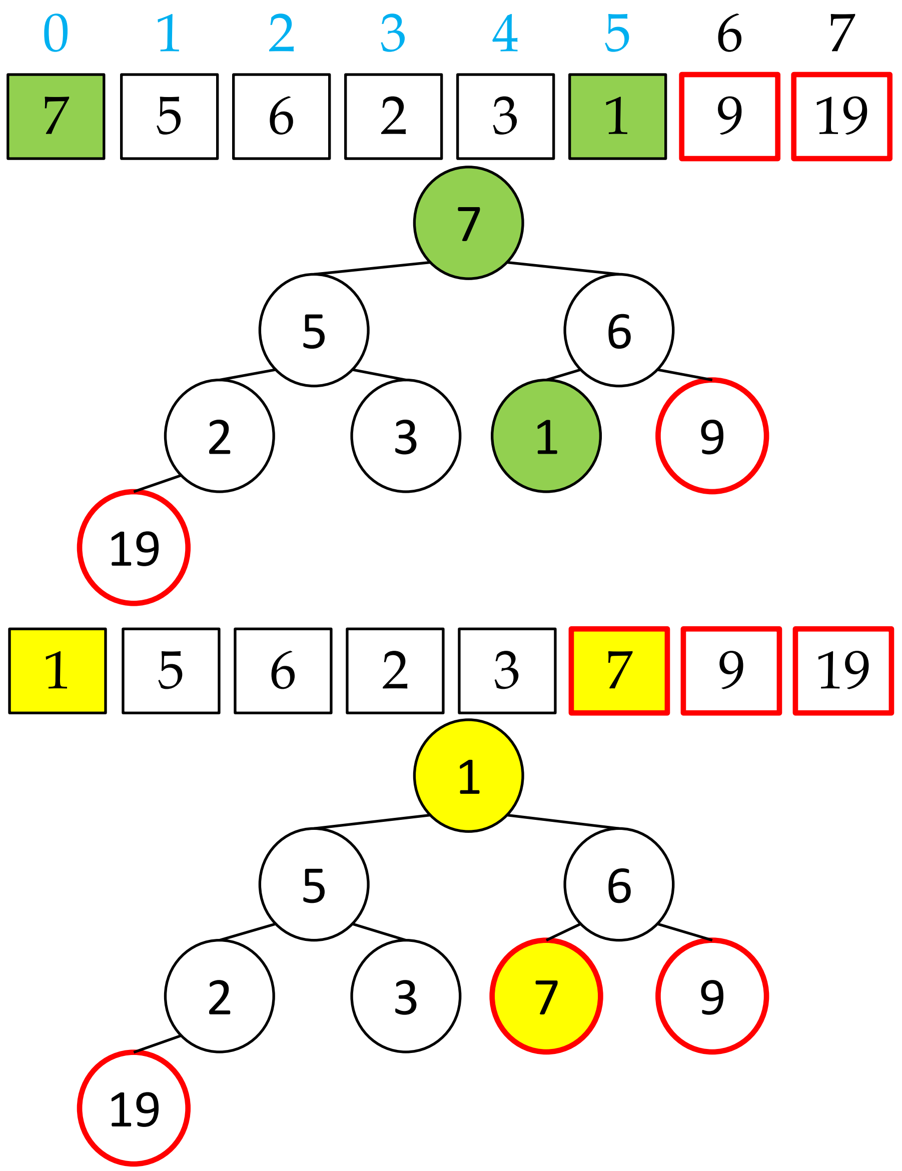
fit@hcmus

Example (10/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

$n = 8$

$i = 5$



Sorted

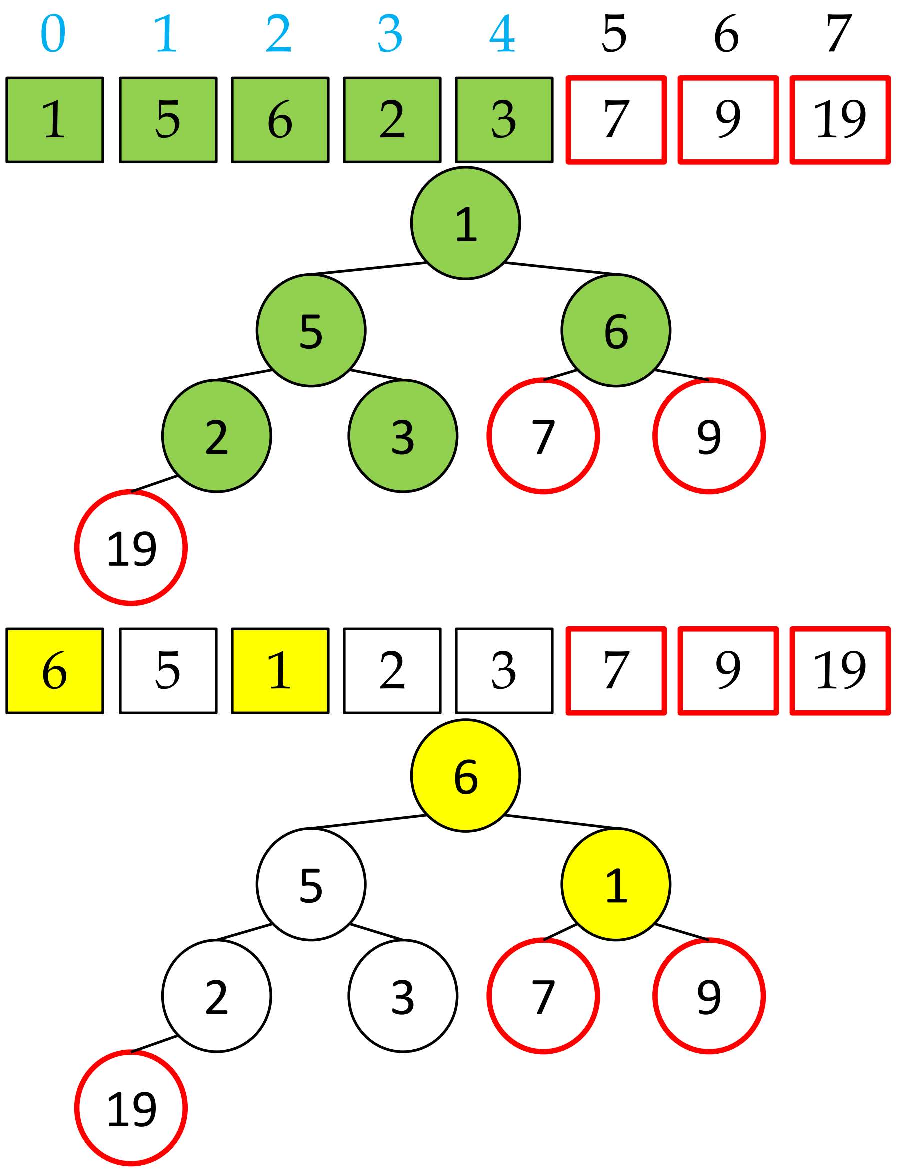
7	9	19

Example (11/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

$n = 8$

$i = 5$



Sorted

7	9	19

fit@hcmus

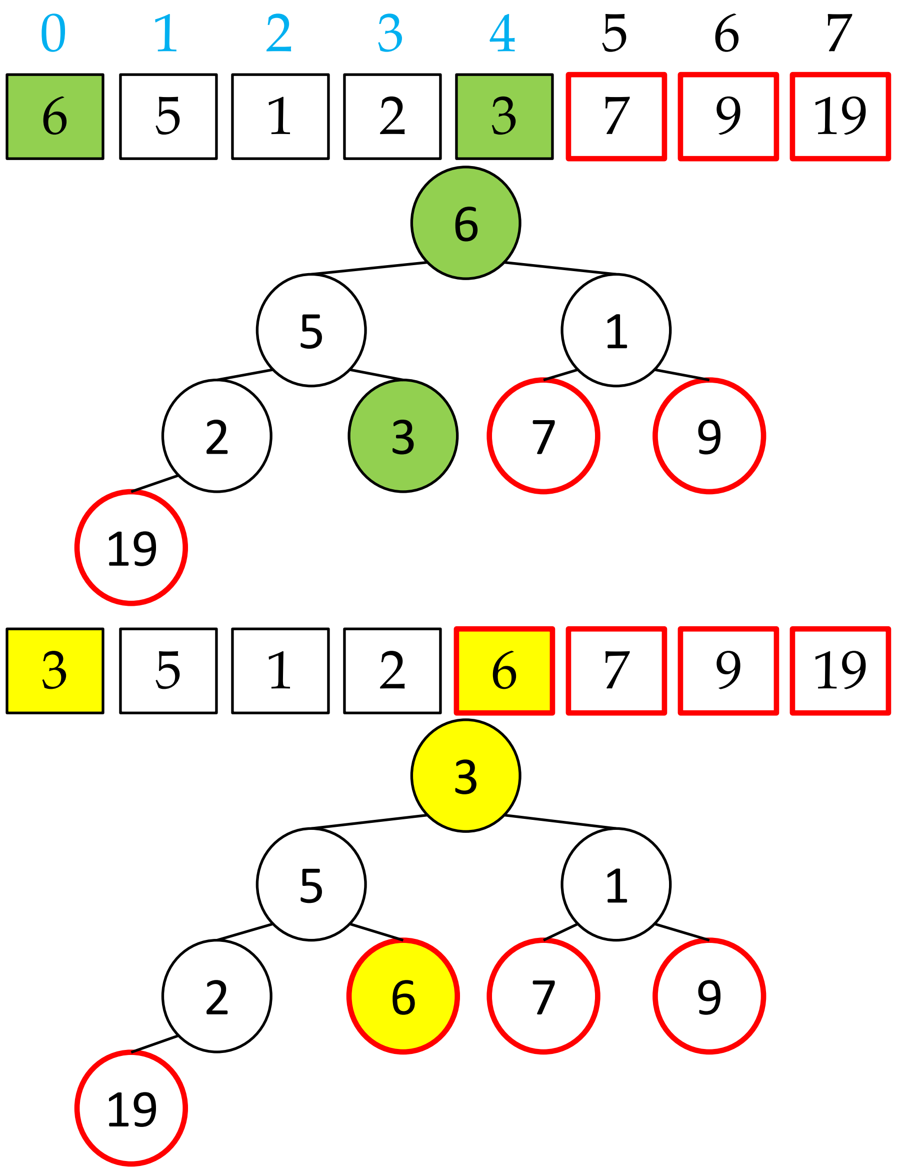
Example (12/19)

1. void heapSort(int arr[], int n) {
2. // Step 1: Build a Max Heap
3. for (int i = n/2 - 1; i >= 0; i--)
4. heapify(arr, n, i);

5. // Step 2: Extract elements one by one from the heap
6. for (int i = n - 1; i > 0; i--) {
7. swap(arr[0], arr[i]); // Move the largest element to the end
8. heapify(arr, i, 0); // Heapify the reduced heap
9. }
10. }

$n = 8$

$i = 4$



Sorted

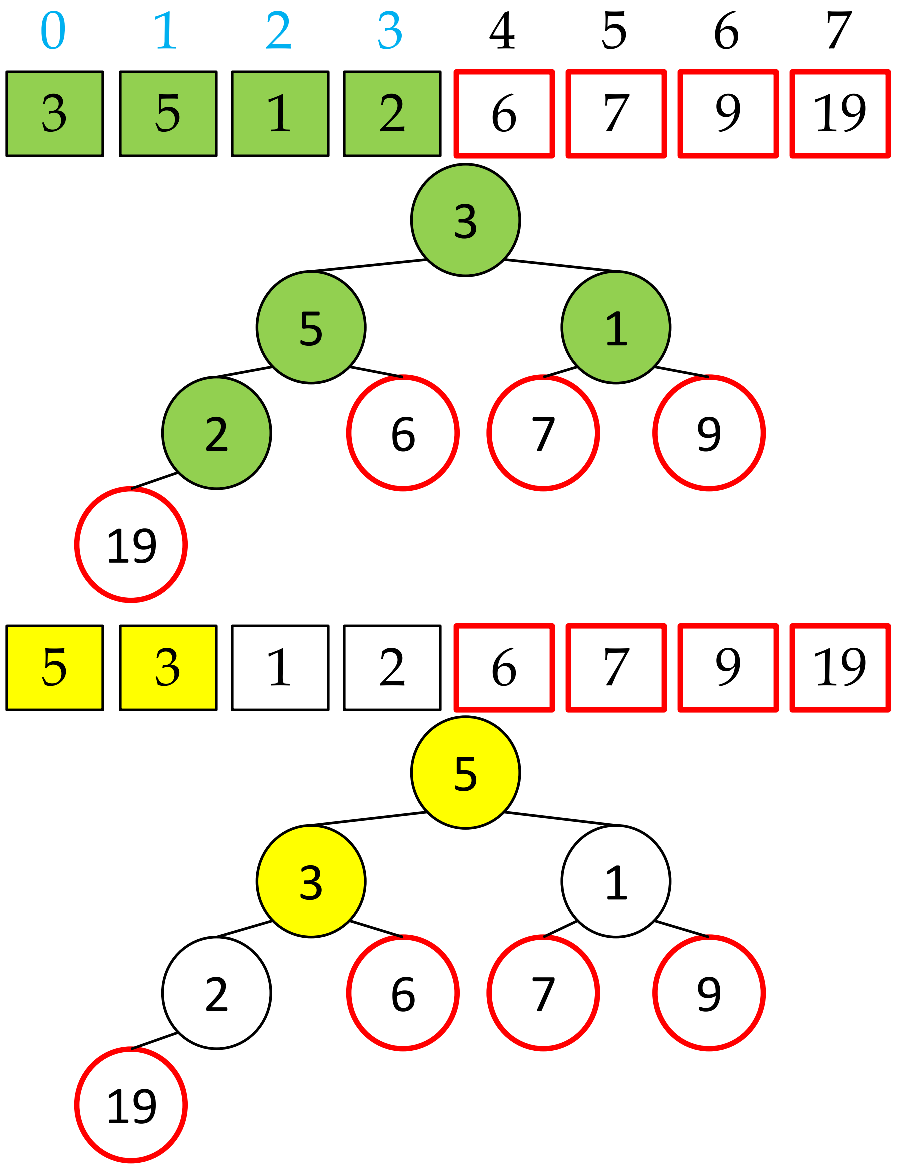
6	7	9	19

Example (13/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

$n = 8$

$i = 4$



Sorted

6	7	9	19

Example (14/19)

