Sorting

Lecture 20

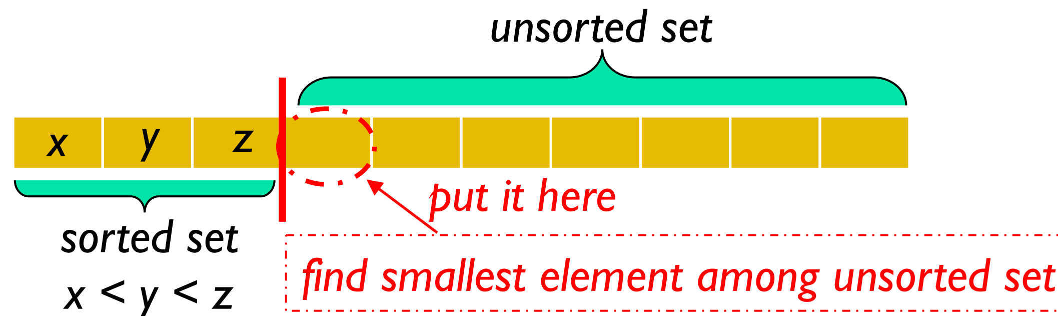
Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition, Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning

Efficient Sorting Algorithms: Heap Sort

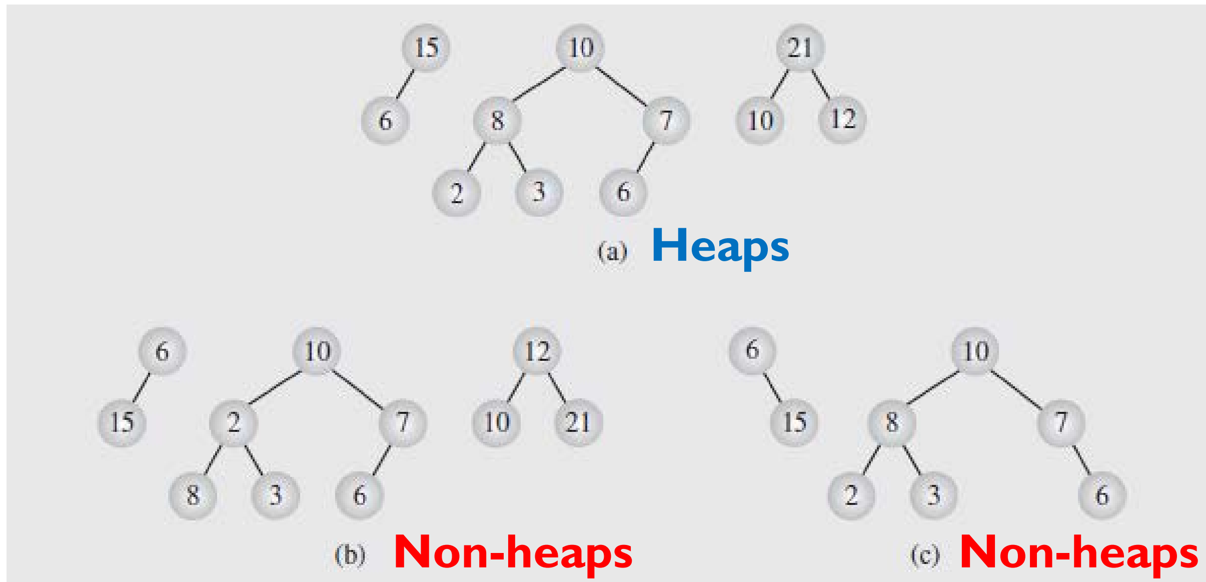
- Motivation of ***heap sort***
  - *selection sort*, fairly *inefficient*, **O(n^2)**
    - recall: selection sort finds the *smallest element* in the list and places it *first*, then the next smallest, etc.



