# Sorting

CSE 2020 Computer Science II

# Learning Objectives

- Define sorting algorithm
- Explain, implement and analyze insertion sort, selection sort, heap sort, and merge sort.

# Sorting

- Given an array of elements, sorting is to arrange these elements from the smallest to the largest.

  $v_0, v_1, v_2, \ldots\ldots, v_{n-1}$ so that $v_i \leq v_{i+1}$ for all i

- Duplicate elements are allowed in the array.
- A sorting algorithm is stable if it maintains the initial ordering among duplicates.
- Sorting algorithm analysis measures the number of comparisons made between elements.
- We assume the array contains only integers, although these algorithms allow more general objects (as long as objects are comparable).

# Insertion Sort

- Consists of N-1 passes.
- For pass i = 1 through N-1,
  - before the pass i, v[0 … i - 1] are sorted
  - in the pass i, v[i] is moved left to the correct position in v[0 … i]
  - after the pass i, v[0 … i] are sorted

# Code of Insertion Sort

```
template <typename C>
void insertionSort(vector<C>& v) {
   for (int i = 1; i < v.size(); i++)  {
        C temp = v[i];
        int j = i;
        while (j > 0 && v[j - 1] > temp)  {
              v[j] = v[j - 1];
              --j;
        }
        v[j] = temp;
    }
}
```

# Example of Insertion sort

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| elements | 42 | 20 | 17 | 13 | 28 | 14 | 23 | 15 |
| i = 1 | 20 | 42 | | | | | | |
| i = 2 | 17 | 20 | 42 | | | | | |
| i = 3 | 13 | 17 | 20 | 42 | | | | |
| i = 4 | 13 | 17 | 20 | 28 | 42 | | | |
| i = 5 | 13 | 14 | 17 | 20 | 28 | 42 | | |
| i = 6 | 13 | 14 | 17 | 20 | 23 | 28 | 42 | |
| i = 7 | 13 | 14 | 15 | 17 | 20 | 23 | 28 | 42 |

# insertion sort analysis

- Outer for loop N-1 times
- Inner while loop depends on input data
  - best case, elements are sorted from smallest to largest, so v[j-1] > temp is false, 1 comparison

    N-1 comparisons ➔ O(N)
  - worst case, elements are sorted from largest to smallest, so v[j-1] > temp is true and is executed i times, i comparisons

$$\sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} \Rightarrow O(N^2)$$

- Stable

# Selection Sort

- Consists of N-1 passes.
- For pass i = 0 through N-2,
  - before the pass i, v[0 … i-1] are in their right places
  - in the pass i, find the smallest from v[i … N-1], swap it to v[i]
  - after the pass i, v[0 … i] are in their right places

# Code of Selection Sort

```
template <typename C>
void selectionSort(vector<C>& v) {
   for (int i = 0; i < v.size() - 1; i++)  {
        int min = i;
        for (int j = i + 1; j < v.size(); j++)
             if (v[j] < v[min])
                  min = j;
        if (min != i)
             swap v[i] and v[min];
    }
}
```

# Example of selection sort

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| elements | 42 | 20 | 17 | 13 | 28 | 14 | 23 | 15 |
| i = 0 | 13 | | | 42 | | | | |
| i = 1 | 13 | 14 | | | | 20 | | |
| i = 2 | 13 | 14 | 15 | | | | | 17 |
| i = 3 | 13 | 14 | 15 | 17 | | | | 42 |
| i = 4 | 13 | 14 | 15 | 17 | 20 | 28 | | |
| i = 5 | 13 | 14 | 15 | 17 | 23 | 28 | 23 | |
| i = 6 | 13 | 14 | 15 | 17 | 20 | 23 | 28 | 42 |

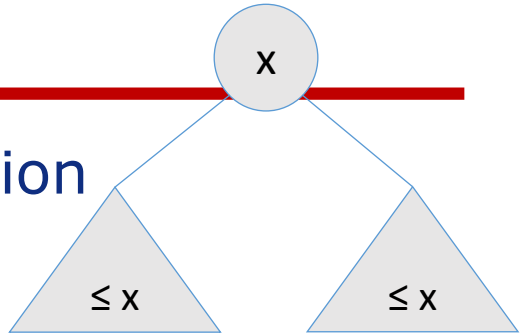
# Selection Sort Analysis

- Outer for loop N-1 times
- Inner for loop N-i-1 times comparisons, no best or worst cases, the input order doesn't affect the number of comparisons