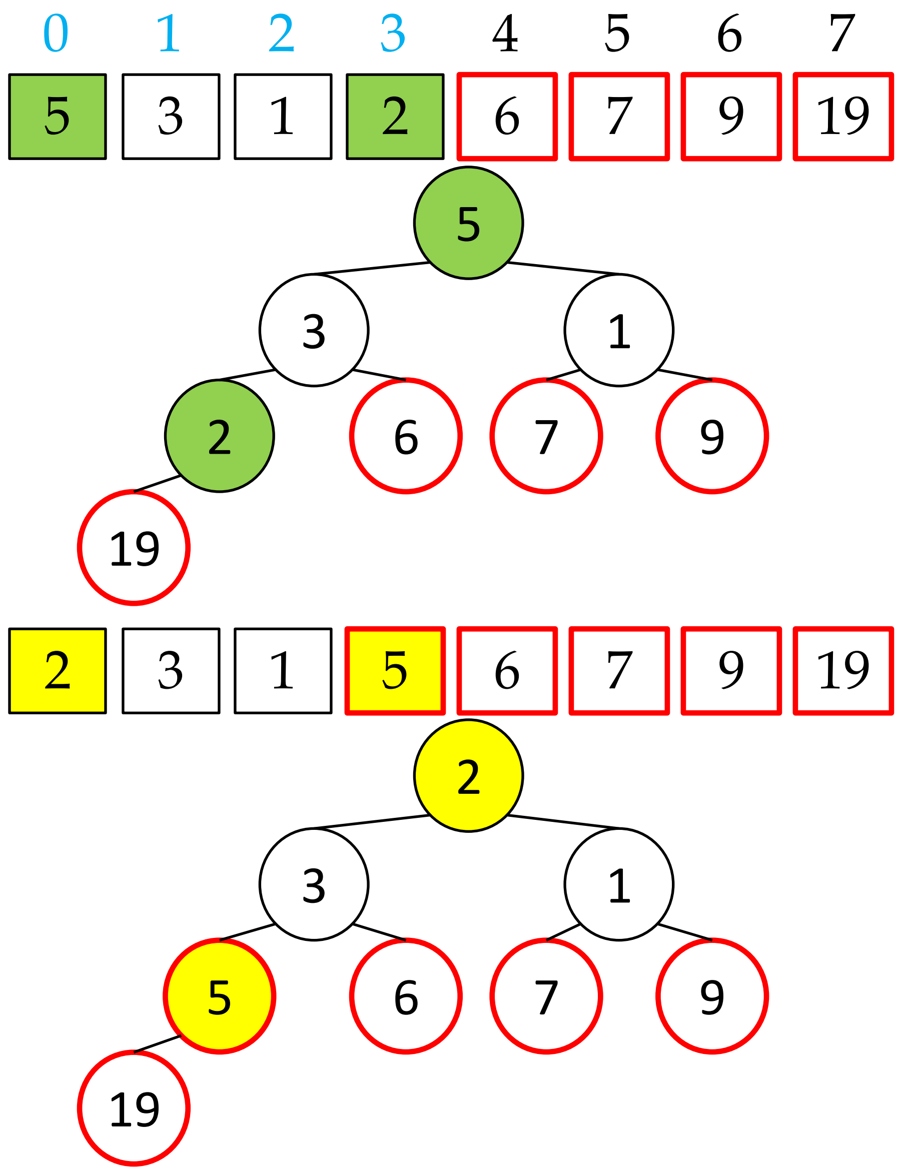
```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

Sorted

5	6	7	9	19

$n = 8$

$i = 3$

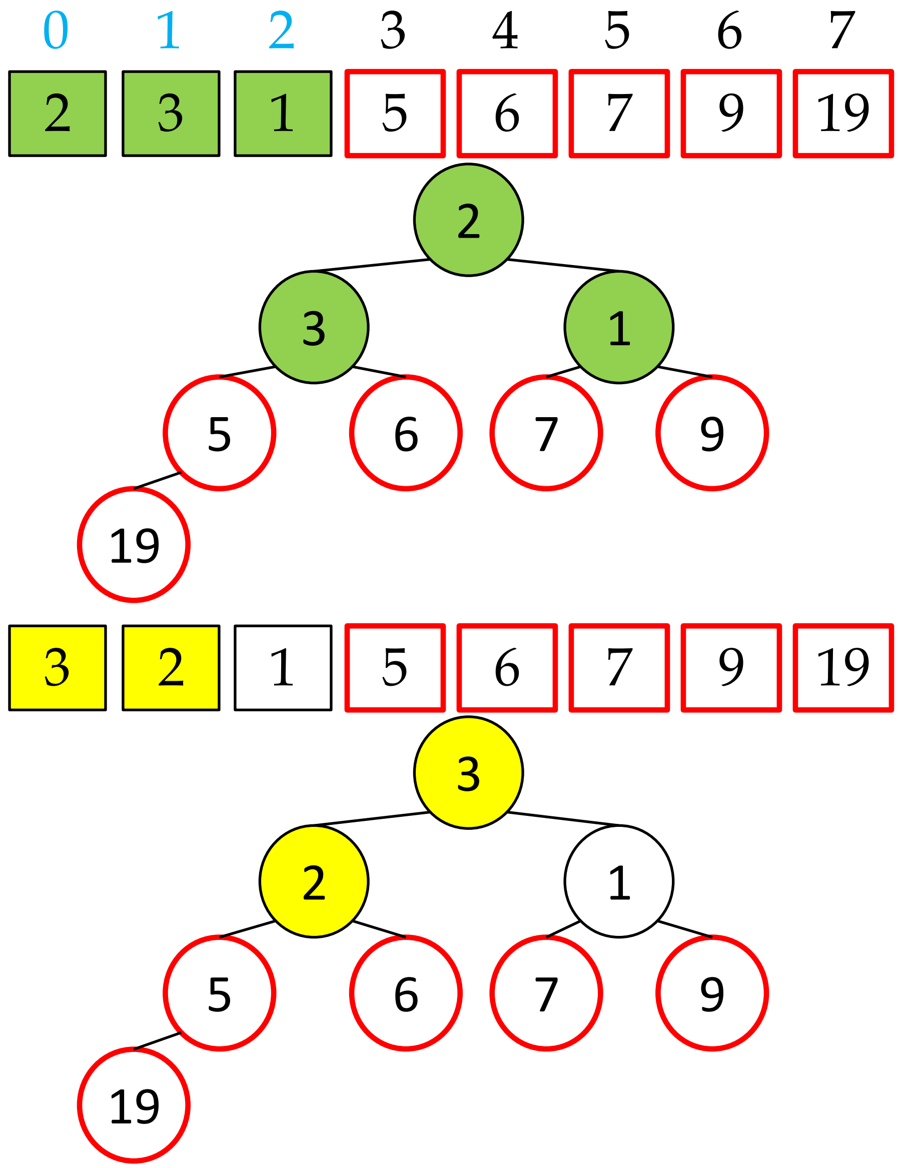


Example (15/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the
    heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to
    the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

$n = 8$

$i = 3$



Sorted

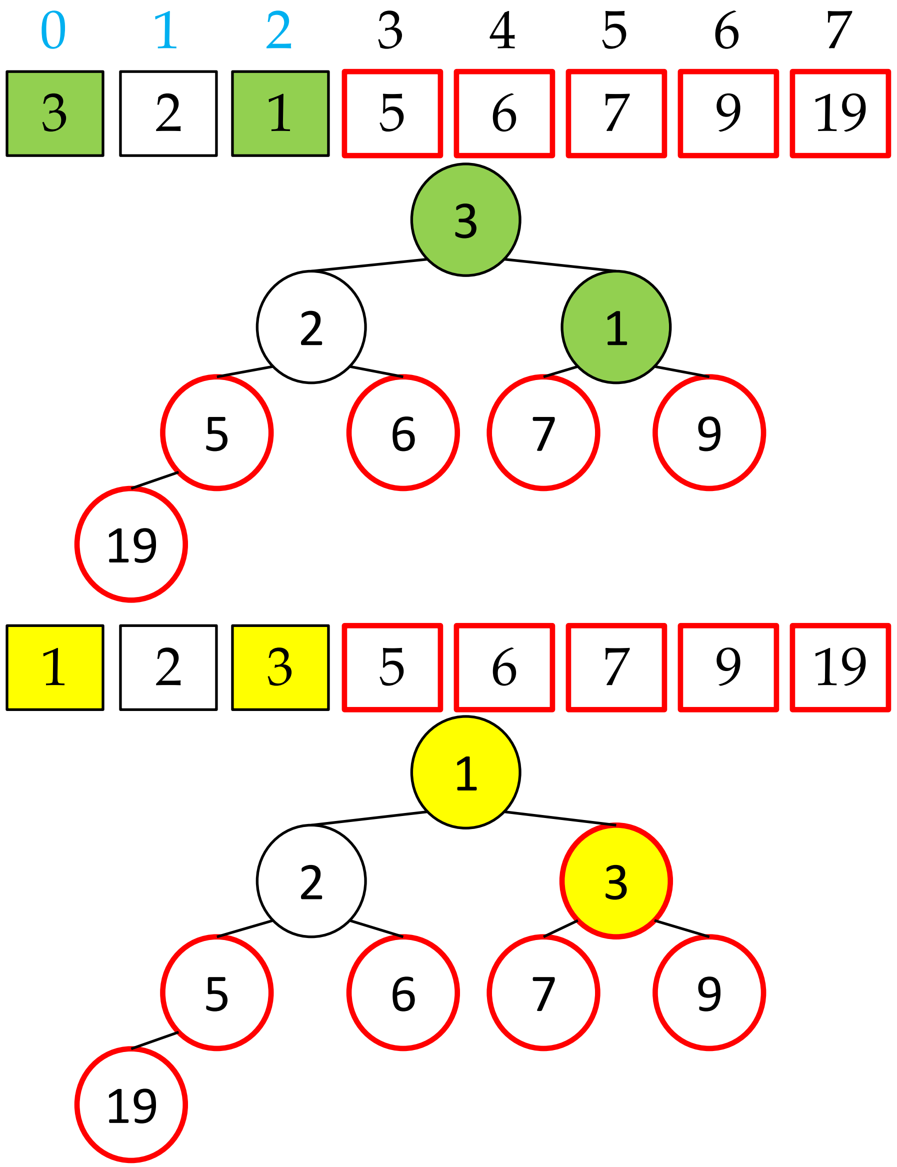
3	5	6	7	9	19

Example (16/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the
    heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to
    the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

$n = 8$

$i = 2$



Sorted

3	5	6	7	9	19

Example (17/19)

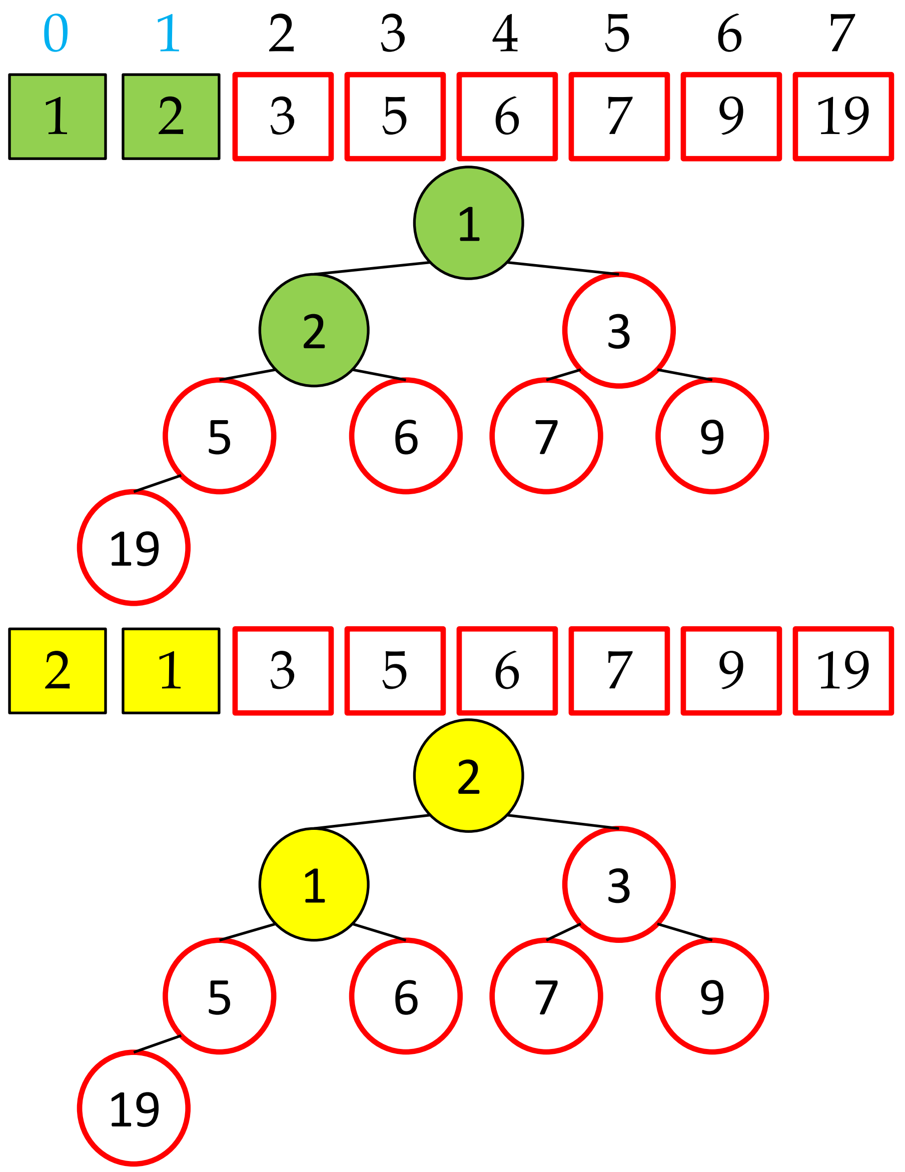
```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the
    heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to
    the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