    - relatively ***few*** moves of the data; ***many*** comparisons
  - if the *comparison portion* of the sort can be improved?
    - performance can likewise be improved

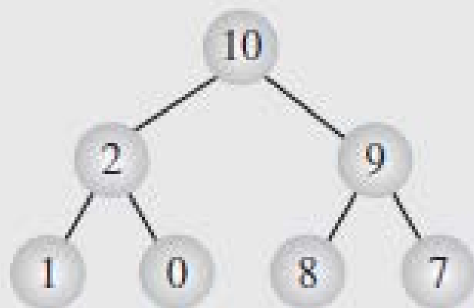
Efficient Sorting Algorithms: Heap Sort

- A ***heap,*** a special type of *binary tree*
  - the value of each node is **greater than or equal** to the values stored in its **children**
  - the tree is ***perfectly balanced***, and the leaves in the last level are **leftmost** in the tree

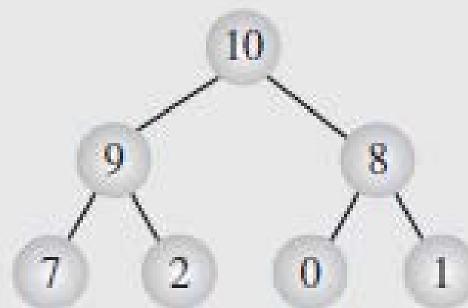


Efficient Sorting Algorithms: Heap Sort

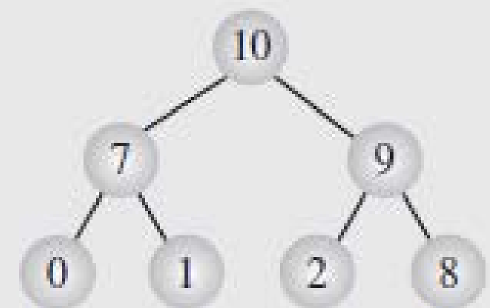
- A ***heap,*** a special type of *binary tree*
  - the value of each node is **greater than or equal** to the values stored in its **children**
  - the tree is ***perfectly balanced***, and the leaves in the last level are **leftmost** in the tree
- Different heaps constructed with the same elements [0 1 2 7 8 9 10]



(a)



(b)



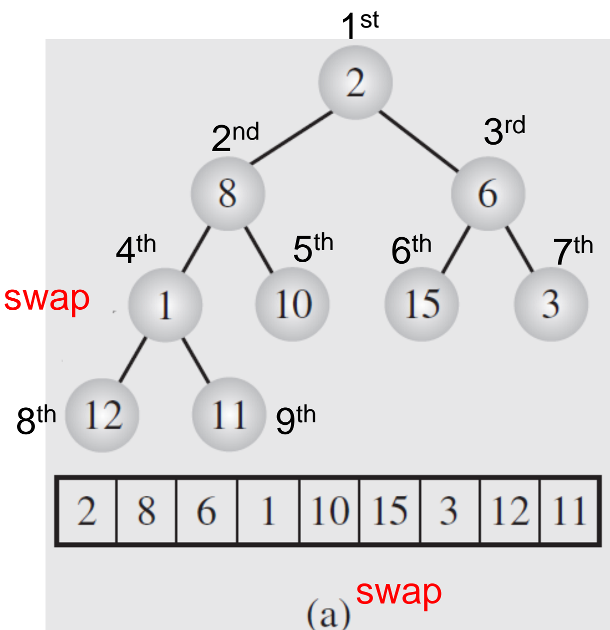
(c)