$$\sum_{i=0}^{N-2} N - i \ - 1 = \frac{N(N-1)}{2} \Rightarrow O(N^2)$$

- Stable

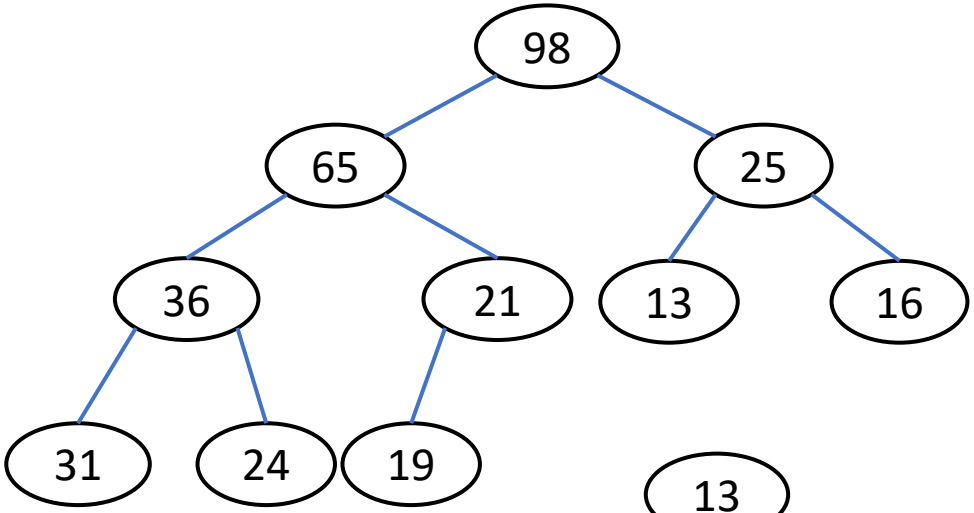
# Heap Sort



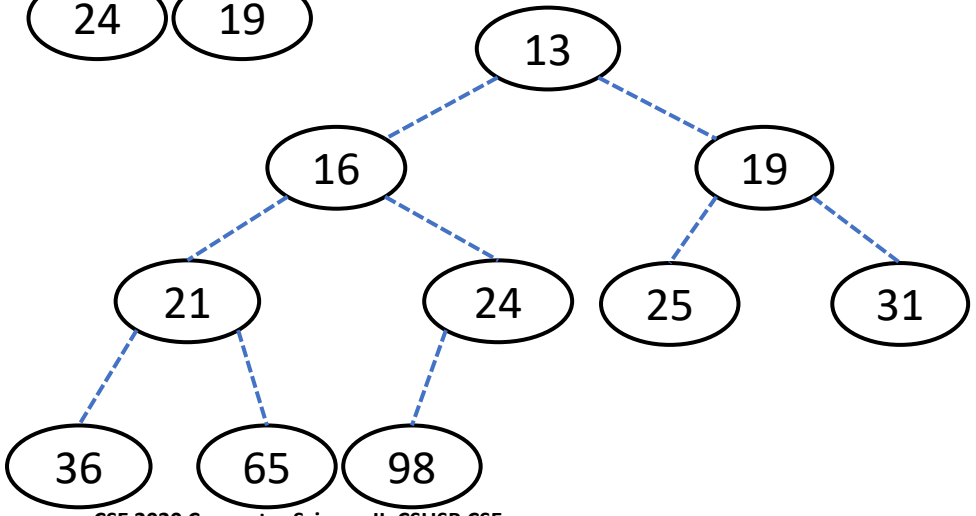

- Binary max heap with *deleteMax* operation
- Steps:
  1. build a max heap from a vector of elements
  2. swapping the last element in the heap with the first
  3. decrement the max heap size
  4. percolate down the first element to the right position until heap order is satisfied
  5. repeat steps 2-4 N-1 times
  6. the elements in the vector are sorted from the smallest to largest
- Time complexity – $O(NlogN)$