Sorted

3	5	6	7	9	19

$n = 8$

$i = 2$

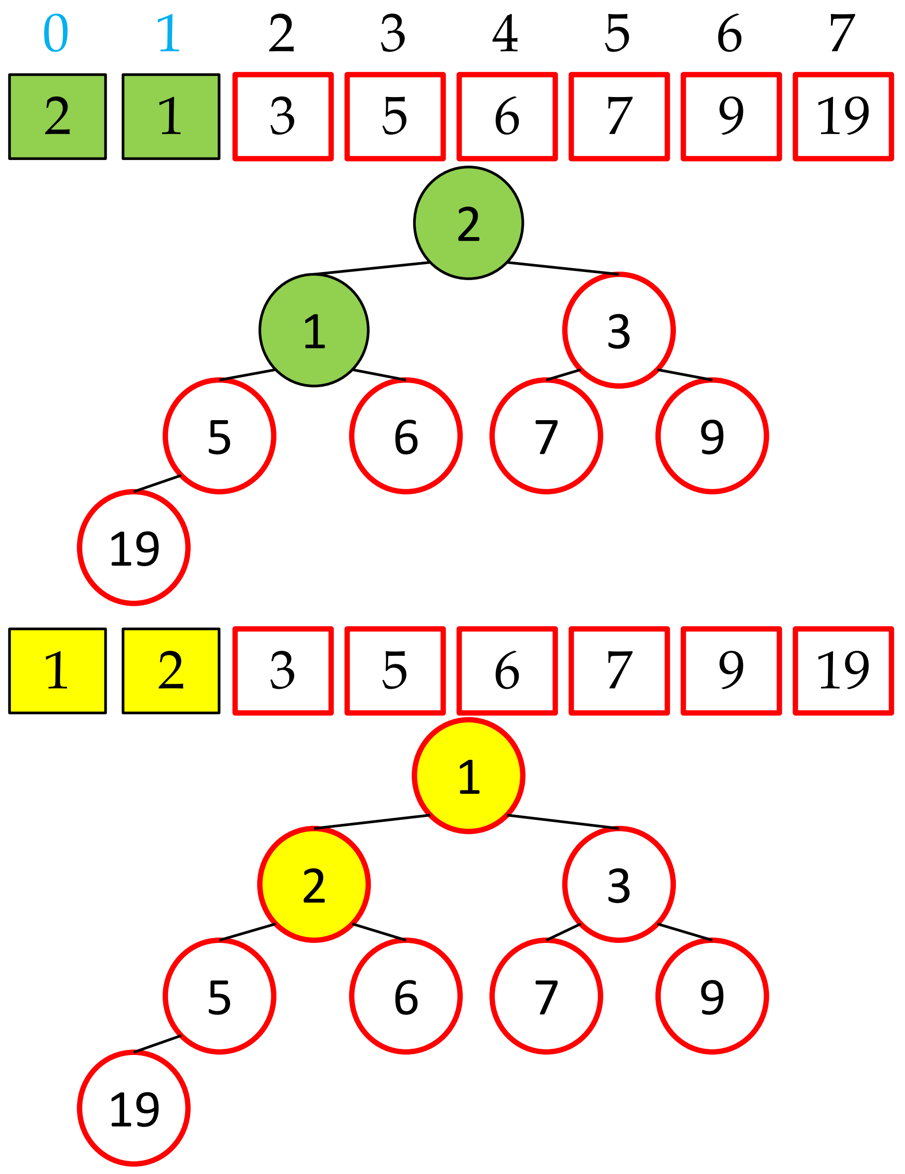


Example (18/19)

```
void heapSort(int arr[], int n) {
    // Step 1: Build a Max Heap
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Step 2: Extract elements one by one from the heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move the largest element to the end
        heapify(arr, i, 0);   // Heapify the reduced heap
    }
}
```

$n = 8$

$i = 1$



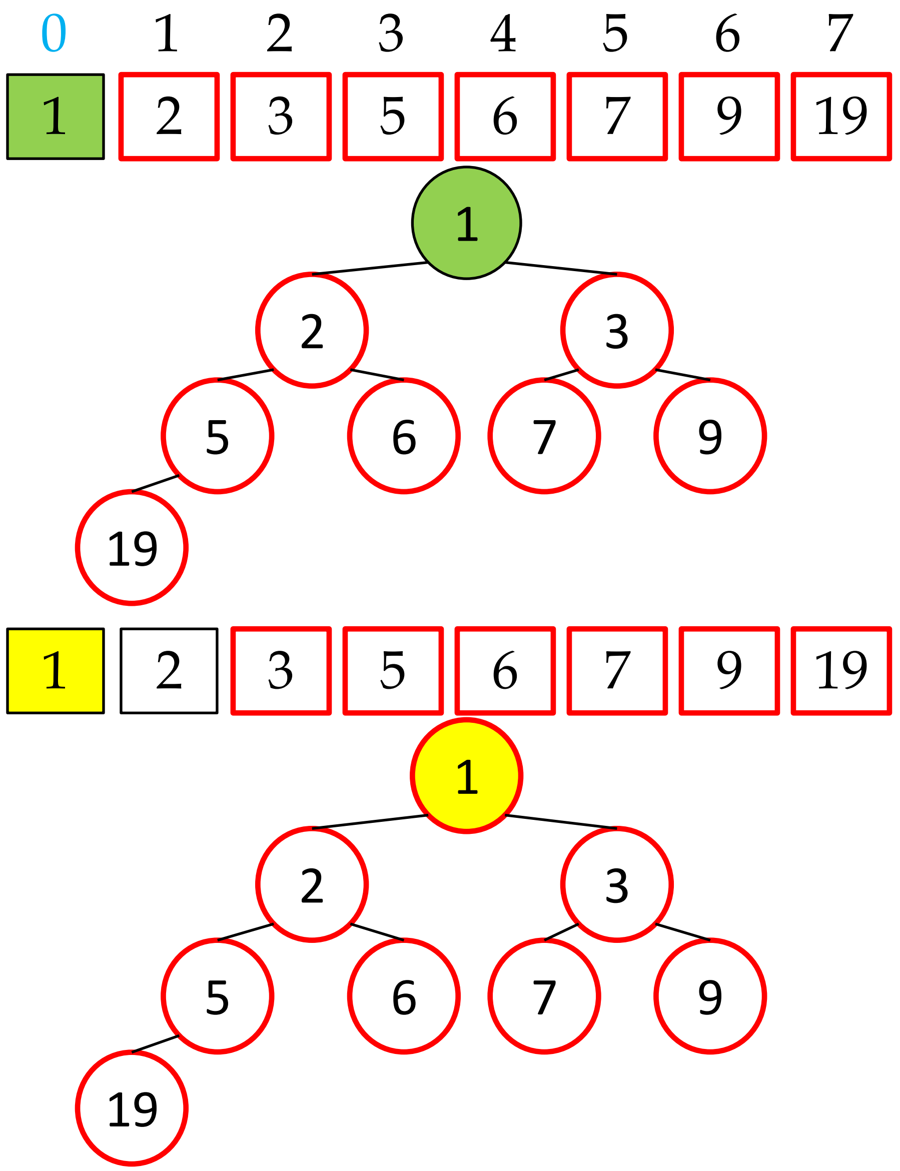
Sorted	1	2	3	5	6	7	9	19

Example (19/19)

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i);

5.      // Step 2: Extract elements one by one from the heap
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

$n = 8$

$i = 1$



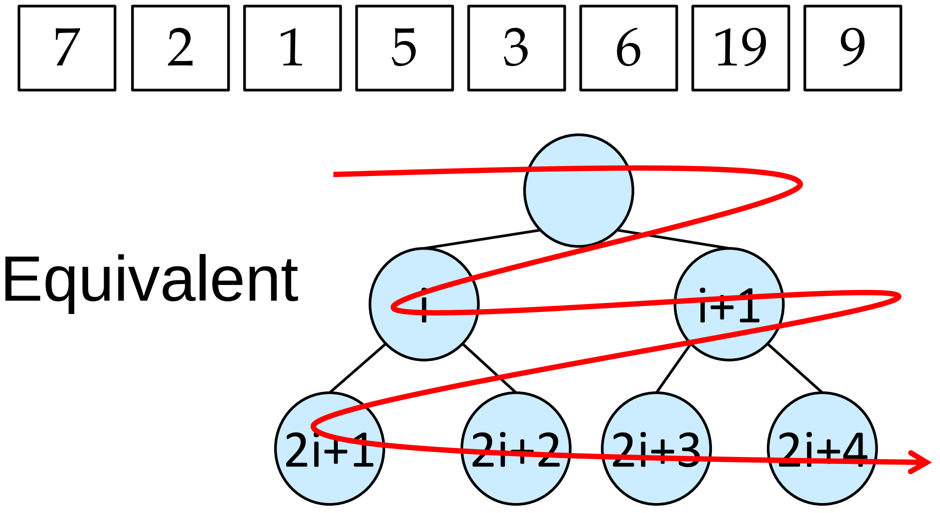
Sorted

1	2	3	5	6	7	9	19

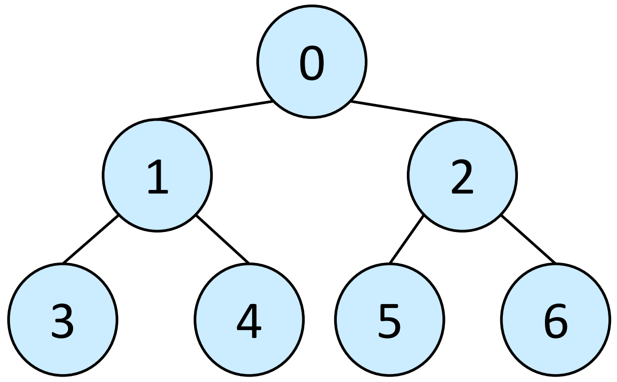
Analysis: Correctness of heapify

```
void heapify(int arr[], int n, int i) {
    int largest = i;      // Assume root (i) is the largest
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than the largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;
    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        swap(arr[i], arr[largest]); // Swap root with the largest child
        heapify(arr, n, largest);  // Recursively heapify the affected subtree
    }
}
```

Analysis: Correctness of HeapSort: The Indices of a Balanced Binary Tree Are Consecutive (1/2)



For $n = 7$ nodes:



We consider a **binary tree** where:

- Root is at index $i = 0$.
- Left child of node i is at index $2i + 1$.
- Right child of node i is at index $2i + 2$.
- The tree is **balanced** (each level is fully filled except possibly the last).

- We will prove that **the indices assigned follow a consecutive sequence** from 0 to $n - 1$, where n is the number of nodes.

Analysis: Correctness of HeapSort: The Indices of a Balanced Binary Tree Are Consecutive (2/2)

Proved by INDUCTION.

- A complete binary tree of height h has $n = 1 + 2 + \cdots + 2^h = 2^{h+1} - 1$.
- Numbering at Level d:
  - At depth d, the first node has index $2^d - 1$, the last node has index $2^{d+1} - 2$.
  - Total nodes at depth d is exactly 2^d, which is exactly the number of indices needed to remain consecutive.
- Inductive Step: If the indices are consecutive at level d, then at level $d + 1$, the children of those nodes will also be consecutive.
  - The first child at level $d + 1$ is at index $2(2^d - 1) + 1 = 2^{d+1} - 1$, which is exactly where we expect the next consecutive index to begin.
  - This ensures no gaps appear in the sequence.
- Conclusion:
- Every node receives a unique index, and every index is used exactly once.
- The numbering follows a level-order sequence, ensuring consecutive indices from 0 to $n - 1$.
- Thus, the indices in this balanced binary tree are consecutive.

Analysis: Correctness

```
void heapSort(int arr[], int n) {
    // Step 1: Build a Max Heap
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Step 2: Extract elements one by one from the heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move the largest element to the end
        heapify(arr, i, 0);   // Heapify the reduced heap
    }
}
```

Loop Invariant: At the start of each iteration of the for loop, each node i+1, i+2, …, n is the ROOT of a max-heap.

Initialization:

- Before first iteration i = n/2
- Nodes n/2+1, n/2+2, …, n are leaves and hence roots of max-heaps.

Maintenance:

- Subtrees at children of node i are max heaps.
- Heapify(i) renders node i a max heap root (while preserving the max heap root property of higher-numbered nodes).
- Decrementing i reestablishes the loop invariant for the next iteration.

Analysis: Complexity: average case

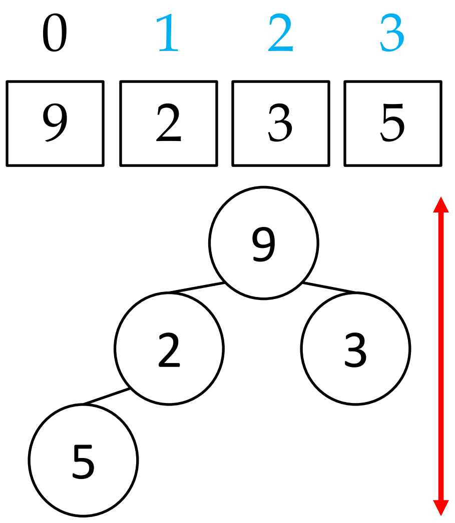
1.	`void heapSort(int arr[], int n) {`	1.	
2.	`    // Step 1: Build a Max Heap`	2.	
3.	`    for (int i = n/2 - 1; i >= 0; i--)`	3.	n/2 iterations
4.	`        heapify(arr, n, i);`	4.	Cost of heapify of an array with length n-i+1
5.	`    // Step 2: Extract elements one by one from the heap`	5.	
6.	`    for (int i = n - 1; i > 0; i--) {`	6.	n-1 iterations
7.	`        swap(arr[0], arr[i]); // Move the largest element to the end`	7.	3 assignments
8.	`        heapify(arr, i, 0);   // Heapify the reduced heap`	8.	Cost of heapify of an array with length i+1
9.	`    }`		
10.	`}`		

$$\leq \sum_{i=0}^{\frac{n}{2}-1} \mathrm{Cost}(\mathrm{Heapify}(n-i+1) + \sum_{i=1}^{n-1} \mathrm{Cost}\big(\mathrm{Heapify}(i+1)\big)$$

Complexity of heapify

```
void heapify(int arr[], int n, int i) {
    int largest = i;      // Assume root (i) is the largest
    int left = 2 * i + 1; // Left child
    int right = 2 * i + 2; // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than the largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        swap(arr[i], arr[largest]); // Swap root with the largest child
        heapify(arr, n, largest);  // Recursively heapify the affected subtree
    }
}
```

$n = 4, i = 1$



Heapify recursively runs up to h times

runs in h times

$h \leq \lceil \log(n - i + 1) \rceil$

Analysis: Complexity: average case

```
1.  void heapSort(int arr[], int n) {
2.      // Step 1: Build a Max Heap
3.      for (int i = n/2 - 1; i >= 0; i--)
4.          heapify(arr, n, i
5.      // Step 2: Extract
6.      for (int i = n - 1; i > 0; i--) {
7.          swap(arr[0], arr[i]); // Move the largest element to
    the end
8.          heapify(arr, i, 0);   // Heapify the reduced heap
9.      }
10. }
```

1.
2.
3. n/2 iterations
4. Cost of heapify of an array with length n-i+1
6. n-1 iterations
7. 3 assignments
8. Cost of heapify of an array with length i+1

$$\le \sum_{i=0}^{\frac{n}{2}-1} \log(n - i + 1) + \sum_{i=1}^{n-1} \log(i + 1) = O(n \log n)$$

$$\le \sum_{i=0}^{\frac{n}{2}-1} \mathrm{Cost}(\mathrm{Heapify}(n - i + 1) + \sum_{i=1}^{n-1} \mathrm{Cost}\big(\mathrm{Heapify}(i + 1)\big)$$

		Design	Run time				Space	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$	$O(\log n)$	$O(n)$		Yes	No
	Insertion sort	Det.	$O(n^2)$				Yes	Yes
	Heap sort	Det.	$O(n \log n)$				Yes	No
	Merge sort	?	?				?	?
	Quick sort	?	?				?	?
Non-comparison-based	Radix sort	?	?				?	?