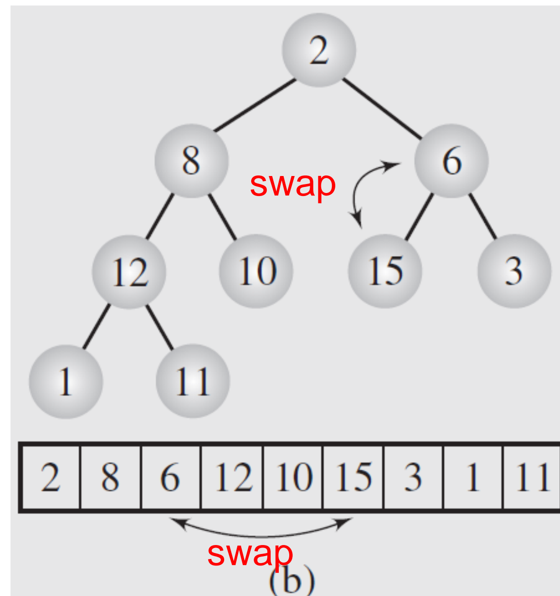
Efficient Sorting Algorithms: Heap Sort

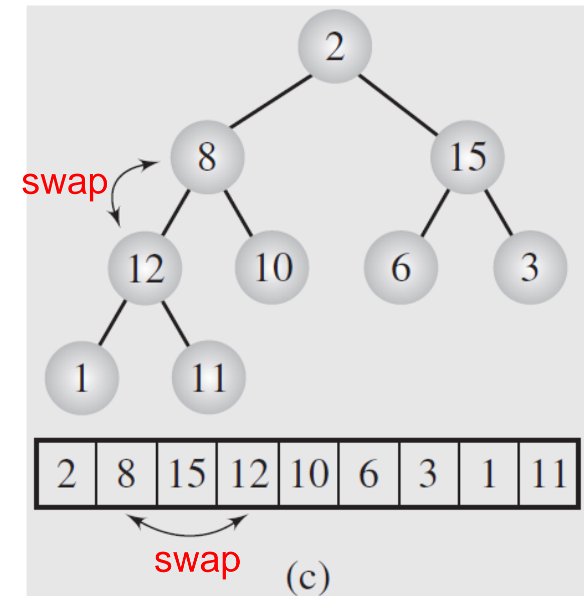
- ***Heap sort***
  - have the array sorted in *ascending order*
    - place the ***first largest element*** at the ***end*** of array;
    - then put the ***second largest element*** in ***front*** of the *first largest element*;
    - …
    - etc.
- Differences between ***heap sort*** and *selection sort*
  - heap sort: the largest element; the end of array
  - selection sort: the smallest element; the beginning of array
  - ***result is same***

