[CSED233-01] Data Structure

Sorting

Jaesik Park

POSTECH

Sorting

- Given a collection of elements
  - Each element is a pair (*key*, *info*)

- Reordering/arranging the elements in a certain order
  - Mostly
    - Numerical order
    - Lexicographical order (a generalization of the alphabetical order)

  - Linear (or total) ordering on keys:
    - Trichotomy (three categories): For any keys a and b,
      exactly one of $a < b$, $a = b$, or $a > b$ is true
    - Transitivity: For any a, b, and c,
      if $a < b$ and $b < c$, then $a < c$

Types of Sorting: Memory Usage

- Internal (In-memory) sort
  - Appropriate for sorting a collection of elements that fit in main memory
  - Simple, but relatively slow - $O(n^2)$
    - *bubble sort, selection sort, insertion sort*
  - Considerably better - $O(n \log n)$ on average
    - *heap sort, quick sort, merge sort*

- External sort
  - Too large collection of elements to fit in main memory, so the records must reside in external memory
  - Based on *merge sort*

Types of Sorting: Comparison

- Comparison sorts
  - Use only the relation among the keys (pair-wise comparison)
    - *bubble / selection / insertion sort*
    - *heap / merge / quick sort*

- Non-comparison sorts
  - Use some properties of keys
    - *bucket sort* (examines bits of keys)
    - *radix sort* (examines individual bits of keys)
    - *counting sort* (indexes using key values)

Types of Sorting: Stability

- Stable sort
  - Retain the original relative ordering of elements with duplicate keys
  - (e.g.) For given pairs (4, a) (3, y) (3, x) (5, b)
    - (3, y) (3, x) (4, a) (5, b) : order maintained
    - (3, x) (3, y) (4, a) (5, b) : order changed

Bubble Sort

- On path *k*, the *k*-th lowest key rises to *k*-th position
  - Iteratively swap the adjacent items if the below item has a lower key value
  - If there were no swap in the current iteration, the array is sorted

initial	k = 1	k = 2	k = 3	k = 4	k = 5
42	13	13	13	13	13
20	42	14	14	14	14
17	20	42	17	17	17
13	17	20	42	20	20
28	14	17	20	42	28
14	28	28	28	28	42

- $O(n^2)$ in the average & worst cases
  - Only useful for a small collection ($n < 100$), $O(n^2)$ swaps
- $O(n)$ in the *best case* (over an already-sorted list)

Selection Sort

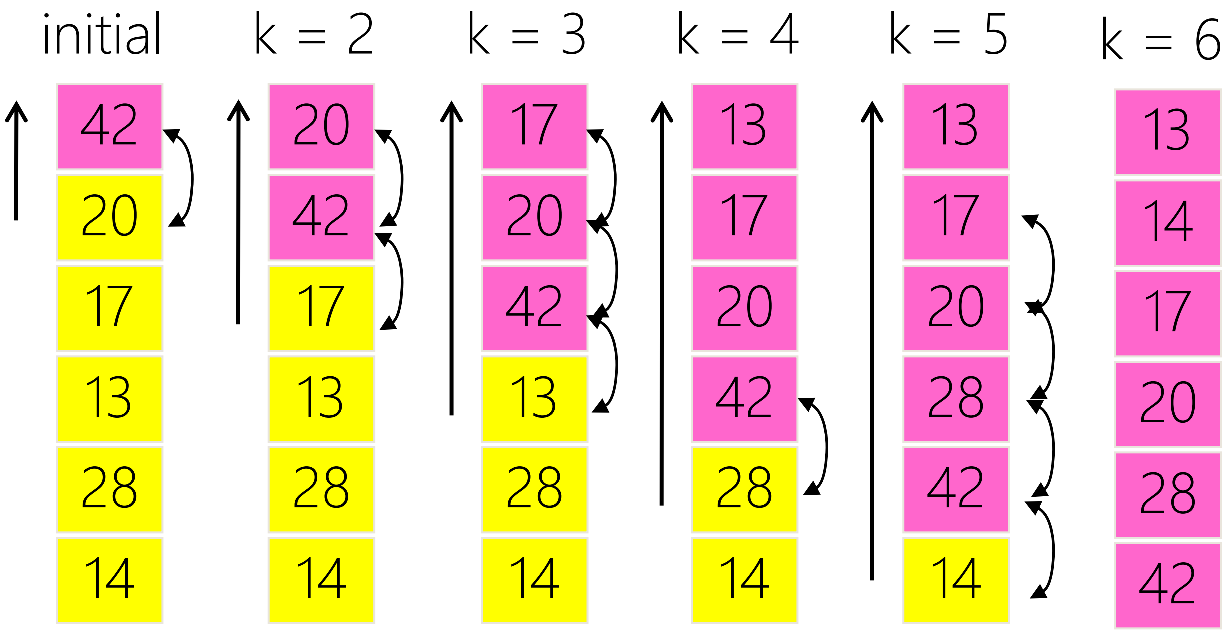
- On path *k*, select the *k*-th lowest key & swap with A[*k*]
  - For each iteration, find the minimum element from the unsorted part and putting it at the beginning
  - At most one swapping happens for each iteration