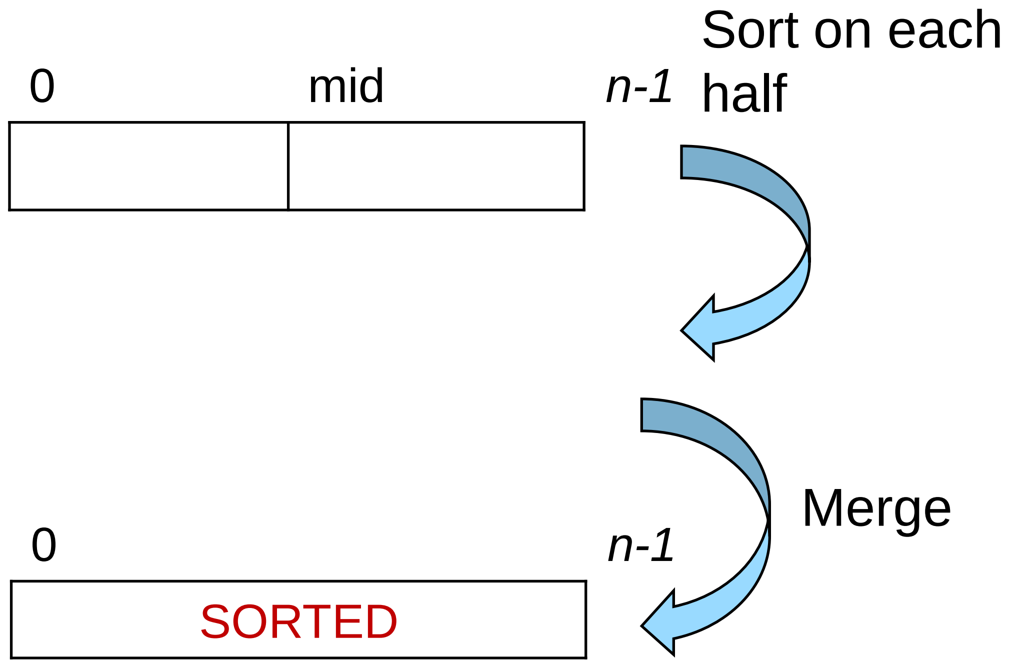
Outline

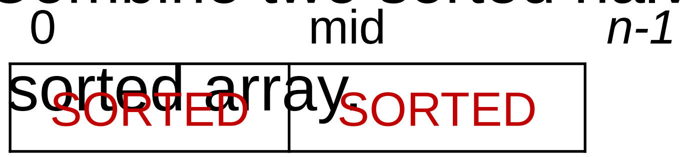
1. Selection Sort
2. Insertion Sort
3. Heap Sort
4. Merge Sort
5. Quick Sort
6. Radix Sort

Idea

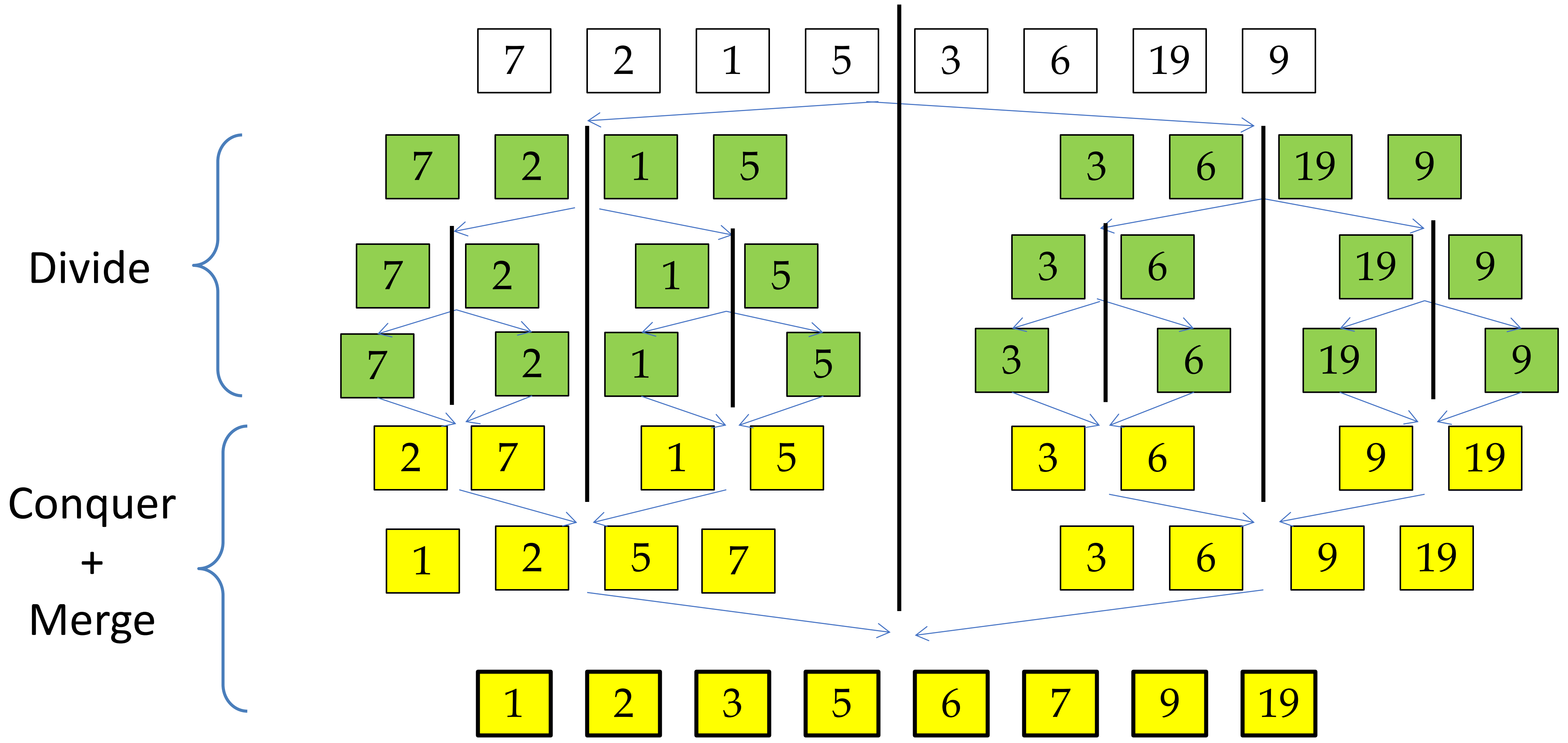
Merge Sort is a divide and conquer sorting algorithm that splits the array into smaller parts, sorts them, and then merges them back together:

1. Divide: Split the array into two halves until each part has only one element.
   – An array with 0 or 1 element is a sorted array.
2. Conquer: Recursively sort each half.
3. Merge: Combine two sorted halves back into a single sorted array.



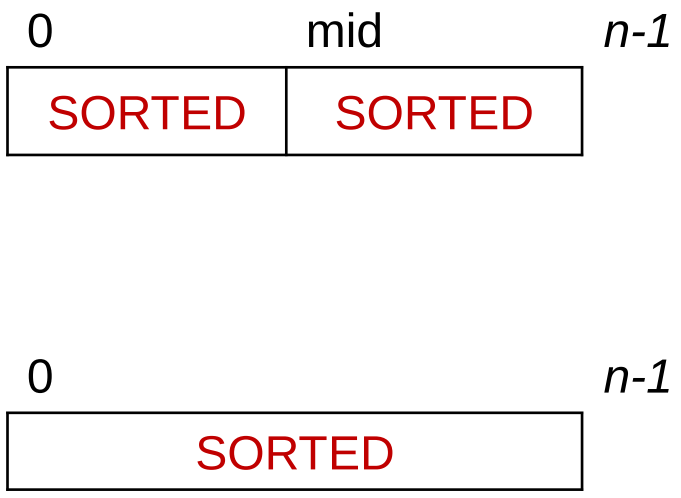


Example


7 2 1 5 3 6 19 9
7 2 1 5
3 6 19 9
7 2
1 5
3 6
19 9
7
2
1
5
3
6
19
9
2 7
1 5
3 6
9 19
1 2 5 7
3 6 9 19
1 2 3 5 6 7 9 19
Divide
Conquer
+
Merge

Source code (1/2)

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```



Source code (2/2): Function to merge two sorted arrays

```c
void merge(int arr[], int left, int mid, int right) {
    int i = left, j = mid + 1, k = 0;
    int temp[right - left + 1]; // Temporary array to store merged elements

    // Merge elements from both halves into temp[]
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else
            temp[k++] = arr[j++];
    }
    // Copy remaining elements from left half
    while (i <= mid) {
        temp[k++] = arr[i++];
    }
    // Copy remaining elements from right half
    while (j <= right)
        temp[k++] = arr[j++];

    // Copy merged elements back into original array
    for (int x = 0; x < k; x++) {
        arr[left + x] = temp[x];
    }
}
```

Analysis: Correctness

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```

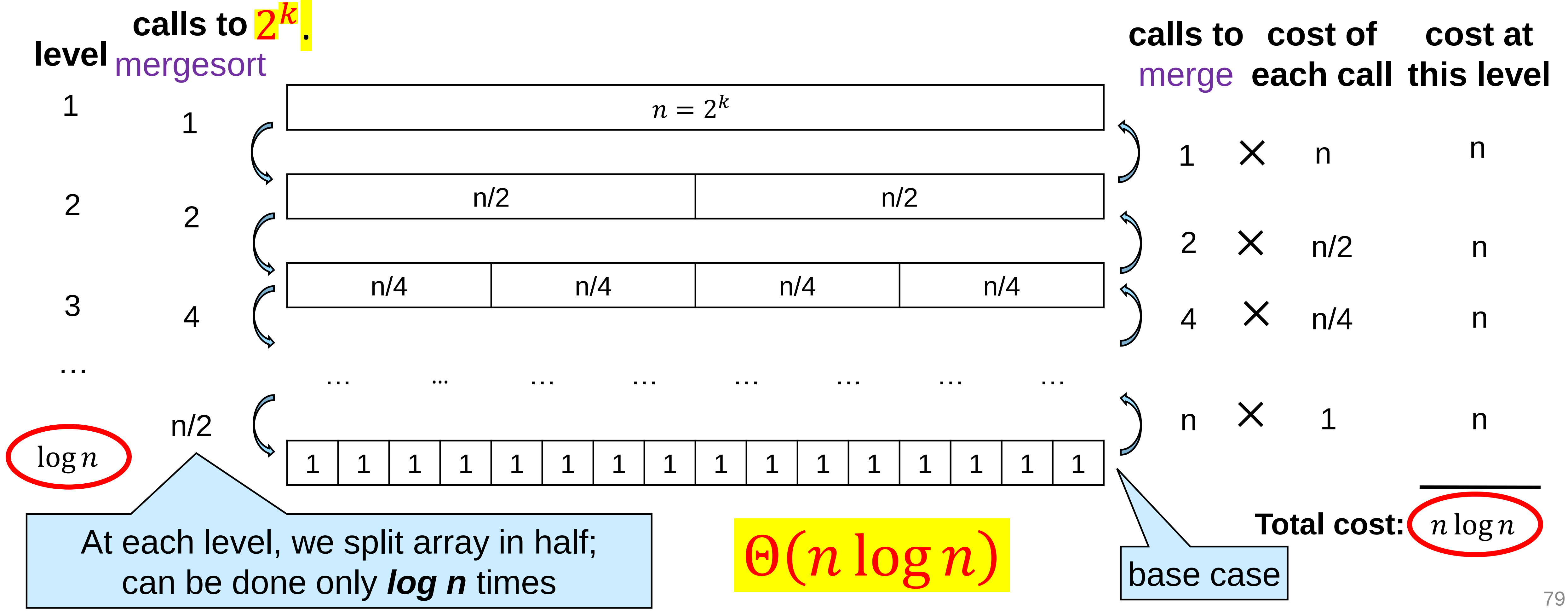
- Function merge always produces a sorted sub-array of arr with indices from left to right as described in the function, then the outcome of mergeSort must be a sorted array with indices from left to right.
- Set left = 0 and right = n-1, then the array arr is sorted.

Analysis: Complexity (1/2)

1.	`void mergeSort(int arr[], int left, int right) {`	1.	T(right-left)
2.	`    if (left < right) {`	2.	1 comparison
3.	`        int mid = left + (right - left) / 2;`	3.	1 assigment
4.	`        // Recursively sort first and second halves`	4.	// Recursively sort first and second halves
5.	`        mergeSort(arr, left, mid);`	5.	T(mid-left+1);
6.	`        mergeSort(arr, mid + 1, right);`	6.	T(right-mid);
7.	`        // Merge the sorted halves`	7.	// Merge the sorted halves
8.	`        merge(arr, left, mid, right);`	8.	O(right-left)
9.	`    }`	9.	
10.	`}`		

Analysis: Complexity (2/2)

For any positive n, there always exists ONLY k s.t. $2^{k-1} \leq n < 2^k$. Since $T(2^{k-1}) \leq T(n) \leq T(2^k)$, we can set $n = 2^k$.