# Heap Sort Example

- 24, 21, 25, 36, 19, 13, 16, 31, 65, 98
- Max-heap




- Heap sort



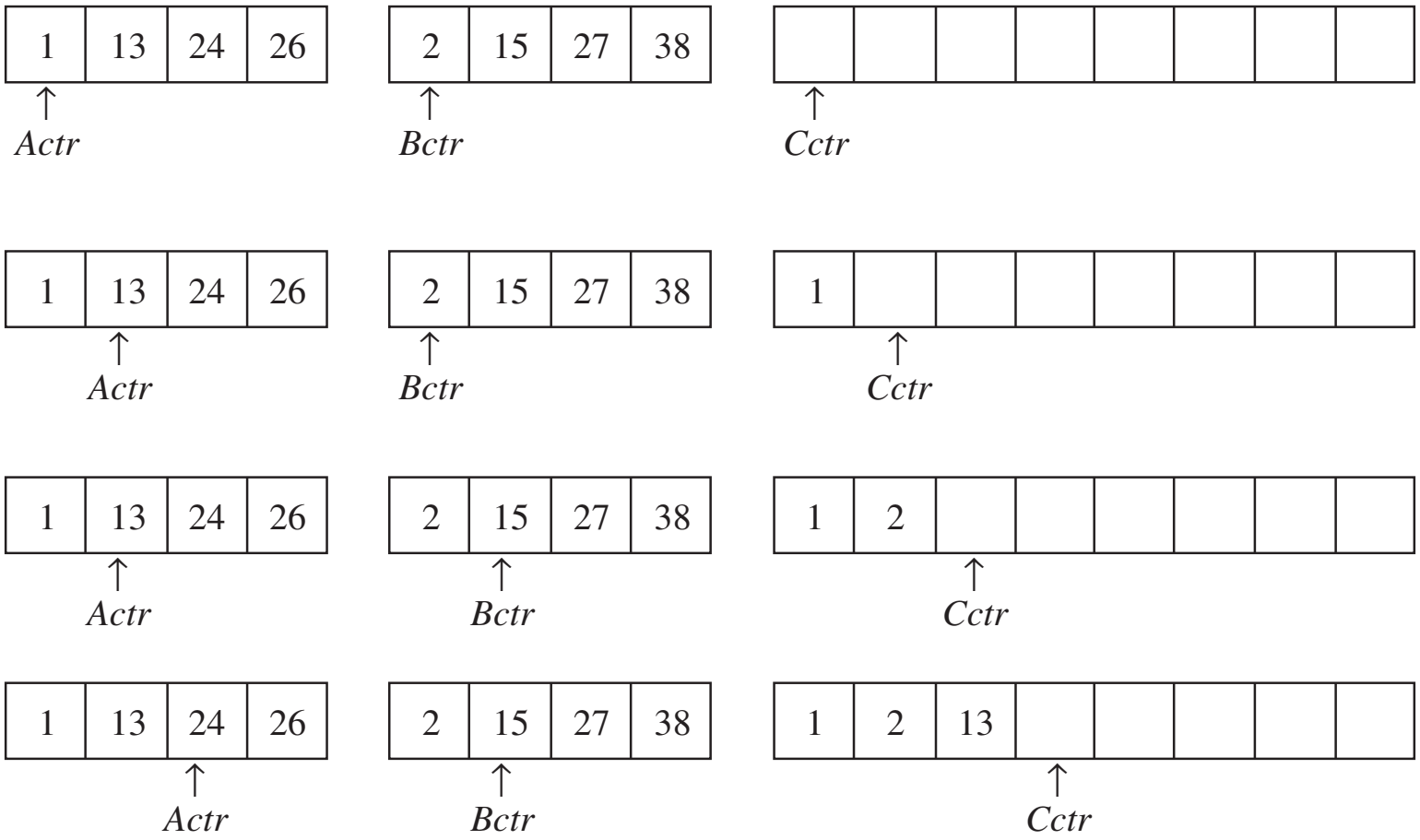
# Merge Sort

- The fundamental operation in merge sort is merging two sorted arrays. Because the arrays are sorted, merging can be done in one pass through the input arrays.