initial	k = 1	k = 2	k = 3	k = 4	k = 5
42	13	13	13	13	13
20	20	14	14	14	14
17	17	17	17	17	17
13	42	42	42	20	20
28	28	28	28	28	28
14	14	20	20	42	42

- $O(n^2)$ in the best, average, and worst cases
- Fewer (n-1) *record swaps* than Bubble sort

Insertion Sort

- On path k, A[k] is inserted at the correct position within an already sorted list A[1], A[2], …, A[k-1]
- Values from the unsorted part are placed at the correct position in the sorted part

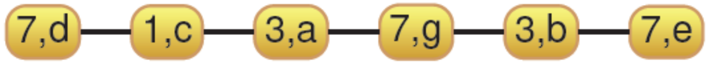


- $O(n^2)$ in the average & worst cases
  - $\Theta(n^2)$ swaps
- $O(n)$ in the *best case* (if keys begin in sorted order)

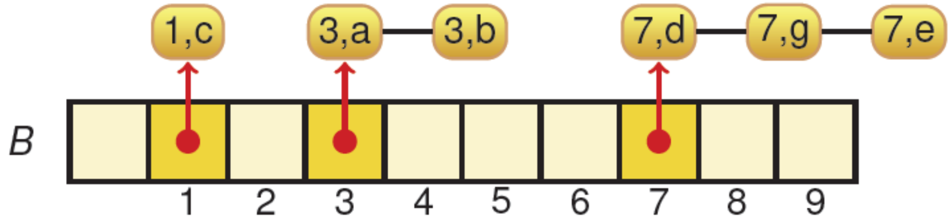
Bucket Sort

- Non-comparison sort
  - Phase 1: scattering keys into a number of buckets
    - If you need to sort a single bucket list, sort each non-empty bucket (either recursively or using a different sorting algorithm, e.g., insertion sort)
  - Phase 2: gathering
    - Visit the buckets in order & empty them into the original list
- Simple example:
  - A list of n (key, info) pairs with key range [0, N-1]

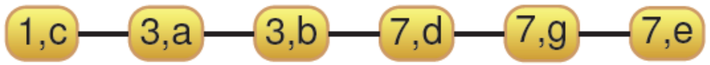
Bucket Sort



- Phase 1: scattering into buckets → O(n)



- Phase 2: Gathering → O(n + N)



- O(n + N) time in the average case
- Efficient
  - if keys come from a small interval [0, N - 1]

Bucket Sort

- What if your keys are floating numbers?
  - <0.78, 0.12, 0.45, 0.26, 0.36, 0.48, 0.11 >

- Can you apply bucket sort here as well?