At each level, we split array in half; can be done only ***log n*** times

$\Theta(n \log n)$

Complexity in **average**

		Design	**Run time**				**Space**	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$				Yes	No
	Insertion sort	Det.	$O(n^2)$				Yes	Yes
	Heap sort	Det.	$O(n \log n)$	$O(\log n)$			Yes	No
	Merge sort	Det.	$\Theta(n \log n)$		$O(n)$		No	Yes
	Quick sort	?	?				?	?
Non-comparison-based	Radix sort	?	?				?	?

Outline

1. Selection Sort
2. Insertion Sort
3. Heap Sort
4. Merge Sort
5. Quick Sort
6. Radix Sort

Idea (1/2)

- Like Merge Sort, Quick Sort is based on divide-and-conquer strategy.

- It works as follows:
- First, it partitions an array into two parts,
- Then, it sorts the parts independently,
- Finally, it combines the sorted parts by a simple concatenation.

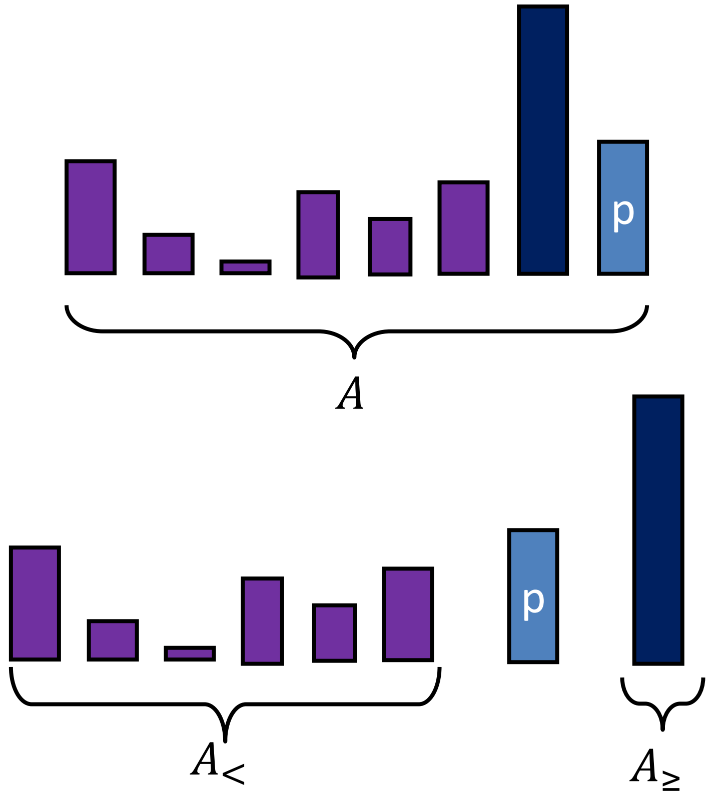
Idea (2/2)

The algorithm consists of the following three steps:

- Divide step:
  - Pick any element (***pivot***) p in A
  - Partition $A - \{p\}$ into two disjoint groups

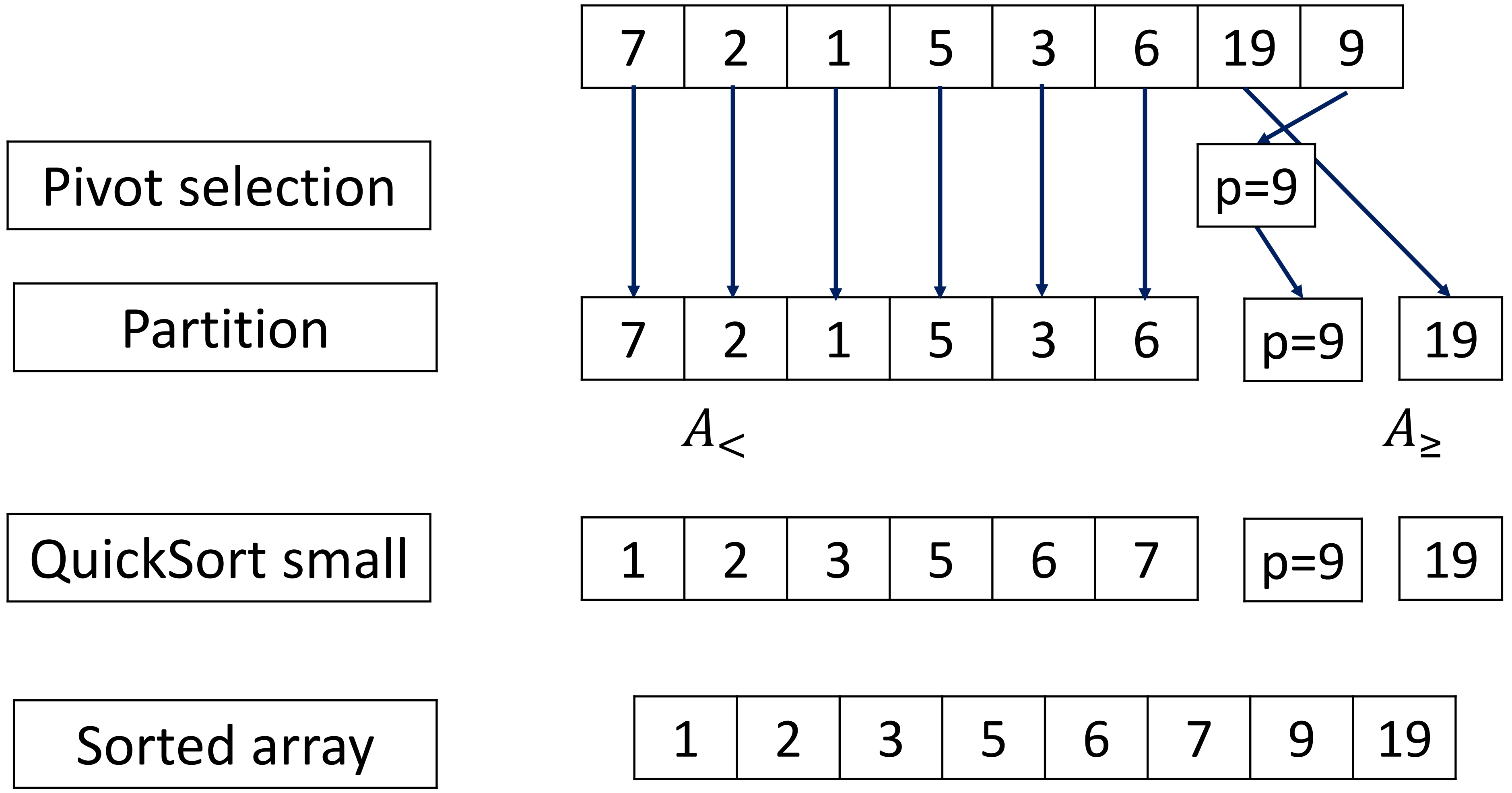
    $A_< = \{ x \in A - \{p\} \mid x < p \}$

    $A_\geq = \{ x \in A - \{p\} \mid x \geq p \}$



- Conquer step: recursively sort $A_<$ and $A_\geq$
- Combine step: the sorted $A_<$ (by the time returned from recursion), followed by p, followed by the sorted $A_\geq$ (i.e., nothing extra needs to be done)

Example



7 2 1 5 3 6 19 9
Pivot selection
p=9
Partition
7 2 1 5 3 6
p=9
19
$A_<$
$A_\geq$
QuickSort small
1 2 3 5 6 7
p=9
19
Sorted array
1 2 3 5 6 7 9 19

Pseudocode

```
void quickSort(a, left, right) {
    if (left < right) {
        pivot = Partition(a, left, right);
        quickSort(a, left, pivot-1);
        quickSort(a, pivot+1, right);
    }
}
```

Compare with mergeSort:

```
void mergeSort(a, left, right) {
    if (left < right) {
        mid = divide(a, left, right);
        mergeSort(a, left, mid-1);
        mergeSort(a, mid+1, right);
        merge(a, left, mid+1, right);
    }
}
```

Source code

```
void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex-1);  // Sort left part
        quickSort(arr, pivotIndex + 1, high); // Sort right part
    }
}
```

```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // Choose last element as pivot
    int i = low - 1; // Index for smaller elements

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]); // Swap smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place pivot in the correct position
    return i + 1; // Return pivot index
}
```

Two key steps:

- How to pick a pivot?
- How to partition?

Pick a pivot

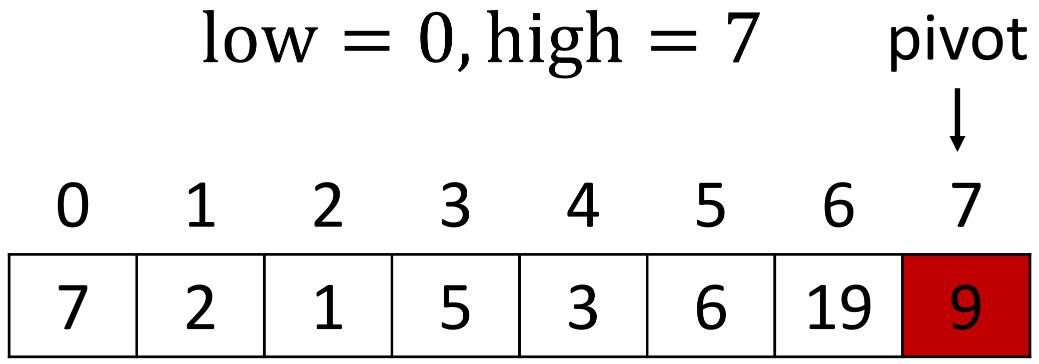
- Use the first element as pivot
  - if the input is random, ok
  - if the input is presorted (or in reverse order)
    - all the elements go into $A_>$ (or $A_\leq$)
    - this happens consistently throughout the recursive calls
    - Results in $O(n^2)$ behavior (Analyze this case later)
- Choose the pivot randomly
  - generally safe
  - random number generation can be expensive

In-place partition

- If use additional array (not in-place) like MergeSort
  - Straightforward to code like MergeSort (write it down!)
  - Inefficient!

- Many ways to implement
  - Even the slightest deviations may cause surprisingly bad results.
  - Not stable as it does not preserve the ordering of the identical keys.
  - Hard to write correctly!