Efficient Sorting Algorithms: Heap Sort

- Two phases of heap sort
  - 1st phase: build a heap out of the data set

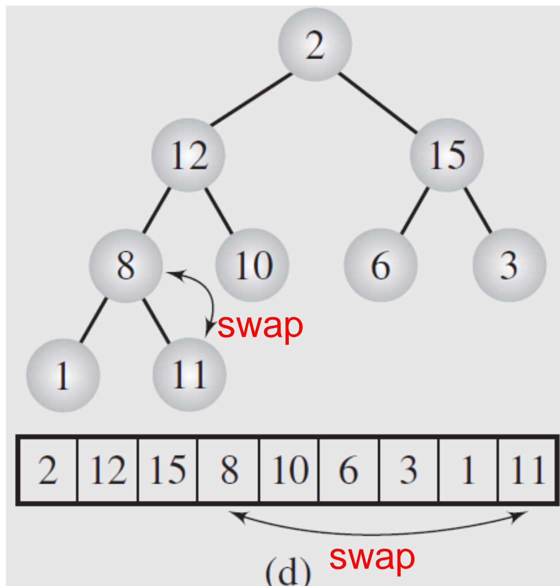


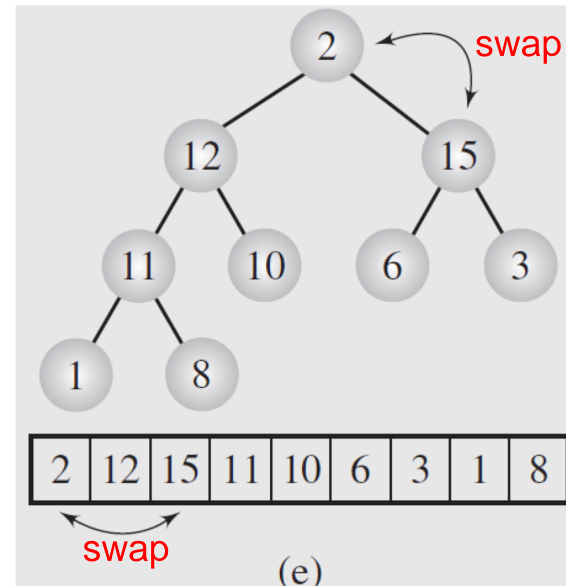




Efficient Sorting Algorithms: Heap Sort

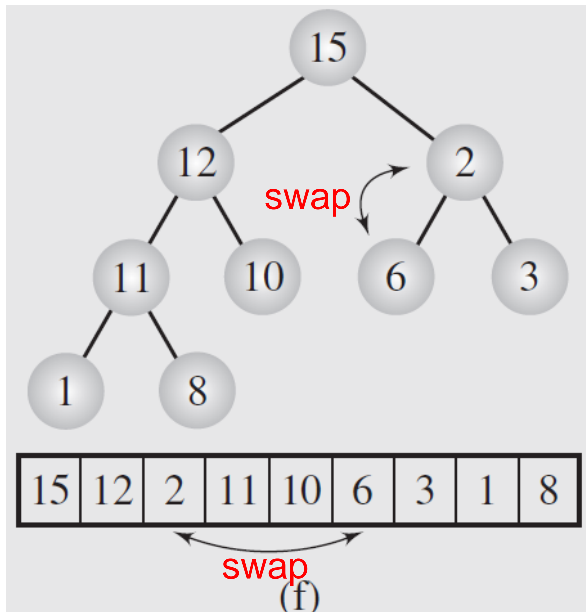
- Two phases of heap sort
  - 1st phase: build a heap out of the data set

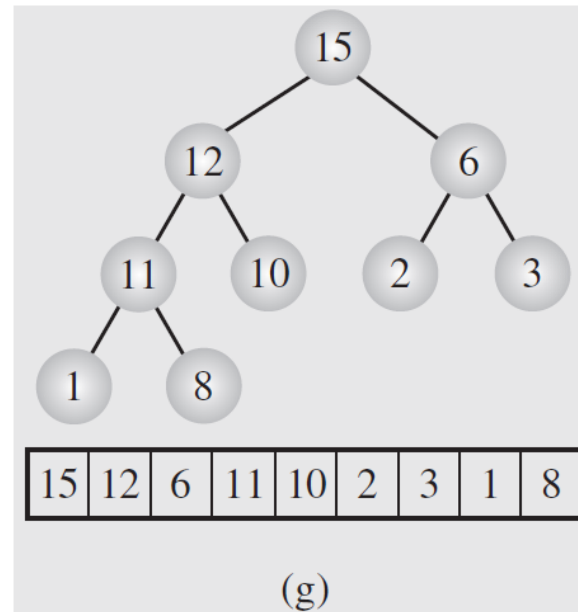




Efficient Sorting Algorithms: Heap Sort

- Two phases of heap sort
  - 1st phase: build a heap out of the data set



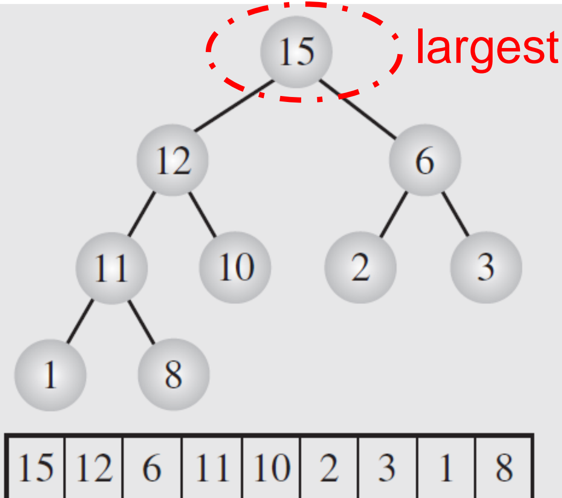


- the value of each node is greater than or equal to the values stored in its children
- the tree is perfectly balanced, and the leaves in the last level are leftmost in the tree

Efficient Sorting Algorithms: Heap Sort

- Two phases of heap sort
  - 1st phase: build a heap out of the data set
  - 2nd phase: find the largest item from the heap and move it to the end of array

violating heap property! ***restore***



(a)