Mergesort

- Divide-and-Conquer (DQ) algorithm
  - *Split* a list of elements into two equal sublists
  - Sort each of the sublists recursively
  - *Combine* the two sorted sublists into one sorted list
    - Using "*merge*" process

- Complexity: O(n log n)
- Usually implemented *non-recursively*

Recursive Mergesort

```
Merge-sort (L, n) {
   if n = 1 then
      return (L)
   else  begin
      break L into two halves L1 and L2 of length n/2;
      return (merge (Merge-sort(L1, n/2), Merge-sort(L2, n/2)));
   end
}
```

Merge Two Sorted Lists

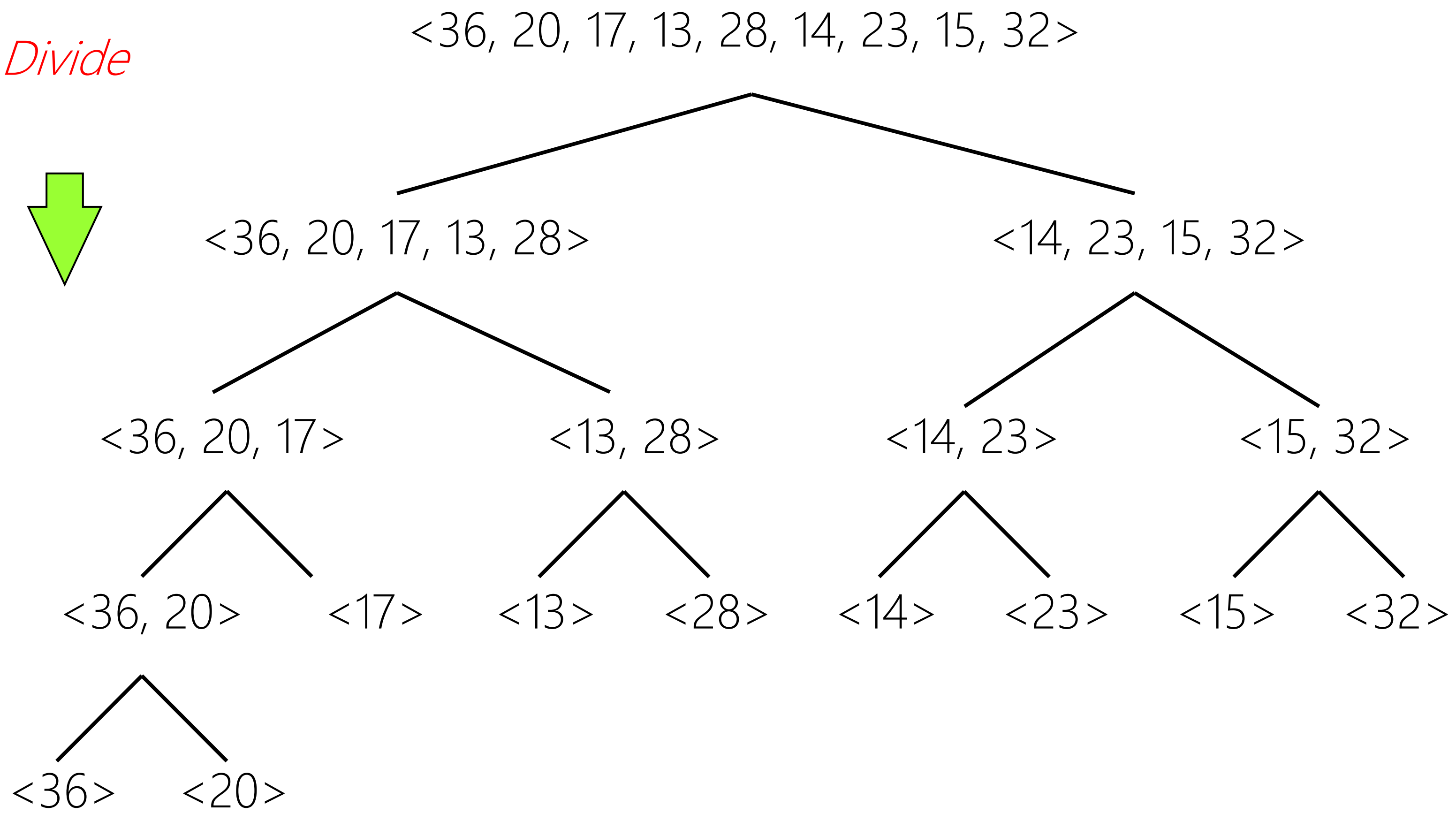
- Given two sorted lists
  - A = <13, 17>
  - B = <14, 15, 23, 28>
  - C = < >: the output list