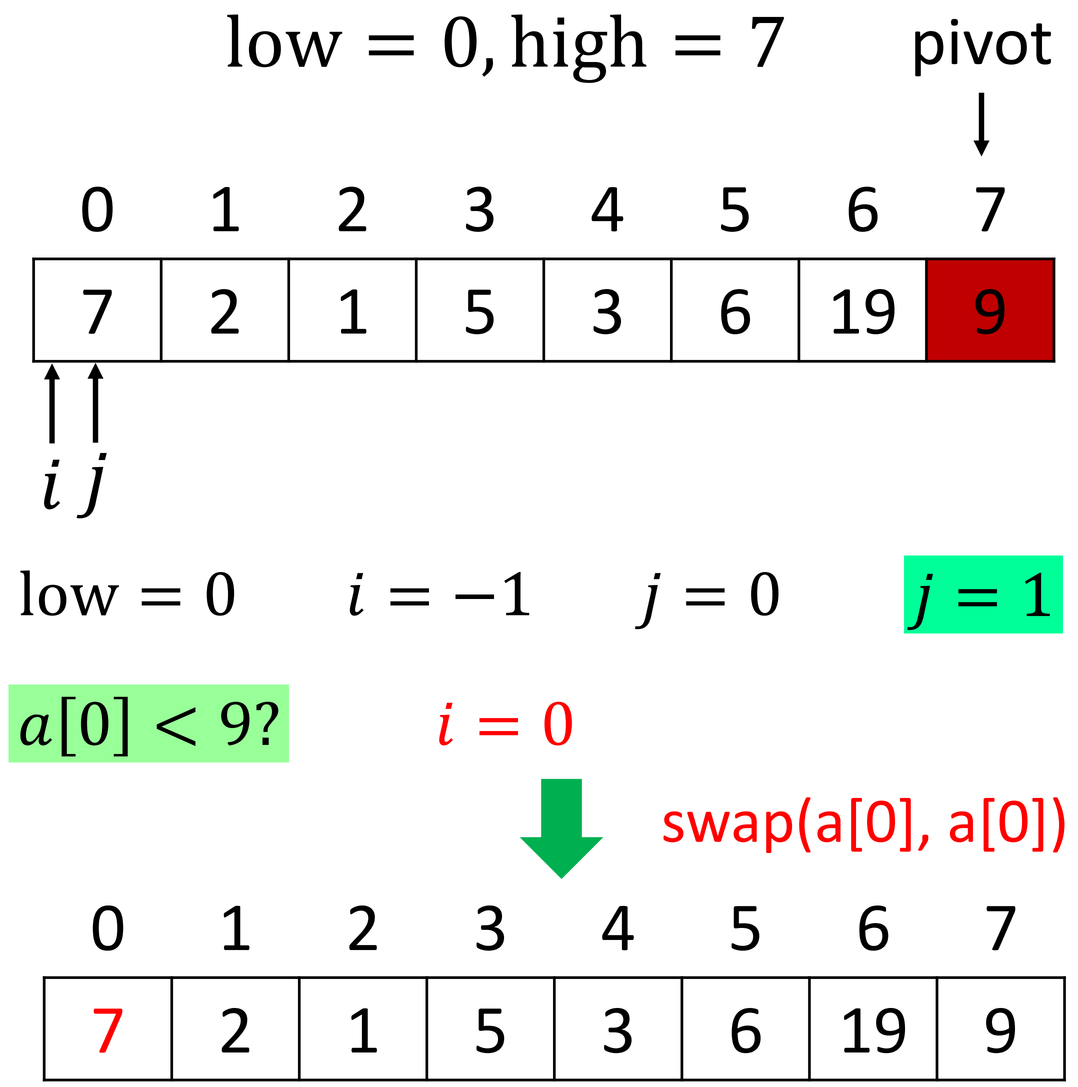
Illustration for partition function



```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

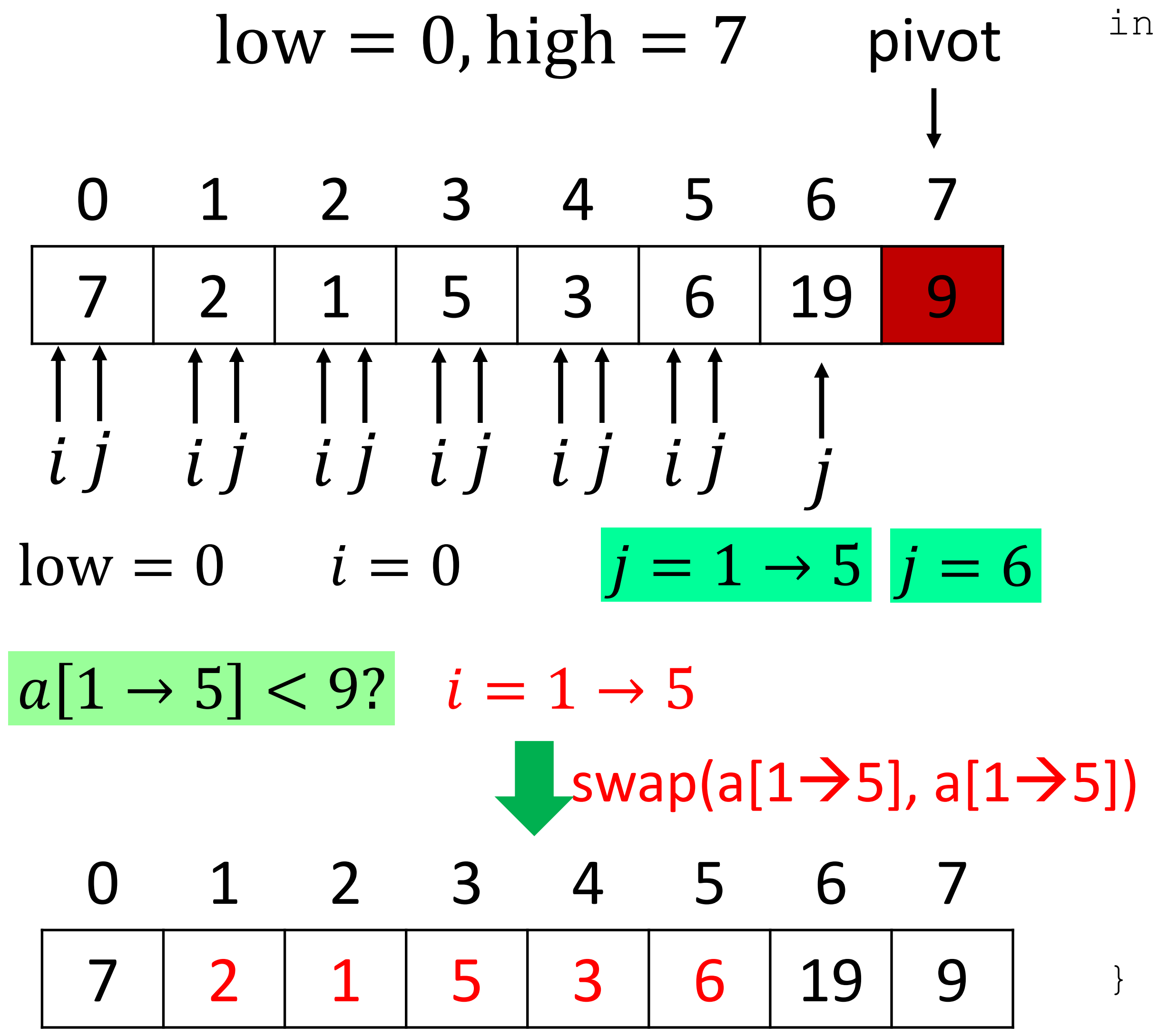
fit@hcmus

Illustration for partition function



```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {   TRUE
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

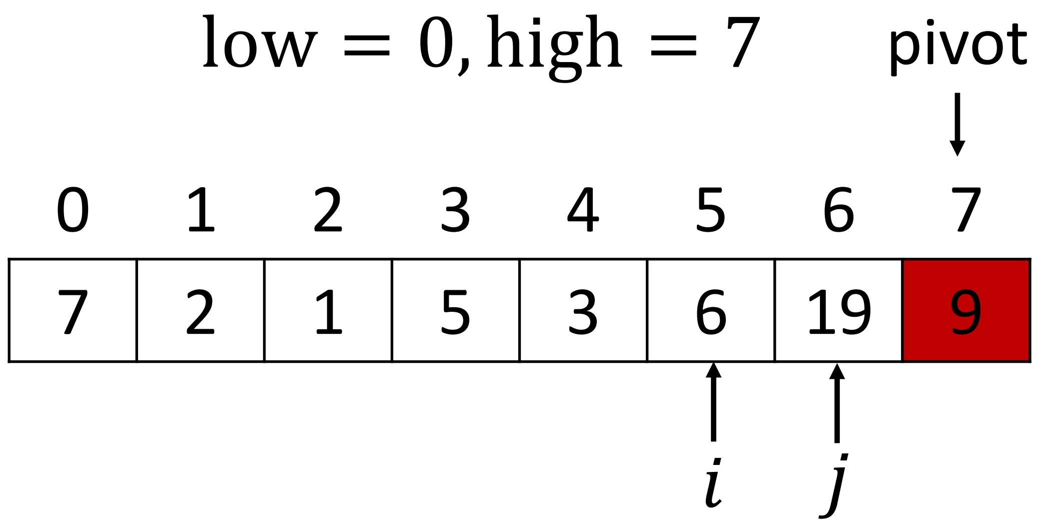
Illustration for partition function



```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {   TRUE
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

fit@hcmus

Illustration for partition function

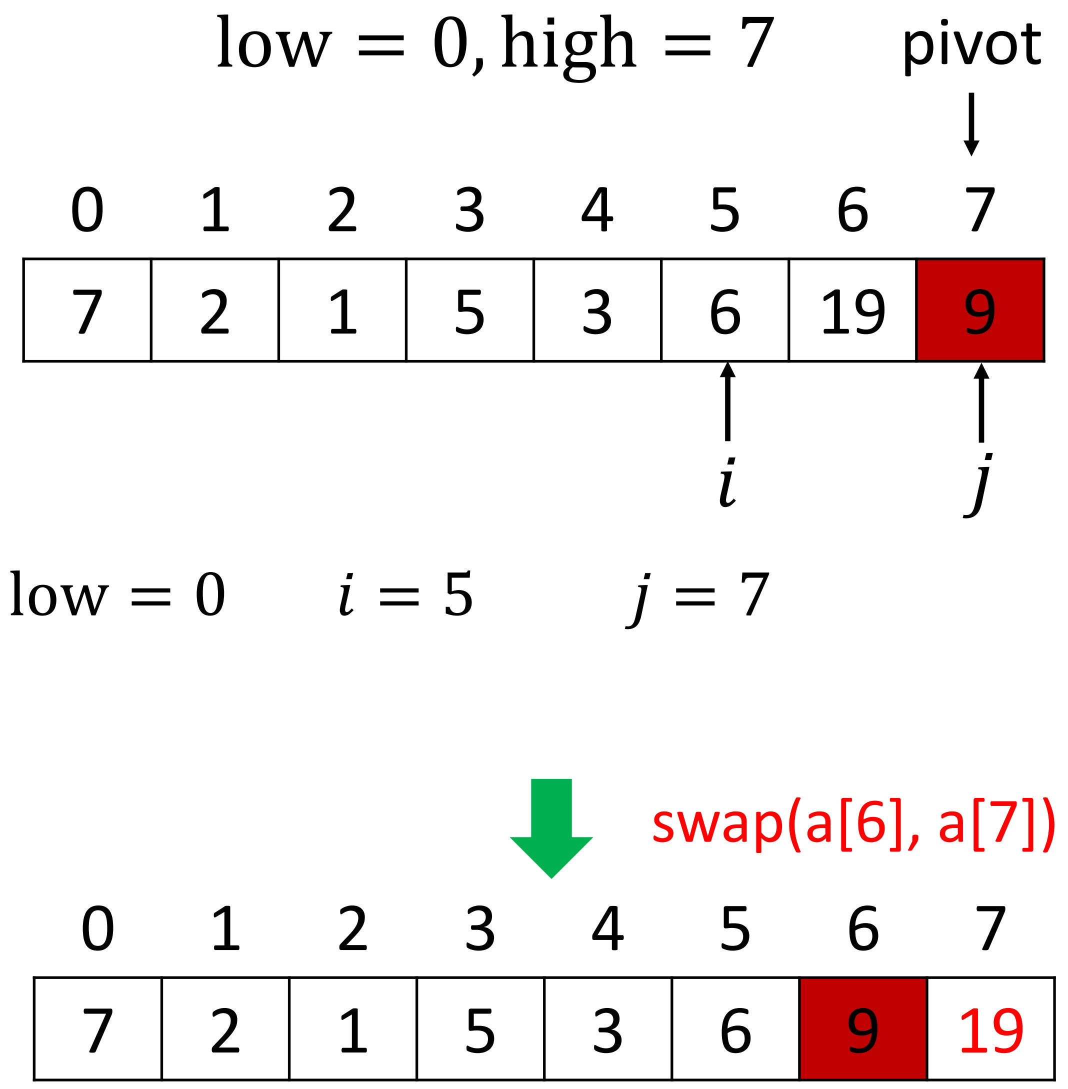


$\mathrm{low} = 0 \quad i = 5 \quad j = 6$

$a[6] < 9?$ $\quad j = 7$

```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements
                                  FALSE
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {  FALSE
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

Illustration for partition function



```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements
                            FALSE
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {  FALSE
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

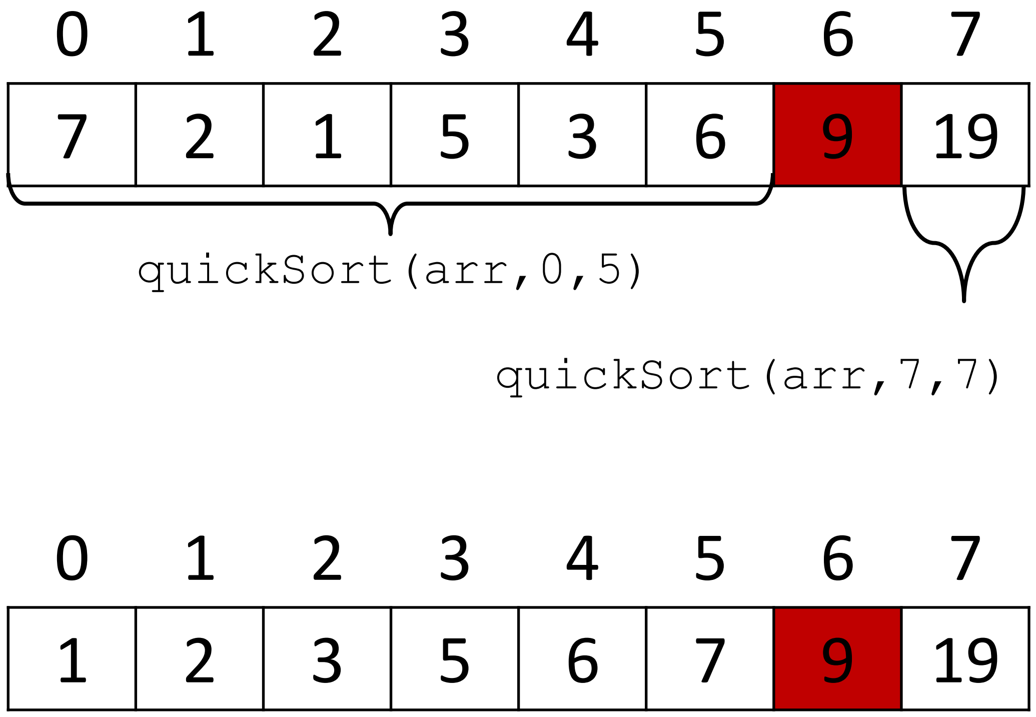
return 6

Putting all together