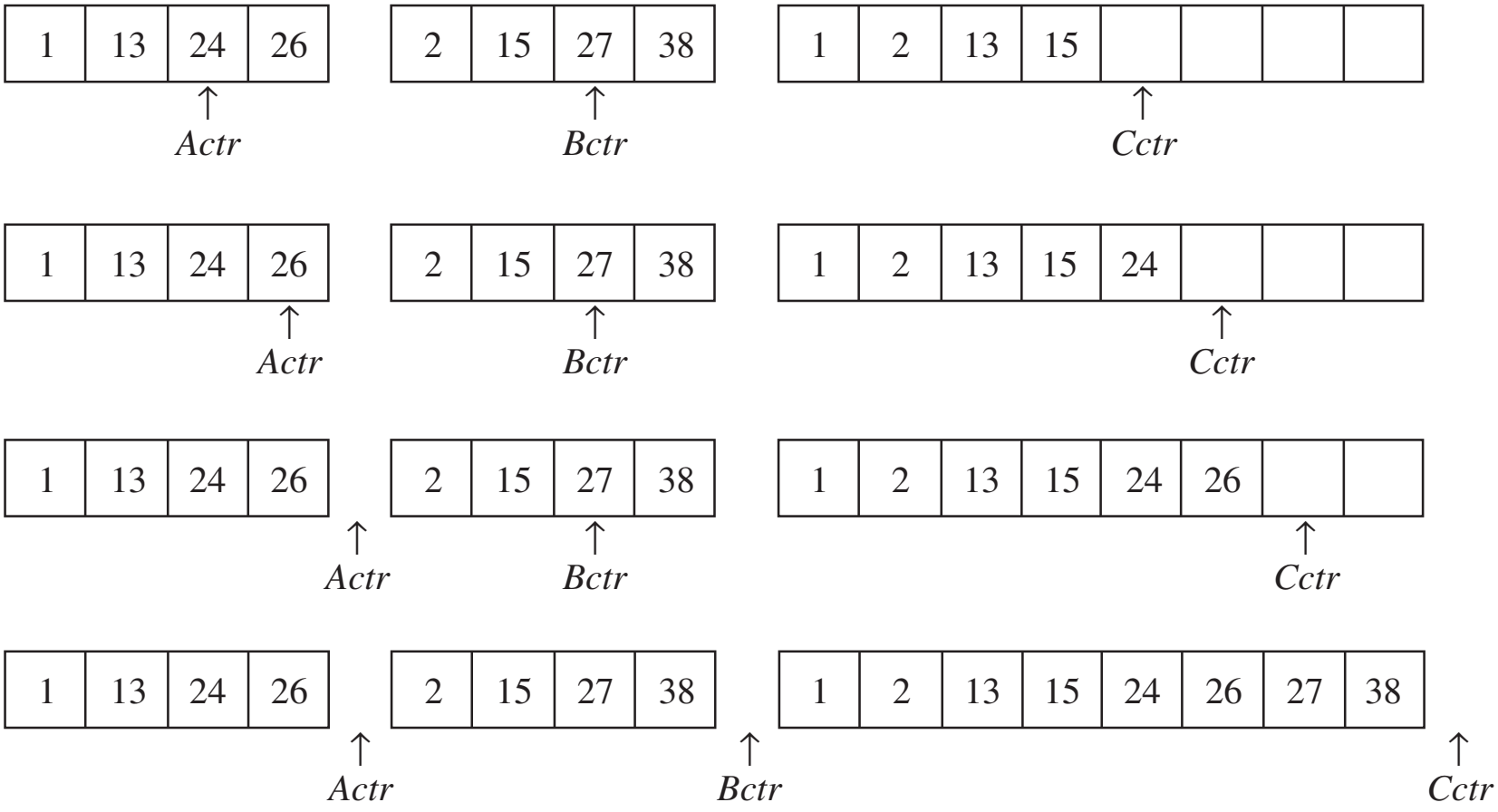
# Merge Two Sorted Arrays

- Merge sorted arrays A and B into C
  1. 3 counters, Actr, Bctr, Cctr are initially set to the beginning of A, B, C
  2. the smaller of A[Actr] and B[Bctr] is copied to C[Cctr]
  3. increment corresponding counters
  4. repeat 2 and 3
  5. when either A or B is exhausted, the remaining of the other array is copied to C
- Comparison times $O(N)$