Merge Two Sorted Lists

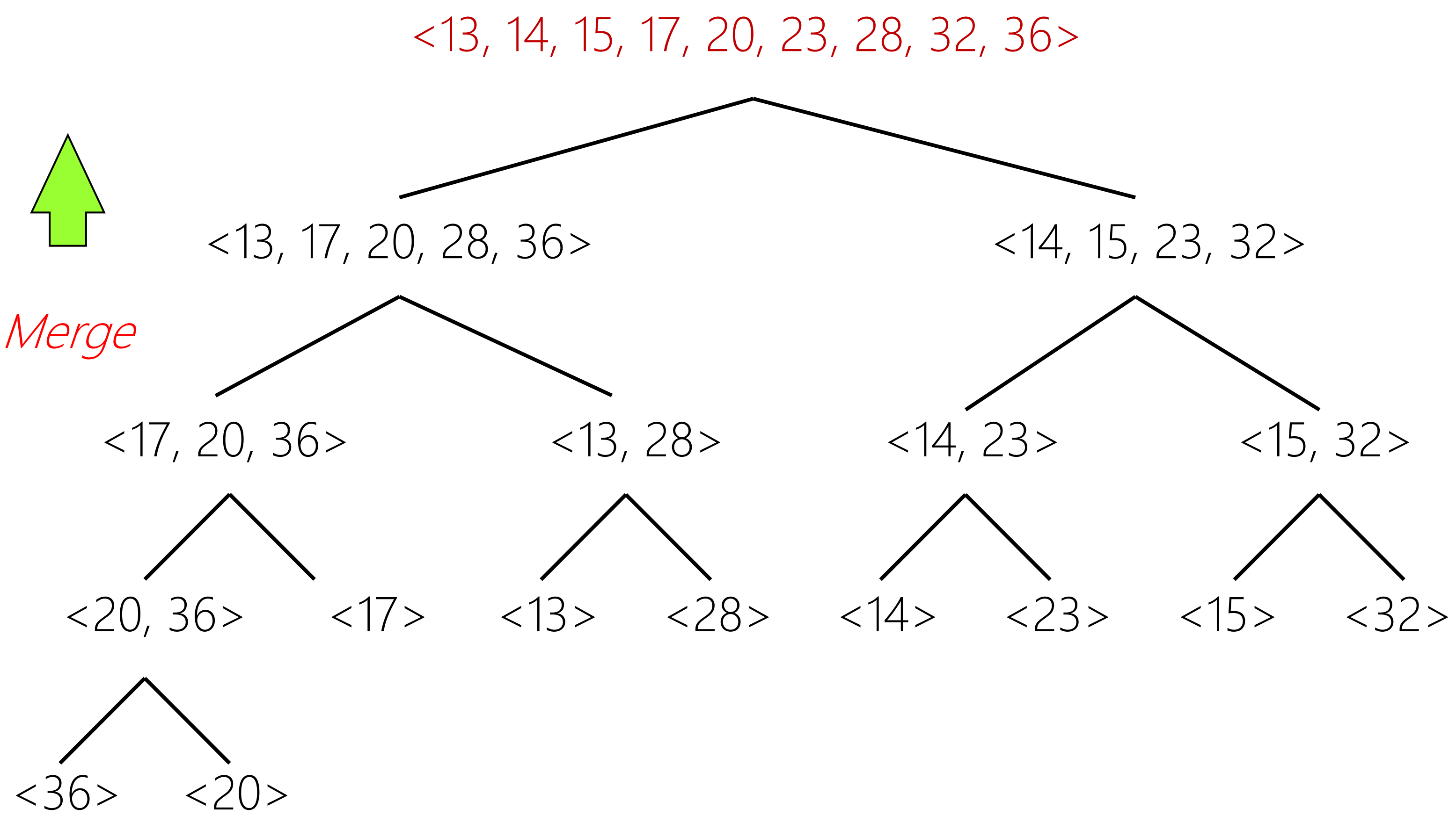
- Compare the first elements of A and B, and append the smaller into the output list C
    - Step 1
        - A = <17>
        - B = <14, 15, 23, 28>
        - C = <13>
    - Step 2
        - A = <17>
        - B = <15, 23, 28>
        - C = <13, 14>
    - Step 3
        - A = <17>
        - B = <23, 28>
        - C = <13, 14, 15>
    - Step 4
        - A = < >
        - B = <23, 28>
        - C = <13, 14, 15, 17>