```
void quickSort(vector<int>& arr, int low,
   int high) {
     if (low < high) {
          int pivotIndex = partition(arr,
   low, high);
          quickSort(arr, low, pivotIndex-
   1);  // Sort left part
          quickSort(arr, pivotIndex + 1,
   high); // Sort right part
     }
}
```

6

$$low = 0, high = 7$$



Analysis: correctness

By math induction on the size n of the list.

1. Basis. If $n = 1$, the algorithm is correct.
2. Hypothesis. It is correct on lists of size smaller than n.
3. Inductive step. After positioning, the pivot p at position i for $i \in \{1, \dots, n-1\}$, splits the array of size n into the left subarray size i and the right subarray of size $n-1-i$.
    1. Elements of the left subarray are not greater than p.
    2. Elements of the right subarray are not smaller than p.
    3. By the induction hypothesis, both the left and right subarrays are sorted correctly.
    4. Therefore, the whole list of size n is sorted correctly.

Complexity analysis: general bound (1/2)

1. Let $T(n)$ be the number of comparisons done by quickSort, where n is the length of the input array.
2. We have

$T(n) = T(q) + T(n-1-q) + \Theta(n)$, for $q = 1, \dots, n-2$.

$$\Rightarrow \sum_{q=1}^{n-2} T(n) = \sum_{q=1}^{n-2} \big(T(q) + T(n-1-q) + \Theta(n)\big)$$

$$\Leftrightarrow (n-2)T(n) = (n-2)\Theta(n) + \sum_{q=1}^{n-2} T(q) \le (n-2)\Theta(n) + (n-2)T(n-2)$$

$$\Leftrightarrow T(n) \le T(n-2) + \Theta(n)$$

Complexity analysis: general bound (2/2)

$$\Rightarrow \sum_{q=1}^{n-2} T(n) = \sum_{q=1}^{n-2}\big(T(q) + T(n-1-q) + \Theta(n)\big)$$

$$\Leftrightarrow (n-2)T(n) = (n-2)\Theta(n) + \sum_{q=1}^{n-2} T(q) \le (n-2)\Theta(n) + (n-2)T(n-2)$$

$\Leftrightarrow T(n) \le T(n-2) + \Theta(n) = T(n-2) + cn$, for some positive $c > 0$

$$\Rightarrow \sum_{i=3}^{n} T(i) \le \sum_{i=3}^{n} T(i-2) + ci$$

$$\Leftrightarrow T(n) + T(n-1) \le T(1) + T(2) + \sum_{i=3}^{n} ci \le 0 + 1 + c \cdot \frac{n(n-1)}{2} - 3c$$

$$\Rightarrow 2T(n-1) \le T(n) + T(n-1) \le 1 + c \cdot \frac{n(n-1)}{2} - 3c$$

$\Rightarrow T(n) = O(n^2).$

Complexity analysis: average-case bound (1/4)

- Let $\{b_1, \dots, b_n\}$ be the increasingly sorted array of $\{a_1, \dots, a_n\}$.
- For $i < j$, let X_{ij} be a random variable that takes on the value 1 if there is a comparison between b_i and b_j at any time of the course of the algorithm, and 0 otherwise.

Complexity analysis: average-case bound (2/4)

- The total number of comparisons is

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

- Therefore, we get

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\left(X_{ij} = 1\right)$$

because X_{ij} only takes two values 0 and 1.

Complexity analysis: average-case bound (3/4)

- Hence all we need to do is compute the probability that two elements b_i and b_j are compared.
- Consider the ordered set $Y^{ij} = \{b_i, \dots, b_j\}$. b_i and b_j are compared iff b_i (or b_j) is selected as a pivot. Otherwise, these are separated into two sublists and will never be compared.
- Since our pivots are chosen independently and uniformly at random from each sublist, it follows that, the first time a pivot is chosen from Y^{ij}, it is equally likely to be any element from this set.

Complexity analysis: average-case bound (4/4)

- Therefore, $\Pr(X_{ij} = 1) = \frac{2}{j-i+1}$.
- Then

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(X_{ij} = 1) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k}$$

$$= 2 \sum_{i=2}^{n} \frac{n-i+1}{i} = -2(n-1) + 2(n+1) \sum_{i=2}^{n} \frac{1}{i}$$

$= 2(n+1)(H_n - 1) - 2(n-1) = O(n \ln n)$, where $H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \Theta(1)$.

Analysis

- Quick Sort is slow when the array is sorted and we choose the first element as the pivot.
- Although the worst case behavior is not so good, its average case behavior is much better than its worst case.
- Quick Sort is one of best sorting algorithms using key comparisons.

Complexity in **average**

		Design	**Run time**				**Space**	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$				Yes	No
	Insertion sort	Det.	$O(n^2)$				Yes	Yes
	Heap sort	Det.	$O(n \log n)$	$O(\log n)$			Yes	No
	Merge sort	Det.	$\Theta(n \log n)$		$O(n)$		No	Yes
	Quick sort	Rnd.	$O(n \log n)$				Yes	No
Non-comparison-based	?	?	?				?	?

Outline

1. Selection Sort
2. Insertion Sort
3. Heap Sort
4. Merge Sort
5. Quick Sort
6. Radix Sort

Complexity in **average**

		Design	Run time				Space	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$	$O(\log n)$	$O(n)$		Yes	No
	Insertion sort	Det.	$O(n^2)$				Yes	Yes
	Heap sort	Det.	$O(n \log n)$				Yes	No
	Merge sort	Det.	$\Theta(n \log n)$				No	Yes
	Quick sort	Rnd.	$O(n \log n)$				Yes	No
Non-comparison-based	Radix sort	Det.	$O\big(d(n+k)\big)$	$O(\log n)$			No	Yes

d is the number of digits and k is the range of digits.

Idea

- Radix Sort is a non-comparative, integer-based sorting algorithm that sorts numbers by processing individual digits.
- It works by sorting numbers from the least significant digit (LSD) to the most significant digit (MSD) or vice versa.
  - 72153619.
- The algorithm is efficient when sorting numbers with a fixed number of digits.

Algorithm

- Treats each element as a character string.

- Repeat (*for all characters from the rightmost to the leftmost*)
  - Groups elements according to their rightmost character.
  - Put these groups into order with respect to this rightmost character.
  - Combine all the groups.
  - Move to the next left position.

- At the end, the sorting process will be completed.

Example

7	2	1	5	3	6	19	9

$d = 2$ is the number of digits and $k = 10$ is the range of digits.

1. First Pass: The rightmost digit

0	1	2	3	4	5	6	7	8	9
	01	02	03		05	06	07		19
									09

Combine after first pass: $\{01, 02, 03, 05, 06, 07, \{19, 09\}\}$

Example

7	2	1	5	3	6	19	9

$d = 2$ is the number of digits and $k = 10$ is the range of digits.

2. Second Pass: The rightmost digit

$\{01,02,03,05,06,07,\{19,09\}\}$

0	1	2	3	4	5	6	7	8	9
01, 02, 03, 05, 06, 07, 09,	19								

Combine after second pass: $\{01, 02, 03, 05, 06, 07, 09, 19\}$

Pseudocode

```
RadixSort(A[], n, d) // sort n d-digit integers in the array A
   for (j = d down to 1) {
         Initialize k=10 groups to empty
         Initialize a counter for each group to 0
         for (i = 0 through n-1) {
               k = $j^{th}$ digit of A[i]
               Place A[i] at the end of group k
               Increase counter for group k by 1
         }
         Replace the items in A with all the items in group 0, group 1, …, group
   9 in orders.
   }
```

Source code (1/2)

```
// Function to perform Counting Sort based on the given digit (exp)
void countingSort(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n);  // Output array
    int count[10] = {0};    // Count array for digits (0-9)

    // Count occurrences of each digit at 'exp' place
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Update count array to store actual positions
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build output array in sorted order
    for (int i = n - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    // Copy sorted numbers back into original array
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}
```

Source code (2/2)

```
// Function to perform Radix Sort
void radixSort(vector<int>& arr) {
    // Find the maximum number to determine the number of digits
    int maxNum = *max_element(arr.begin(), arr.end());

    // Apply counting sort for each digit position
    for (int exp = 1; maxNum / exp > 0; exp *= 10)
        countingSort(arr, exp);
}
```

Analysis: Correctness

- The correctness of Radix Sort is ensured by two key properties:

  1. Stable Sorting of Digits.
  2. Sorting in Multiple Passes (Least to Most Significant Digit).

Stable Sorting of Digits

- A stable sort preserves the relative order of elements with the same digit value.
- This ensures that when sorting higher place values, numbers that were previously ordered remain ordered.

Correct Order in Multiple Passes

- Radix Sort processes numbers digit by digit, starting from the least significant digit (LSD) to the most significant digit (MSD).
- Since each pass is stable, numbers remain in the correct relative order after every pass.
- When the final pass is over, the order is determined by the most significant digits after all passes.

Inductive Proof

- Base Case: After the first iteration (sorting by the least significant digit), the array is sorted according to the least significant digit.
- Inductive Hypothesis: Assume that after the k-th iteration, the array is sorted according to the k least significant digits.
- Inductive Step: We need to show that after the $(k+1)$-th iteration, the array is sorted according to the $k+1$ least significant digits.
- When counting sort is applied to the $(k+1)$-th digit, it sorts the array based on this digit.
- Because counting sort is stable, elements with the same $(k+1)$-th digit maintain their relative order from the previous iterations (which were sorted according to the k least significant digits).
- Therefore, after the $(k+1)$-th iteration, the array is sorted according to the $k+1$ least significant digits.
- **Conclusion**: By induction, after all iterations (processing all digits), the array is completely sorted.

Analysis: complexity

- Best, Average, Worst Case: $O\big(d(n+k)\big)$.
  - n: number of elements.
  - d is the number of digits.
  - k is the range of digits.

Summary

		Design	Run time				Space	
			Run time	Search	Insert	Delete	In-place	Stable
Unsorted	Array	Det.	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Comparison-based	Selection sort	Det.	$O(n^2)$	$O(\log n)$	$O(\log n)$ for list $O(n)$ for array		Yes	No
	Insertion sort	Det.	$O(n^2)$				Yes	Yes
	Heap sort	Det.	$O(n \log n)$				Yes	No
	Merge sort	Det.	$\Theta(n \log n)$				No	Yes
	Quick sort	Rnd.	$O(n \log n)$				Yes	No
Non-comparison-based	Radix sort	Det.	$O(d(n+k))$	$O(\log n)$			No	Yes

d is the number of digits and k is the range of digits.

Exercises

Given the following list of integers:

a) 13, 4, 2, 7, 34, 1.

b) 1, 56, 3, 6, 9, 8, 30, 24, 17, 7.

1. Using the Selection Sort and Insertion Sort, demonstrate the steps to sort in ASCENDING ORDER.
2. Using the Selection Sort and Insertion Sort, demonstrate the steps to sort in DECENDING ORDER.
3. Applying the Heap Construction (heapify) algorithm, demonstrate the steps to create a max-heap from the above list.
4. Using the Merge Sort, demonstrate the steps to sort the list of integers in ASCENDING ORDER.

fit@hcmus

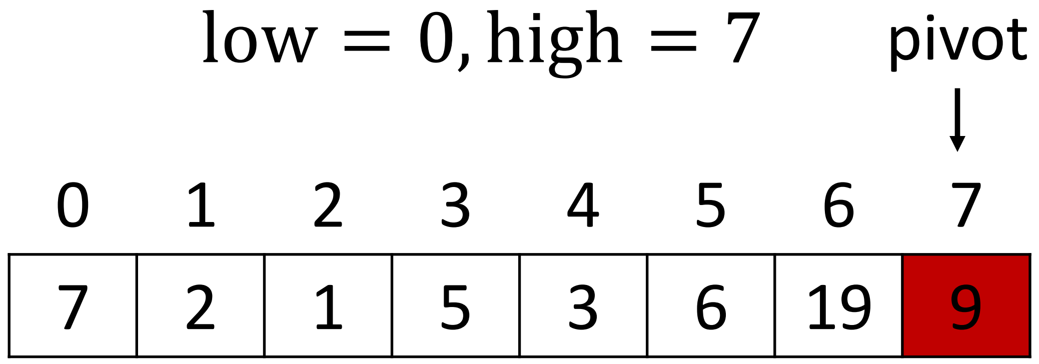
Q & A

Source code: Descending Sorting

```
void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex-1);  // Sort left part
        quickSort(arr, pivotIndex + 1, high); // Sort right part
    }
}
```

```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // Choose last element as pivot
    int i = low - 1; // Index for smaller elements

    for (int j = low; j < high; j++) {
        if (arr[j] > pivot) {
            i++;
            swap(arr[i], arr[j]); // Swap smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place pivot in the correct position
    return i + 1; // Return pivot index
}
```

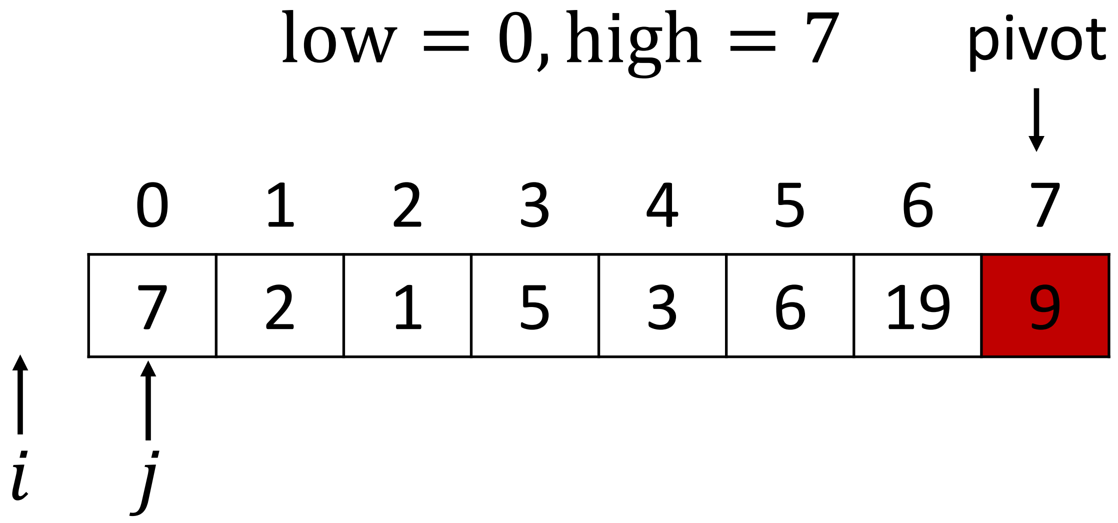
Illustration for partition function



```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // Choose last element as pivot
    int i = low - 1; // Index for smaller elements

    for (int j = low; j < high; j++) {
        if (arr[j] > pivot) {
            i++;
            swap(arr[i], arr[j]); // Swap smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place pivot in the correct position
    return i + 1; // Return pivot index
}
```

fit@hcmus

Illustration for partition function

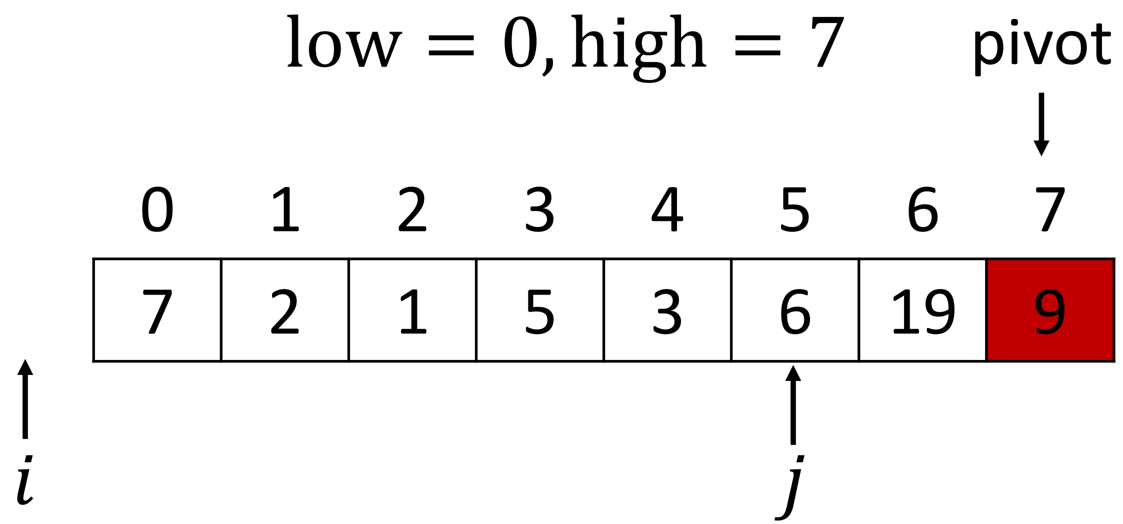


$\text{low} = 0 \quad i = -1 \quad j = 0$

$a[0] > 9?$ $j = 1$