largest

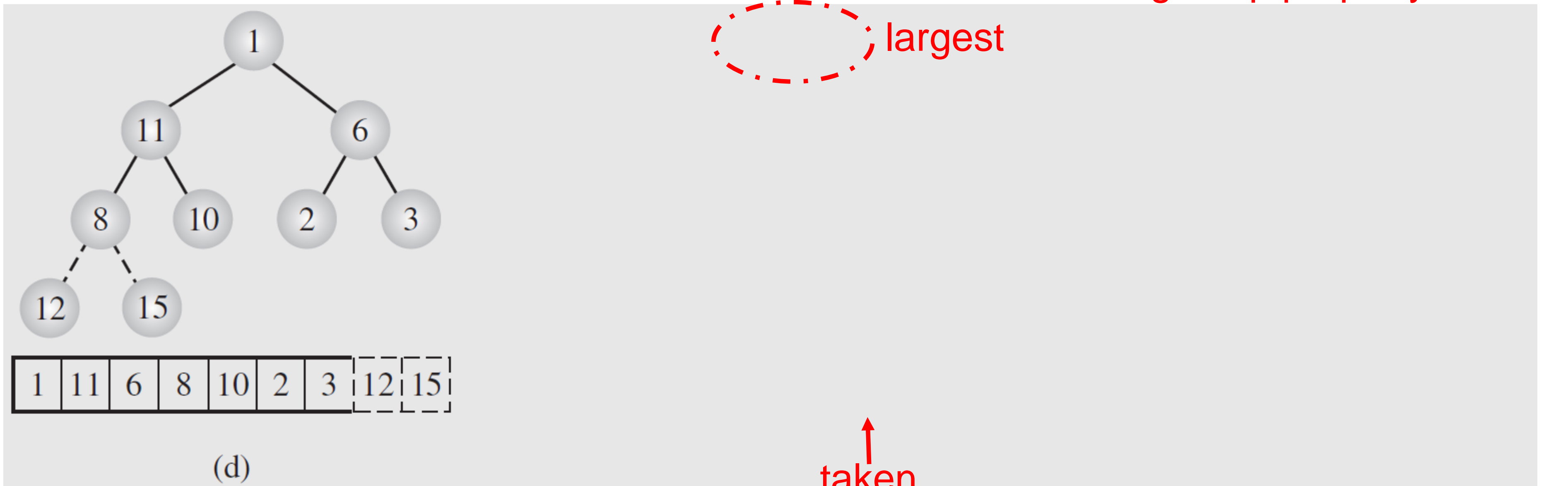
taken

largest

taken

Efficient Sorting Algorithms: Heap Sort

- Two phases of heap sort
  - 1st phase: build a heap out of the data set
  - 2nd phase: find the largest item from the heap and move it to the end of array

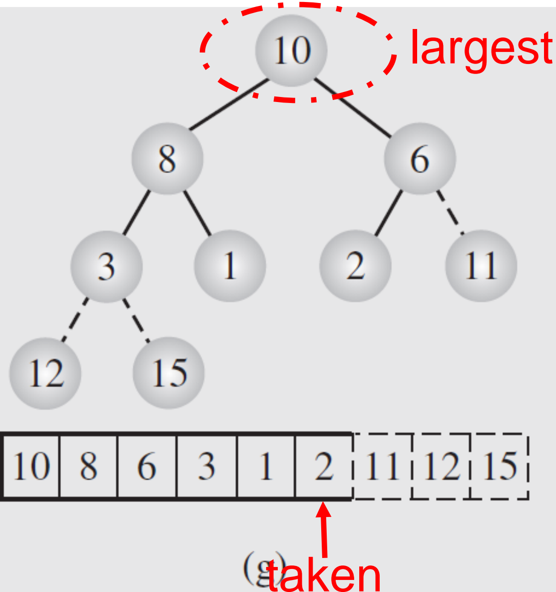


(d)

Efficient Sorting Algorithms: Heap Sort

- Two phases of heap sort
  - 1st phase: build a heap out of the data set
  - 2nd phase: find the largest item from the heap and move it to the end of array

violating heap property! ***restore***



Efficient Sorting Algorithms: Heap Sort

- Implementation of heapsort()

```
template<class T>
void heapsort(T data[], int n) {
    for (int i = n/2 - 1; i >= 0; --i)    // create a heap;
        moveDown (data,i,n-1);
    for (int i = n-1; i >= 1; --i) {
        swap(data[0],data[i]); // move the largest item to data[i];
        moveDown(data,0,i-1);  // restore the heap property;
    }
}
```

- code for swap () and moveDown () can be found on Canvas

Efficient Sorting Algorithms: Merge Sort

- ***Merge sort***
  - three major operations:
    - **divide** ⟵ this process stops when the subarray has one element
      - partition the n-element array to be sorted into two sub-array of n/2 elements
    - **conquer**
      - sort the two sub-arrays **recursively**
    - **combine**
      - merge the two ***sorted*** sub-arrays of size n/2 to produce the sorted array of n elements

Efficient Sorting Algorithms: Merge Sort (cont.)