- When one of A and B becomes empty, append the other list to C
- Total time: $O(n + m)$, where n and m are the # of elements initially in A and B, respectively

Mergesort (Downward Pass)



Mergesort (Upward Pass)



Time Complexity

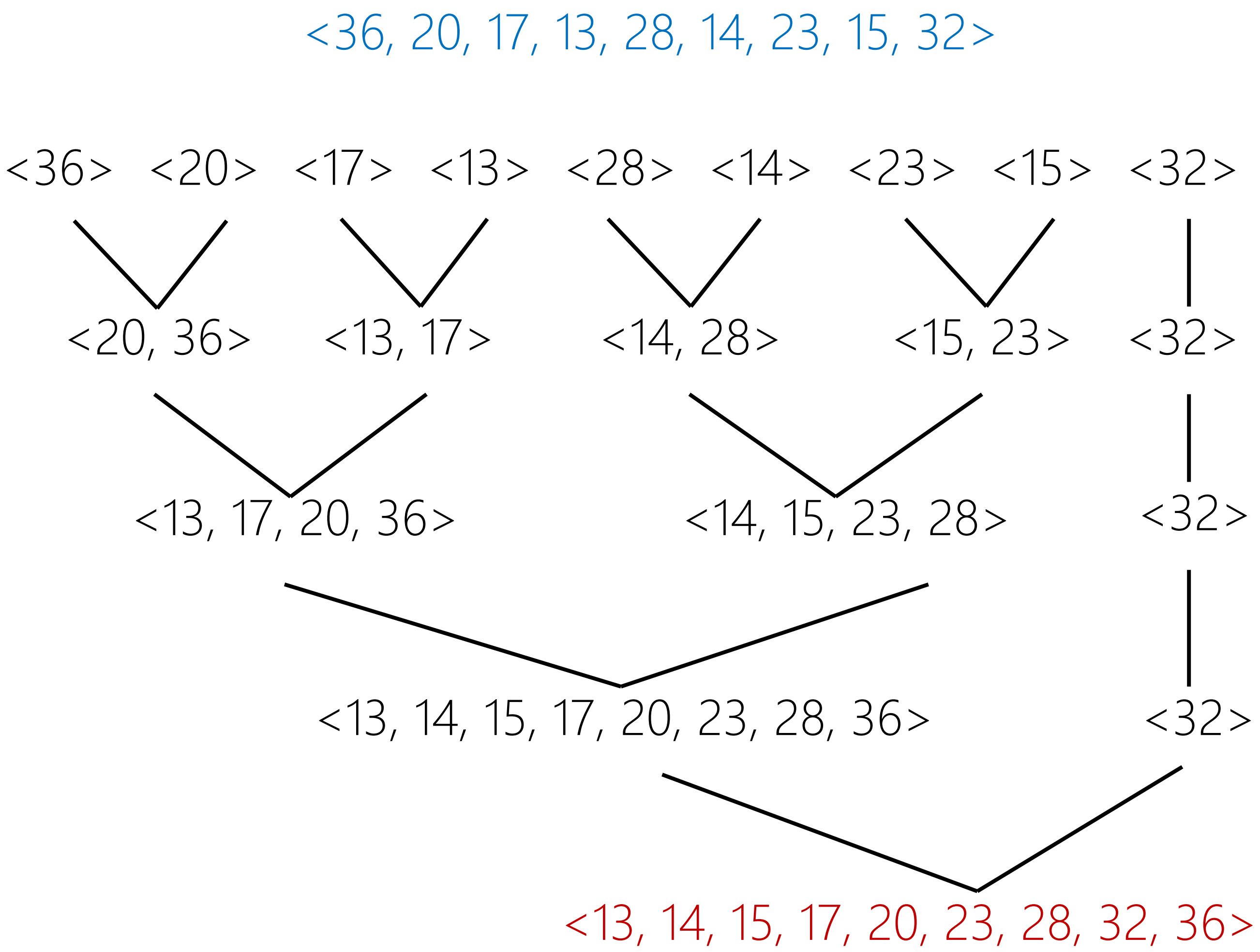
- Recursion tree
  - # of leaf nodes: n
  - # of non-leaf nodes: n-1
- Downward pass over the recursion tree
  - O(1) time at each node
  - O(n) total time at all nodes
- Upward pass over the recursion tree
  - O(n) time to merge at each level that has a non-leaf node
  - O(log n) levels