```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements

    for (int j = low; j < high; j++) {
        if (arr[j] > pivot) {   FALSE
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

Illustration for partition function



$\text{low} = 0 \quad i = -1 \quad j = 0$

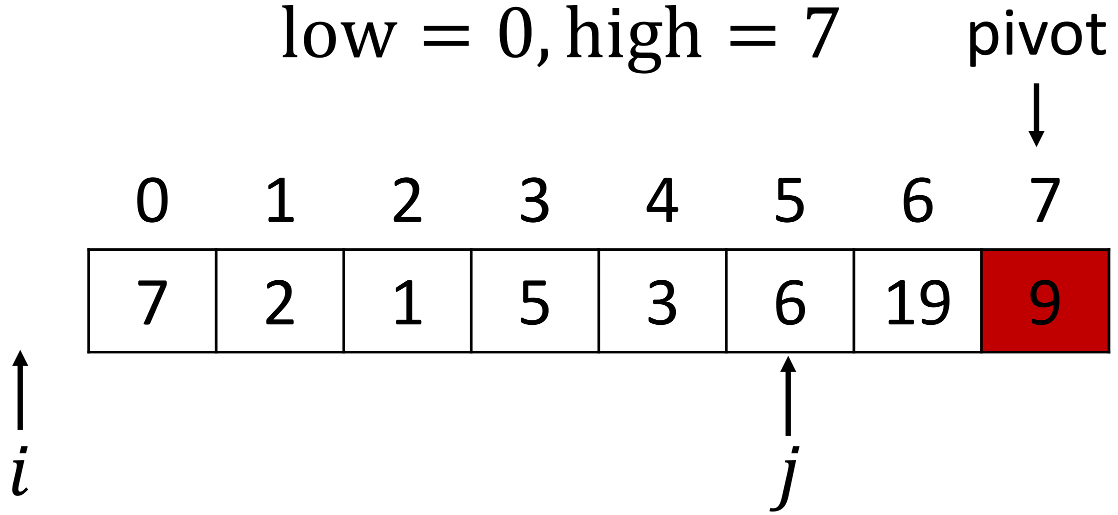
$a[0] > 9?$ $j = 1 \to 5$

```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements

    for (int j = low; j < high; j++) {
        if (arr[j] > pivot) {
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

FALSE

Illustration for partition function

$\text{low} = 0, \text{high} = 7$



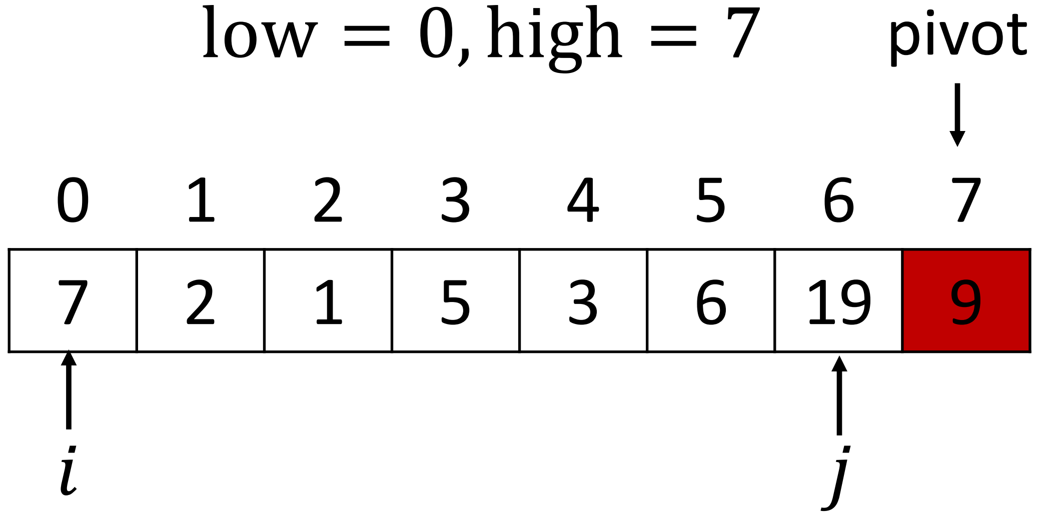
$\text{low} = 0$ $\quad i = -1$ $\quad j = 1 \rightarrow 5$

$a[1 \rightarrow 5] > 9?$ $\quad j = 6$

```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {   FALSE
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

Illustration for partition function

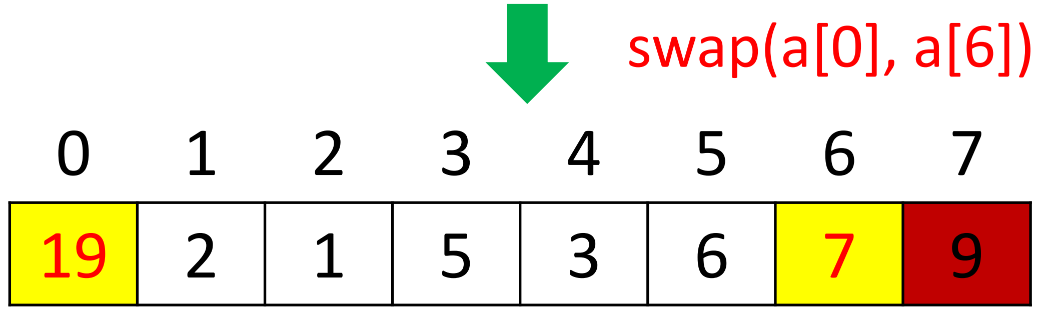
$\text{low} = 0, \text{high} = 7$



$\text{low} = 0 \quad i = -1 \quad j = 6$

$a[6] > 19?$ $\quad i = 0$

swap(a[0], a[6])

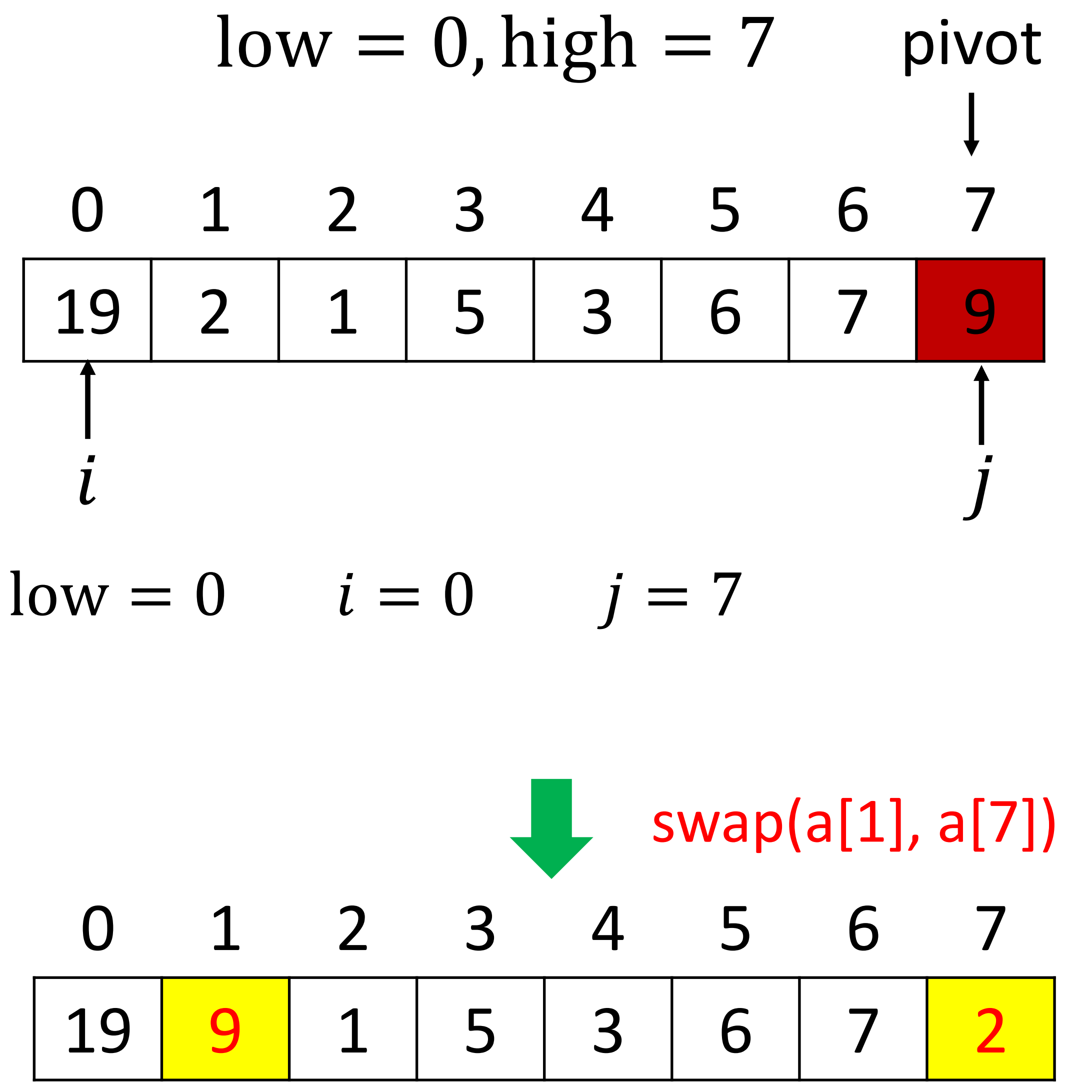


```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements

    for (int j = low; j < high; j++) {
        if (arr[j] > pivot) {
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

$j = 7$

TRUE

Illustration for partition function

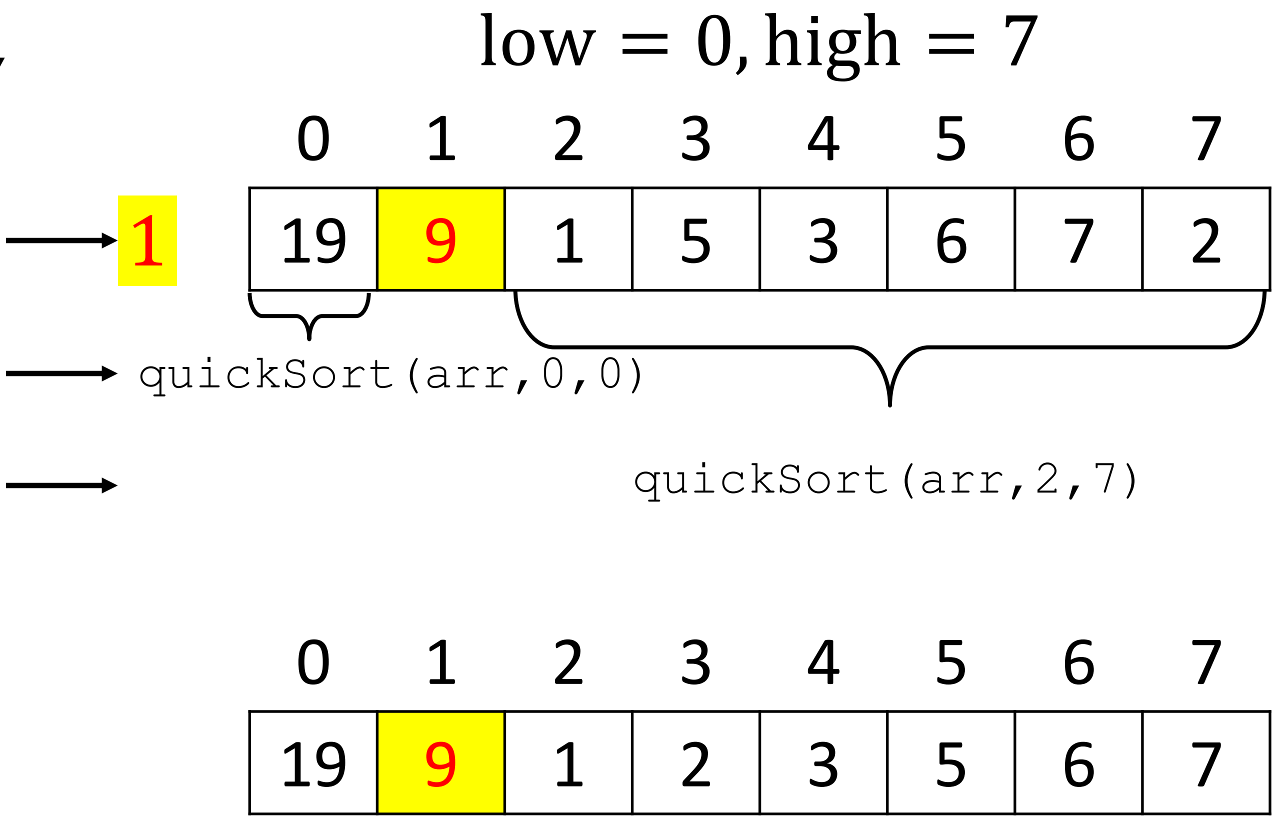


```
int partition(vector<int>& arr, int low, int
   high) {
    int pivot = arr[high]; // Choose last
   element as pivot
    int i = low - 1; // Index for smaller
   elements
                                FALSE
    for (int j = low; j < high; j++) {
        if (arr[j] > pivot) {
            i++;
            swap(arr[i], arr[j]); // Swap
   smaller elements to the left
        }
    }
    swap(arr[i + 1], arr[high]); // Place
   pivot in the correct position
    return i + 1; // Return pivot index
}
```

return 1

Putting all together

```
void quickSort(vector<int>& arr, int low,
   int high) {
     if (low < high) {
          int pivotIndex = partition(arr,
   low, high);
          quickSort(arr, low, pivotIndex-
   1);  // Sort left part
          quickSort(arr, pivotIndex + 1,
   high); // Sort right part
     }
}
```



0	1	2	3	4	5	6	7
19	9	1	5	3	6	7	2

0	1	2	3	4	5	6	7
19	9	1	2	3	5	6	7