- e.g., sort the array using merge sort

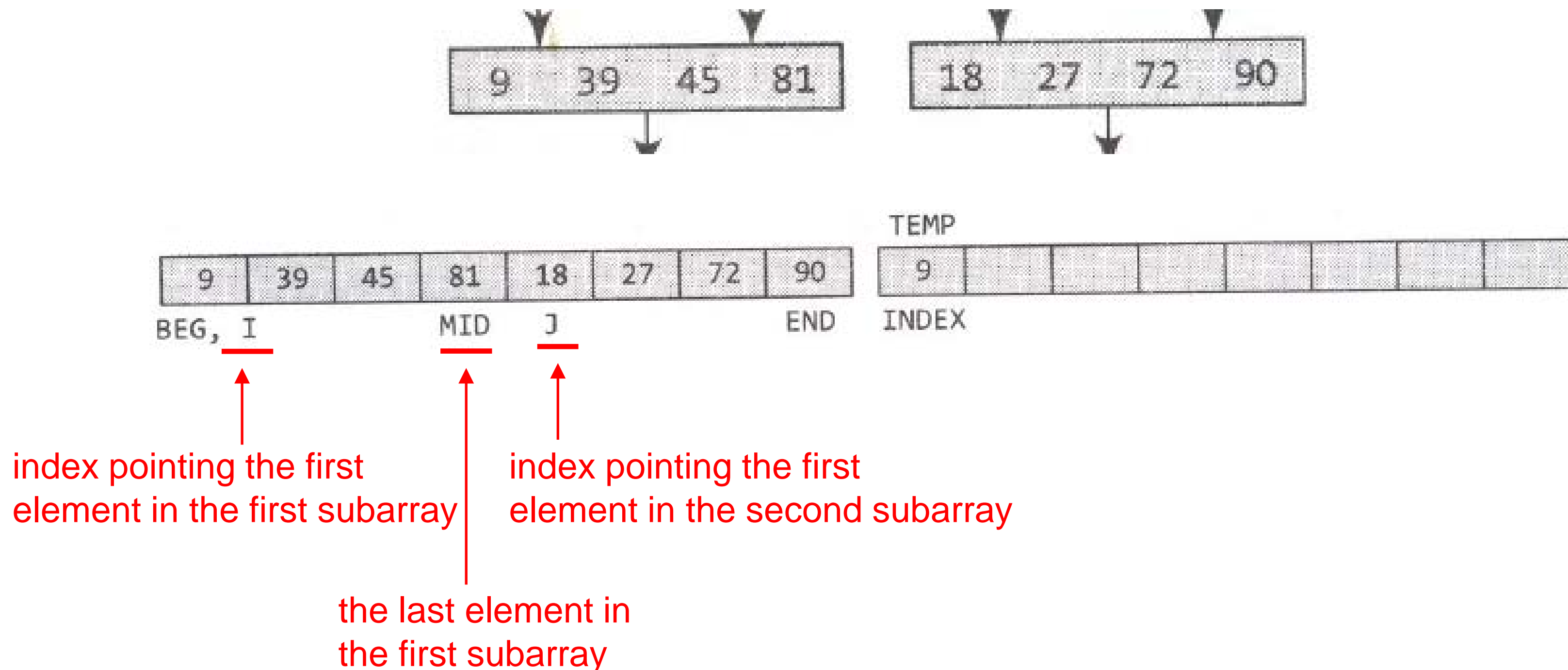
39	9	81	45	90	27	72	18

39	9	81	45	90	27	72	18

(Combine the elements to form a sorted array)

Efficient Sorting Algorithms: Merge Sort (cont.)

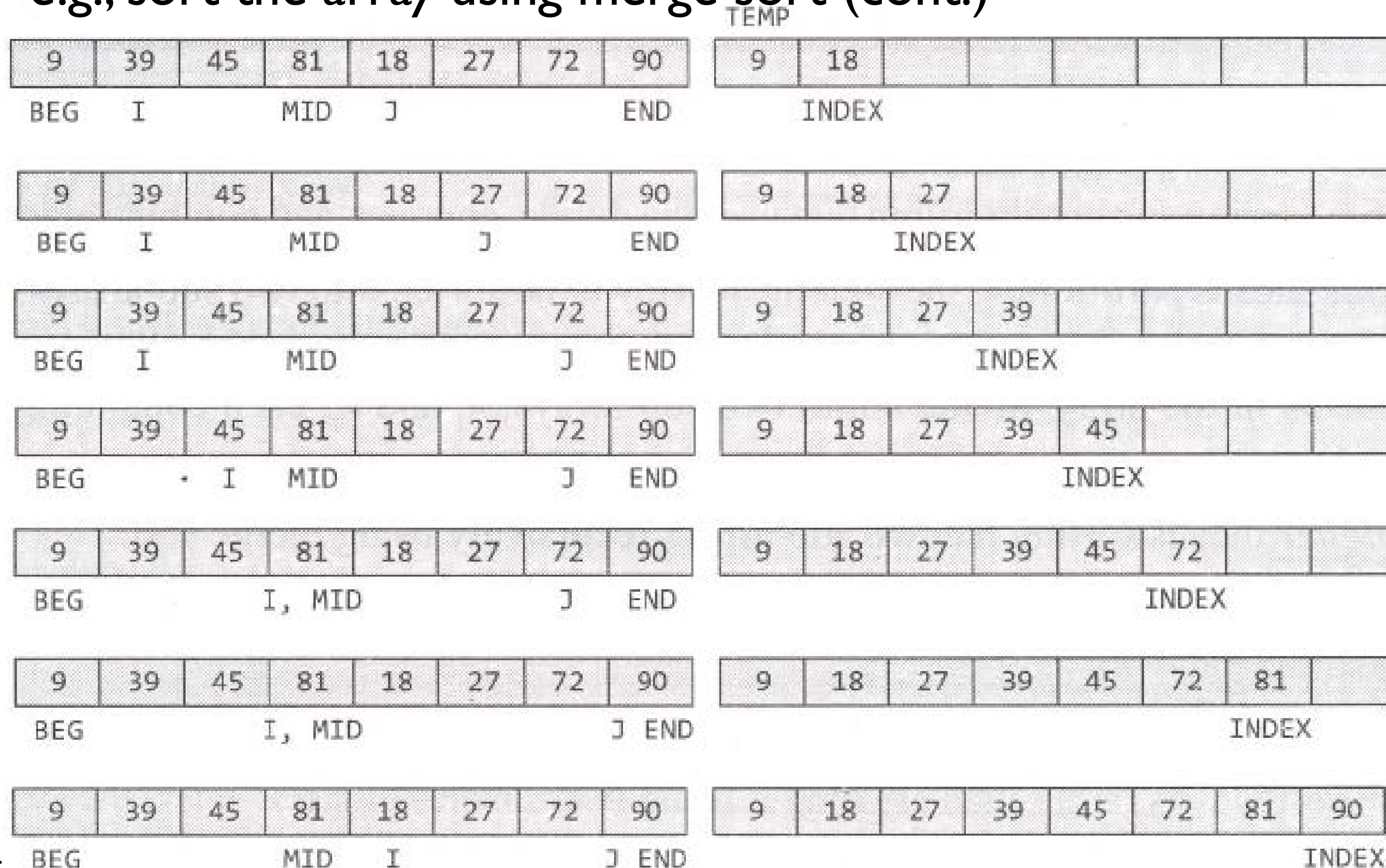
- e.g., sort the array using merge sort (cont.)



Efficient Sorting Algorithms: Merge Sort (cont.)

when I > MID than copy the remaining elements of the right sub-array in TEMP

- e.g., sort the array using merge sort (cont.)



Efficient Sorting Algorithms: Merge Sort (cont.)

```
mergesort(data[])
    if data have at least two elements
        mergesort(left half of data);
        mergesort(right half of data);
        merge(both halves into a sorted list);
```

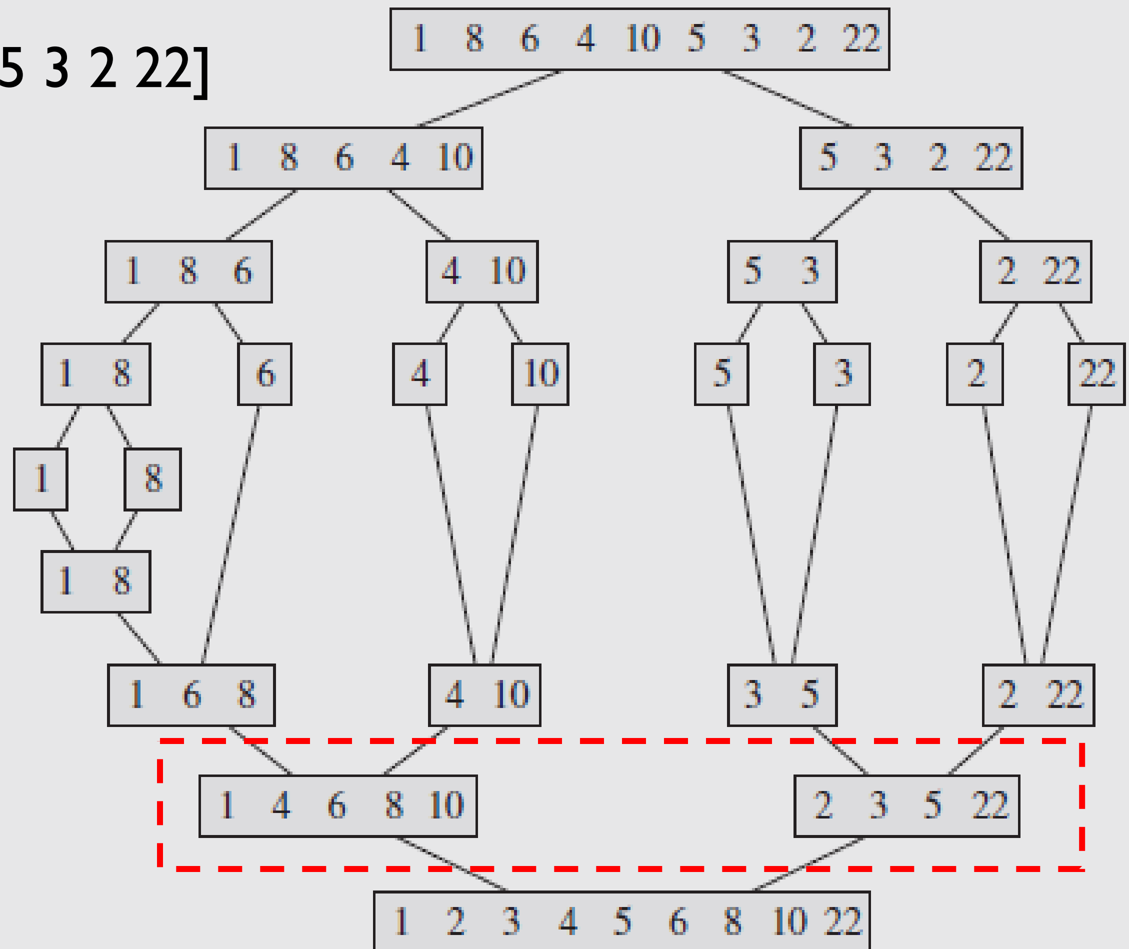
- stop when subarray has one item
- keep dividing left half
- keep dividing right half

```
merge(array1[], array2[], array3[])
    i1, i2, i3 are properly initialized;
    while both array2 and array3 contain elements
        if array2[i2] < array3[i3]
            array1[i1++] = array2[i2++];
        else array1[i1++] = array3[i3++];
    load into array1 the remaining elements of either array2 or array3;
```

- Key operations
  - merge the ***sorted*** halves of the array into a single array
  - these halves must be sorted, which occurs by merging the sorted halves of these halves

Efficient Sorting Algorithms: Merge Sort (cont.)

- e.g., the array [1 8 6 4 10 5 3 2 22] sorted by merge sort
- Drawback of merge sort?
  - ***additional storage*** for merging array



ITLE Course Observation

- Scan the following QR code

- Or visit https://forms.office.com/r/pd2dCBkJew