➔ Total time = O(n log n)

Non-recursive Version

- Eliminate downward pass
- Start with the sorted segments of size 1 and do pairwise merging of these sorted segments as in the upward pass

Non-recursive Mergesort



Complexity

- *Mergesort* is slower than *Insertion Sort* when approximately $n \leq 15$
  - So, define a *small instance* to be an instance with $n \leq 15$
  - And sort small instances using *Insertion Sort*

- Start with segment size $= 15$

Quicksort

- Divide phase
  - Select a *pivot* element from out of the *n* elements
  - *Partition* the elements into 3 groups
    - Left partition: key values < *pivot*
    - *Pivot* itself
    - Right partition: key values ≥ *pivot*
  - Sort the left & right partitions *recursively*

- Conquer phase
  - Answer is the sorted left partition, followed by the pivot and by the sorted right partition

Example (Partitioning)

A =

17	20	17	33	15	2	21	14

- Select the leftmost as the pivot (*pivot* = 17)
- Method 1: When another lists *L* and *R* are available
  - Scan *A* from left to right, appending elements (< pivot) to *L* and the others to *R*

L =

15	2	14

R =

20	17	33	21

17

- Sort *L* and *R* recursively

Quicksort

```
Quicksort(A) { // Sort A[1..n] in ascending order
   var list L, R
        if length(A) ≤ 1 return A
        select & remove pivot from array
        for each x in A do {
                if x < pivot then append x to L
                else append x to R
        }
        return concatenate(Quicksort(L), pivot,
                                   Quicksort(R))
}
```

Example (Partitioning)

A =	17	20	17	33	15	2	21	14

- Select the leftmost as the pivot (*pivot* = 17)
- Method 2: When another array *B* is available
  - Scan *A* from left to right, placing elements (< *pivot*) at the left end of *B* and the remaining at the right end of *B*

B =	15	2	14	17	21	33	17	20

- Sort the left and right groups recursively

Choice of Pivot in Many Ways

- Ideal case – choose the median key value
    - All the elements can be partitioned into two halves

- Use the first (or last) element
    - If the input is sorted, this will produce poor partitioning with all elements to one side of the pivot

- Pick an element at random
    - Using a random number generator is relatively expensive