# Example of Merging



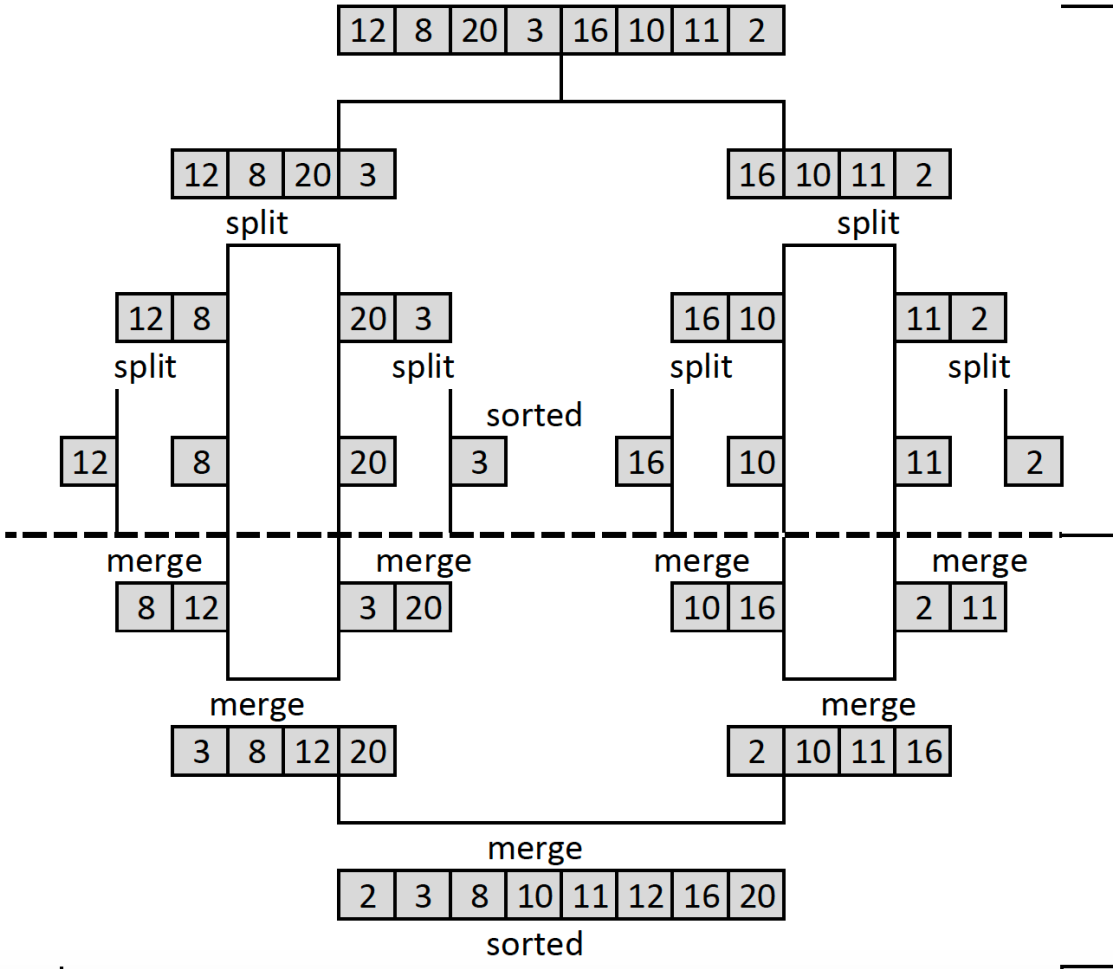
# Example of Merging (cont.)



# Merge Sort – Recursive function

- Merge sort uses divide and conquer strategy, in which the problem is divided into smaller problems and solved recursively.
  - Base case: N = 1, there is only one element to sort, in fact, it is sorted.
  - Recursive function call: recursively merge sort the left half and the right half, which give two sorted halves, then use merging algorithm to merge two sorted halves.

# Merge Sort Example



# Implementation

- template <typename C>
- void mergeSort( vector<C>& v );

- template <typename C>
- void mergeSort( vector<C>& v, vector<C>& tmp,
  int left, int right);

- template <typename C>
- void merge(vector<C>& v, vector<C>& tmp,
  int leftPos, int rightPos, int righEnd);

# Time complexity

- $T(N) = 2T\left(\frac{N}{2}\right) + N$

$$= 2\left[2\,T\left(\frac{N}{2\times 2}\right) + \frac{N}{2}\right] + N$$

$$= 2\times 2\,T\left(\frac{N}{2\times 2}\right) + N + N$$

$$= 2\times 2\left[2T\left(\frac{N}{2\times 2\times 2}\right) + \frac{N}{2\times 2}\right] + N + N$$

$$= 2\times 2\times 2T\left(\frac{N}{2\times 2\times 2}\right) + N + N + N$$

$$= \ldots\ldots\ldots$$

$$= 2^k T\left(\frac{N}{2^k}\right) + kN$$

# Time complexity (cont.)

- Assuming $\mathrm{N} = 2^k$, so $k = log_2 N$

$$T(N) = 2^k T\left(\frac{N}{2^k}\right) + kN$$

$$= NT(1) + Nlog_2 N$$

$$= N + Nlog_2 N$$

$$\Rightarrow O(NlogN)$$