- Select the middle element

Choice of Pivot in Many Ways

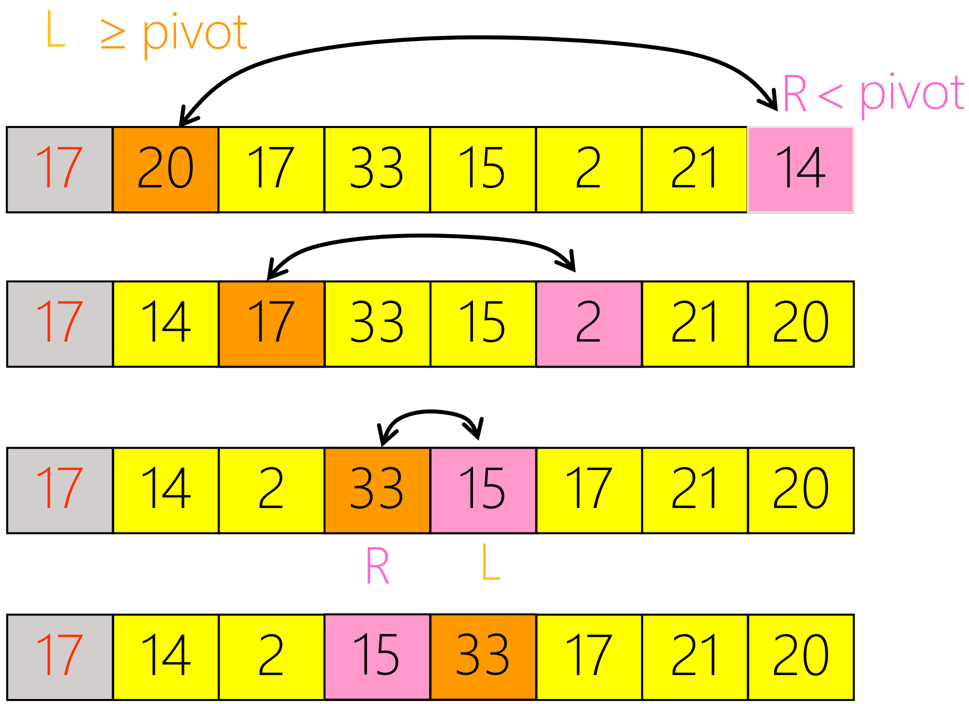
- Using median-of-three rule
  - From the three elements (first, middle, & last), select the one with median key
  - (e.g.) When sorting an array A[1:9]
    - Examine A[1], A[5], and A[9]
    - If they have keys {30, 2, <u>10</u>} → A[9]
    - If {<u>3</u>, 2, 10} → A[1]
    - If {35, <u>20</u>, 10} → A[5]

- Select the larger of the first two distinct elements
  - [in Aho83]

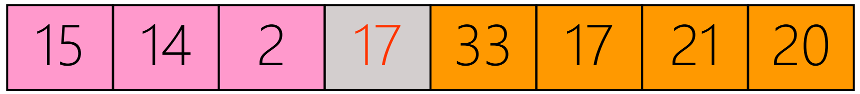
“In-Place” Partitioning

- Repeat
  - Find the leftmost element (L) ≥ pivot
  - Find the rightmost element (R) < pivot
  - Swap L & R if L is the left of R

Example (In-Place Partitioning)



- L is not to the left of R
- Thus, terminate process and swap pivot & R



Time Complexity

- To partition an array of n elements
  - $O(n)$ time
- Let $t(n)$ be the time needed to sort the n elements
  - $$t(n) = \begin{cases} c & (n = 0, 1) \\ t(|\text{left}|) + t(|\text{right}|) + d*n & (n > 1) \end{cases}$$

  where c, d : constant

- Best-case time when
  - |left| & |right| are equal (or differ by 1) at each partitioning step
  - $O(n \log n)$ time in the best & average cases

Complexity

- Worst-case time when
  - either |left| = 0 or |right| = 0 at each partitioning
  - the pivot is always the smallest element
  - the input is sorted and the leftmost element is chosen
  - $O(n^2)$ time in the worst case

- To improve performance
  - Define a small instance to be one with $n \leq 15$, and sort small instances using insertion sort

Comparing Quick & Heap Sorts

n	Quick sort		Heap sort		Insertion sort	
	Compare	Exchange	Compare	Exchange	Compare	Exchange
100	712	148	2,842	581	2,596	899
200	1,682	328	9,736	1,366	10,307	3,503
500	5,102	919	53,113	4,042	62,746	21,083

- Heap sort? (heapify!)
- Empirically, quick sort is considerably faster than heap sort
- However, quick sort should never be used in applications which require a guarantee of response time unless it is treated as an $O(n^2)$ algorithm

Quick sort vs. Merge sort

- Quick sort can be implemented with in-place operation
  - No additional memory is required as in Merge sort
  - Working on a same array increases access speed of the data by making use of cache coherence (something that you can learn from computer architecture)
- In many cases, quick sort can avoid $O(n^2)$ by choosing a right pivot

References

- Further reading list and references
  - https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee