



# Computer Programming

Dr. Deepak B Phatak

Dr. Supratik Chakraborty

Department of Computer Science and Engineering  
IIT Bombay

Session: Merge Sort in C++ and Its Analysis

# Quick Recap of Relevant Topics

---



- The sorting problem
- Selection sort
- Merge sort
  - Intuition
  - Divide-and-conquer strategy
  - Key role of merging sorted sub-arrays

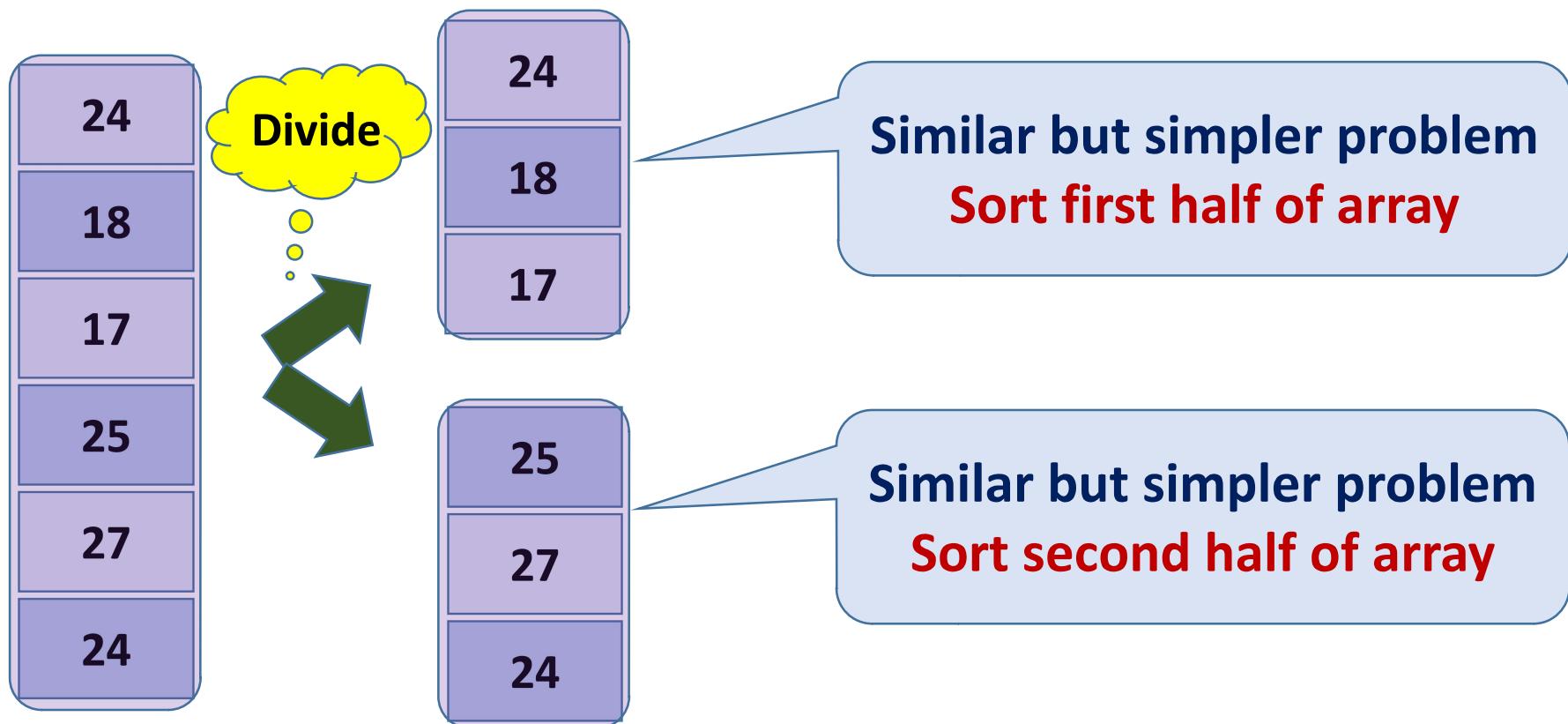
# Overview of This Lecture

---

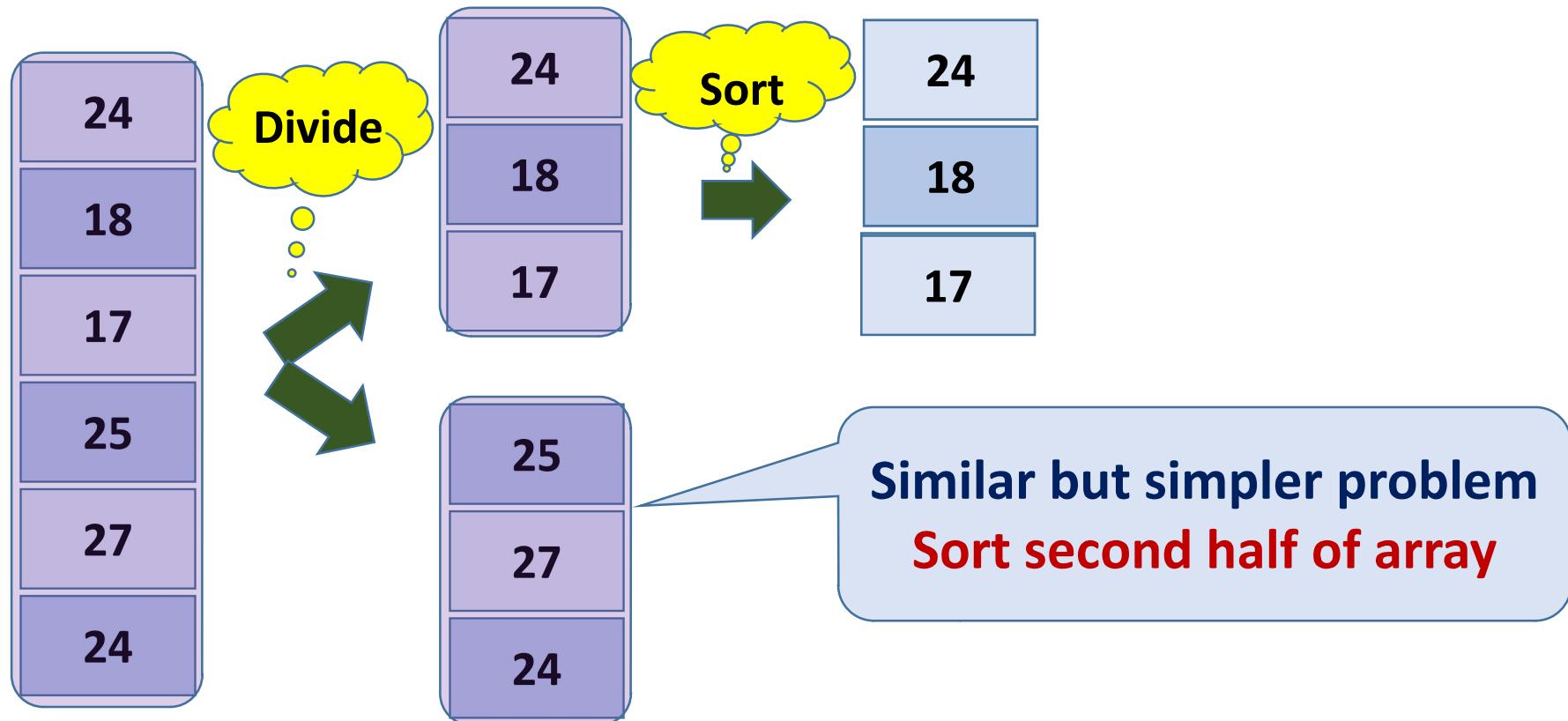


- Merge sort
  - C++ implementation
  - Analysis of performance
    - Counting “basic” steps to sort an array of size  $n$

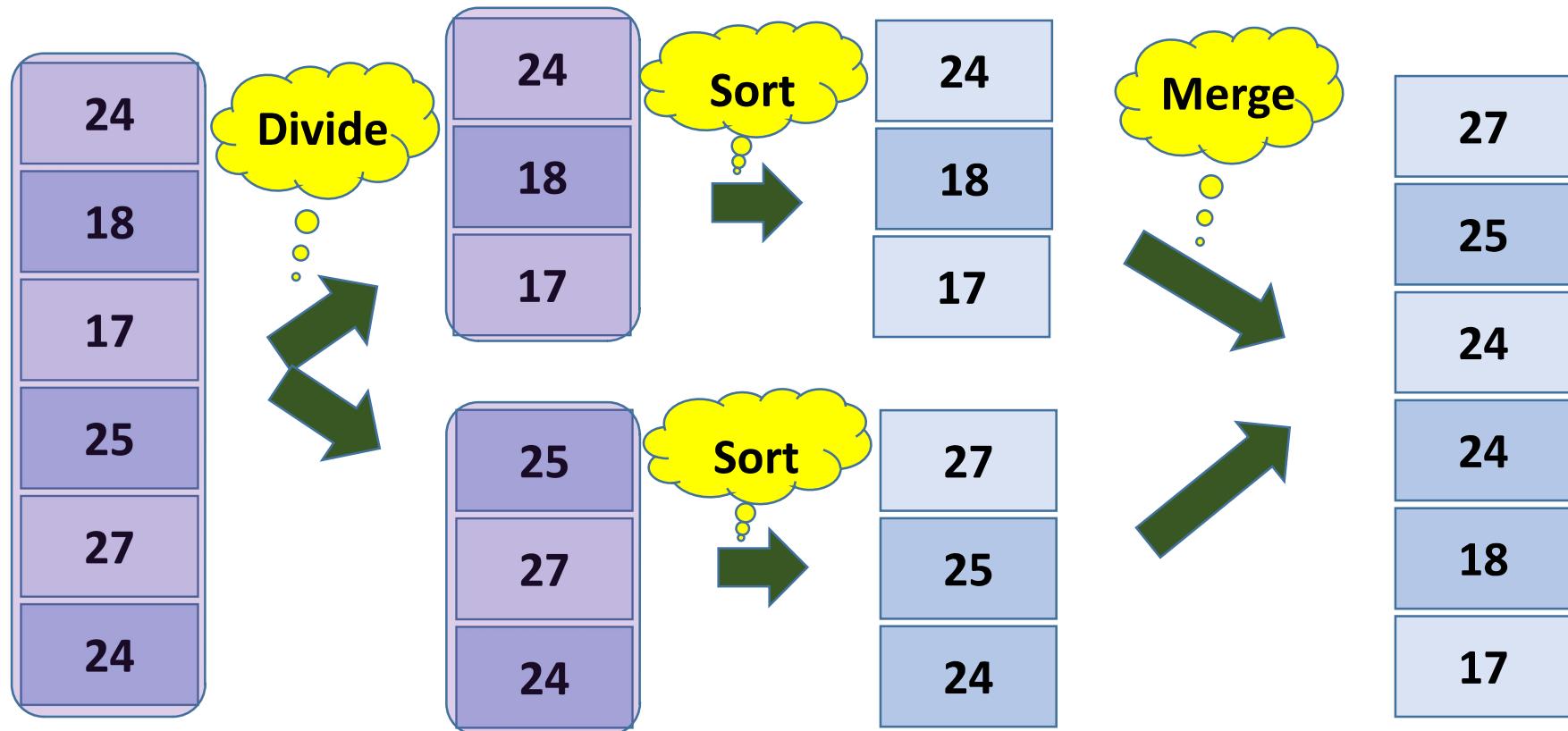
# Merge Sort: Basic Idea



# Sorting by Divide-and-Conquer



# Sorting by Divide-and-Conquer



# What Were The Steps?

---



- Divide an array of size  $n$  into two sub-arrays of size  $\approx n/2$ 
  - Sub-array sizes may differ by 1 if  $n$  is odd
  - Easy!
- Sort each sub-array of size  $n/2$ 
  - Use same technique as for sorting original array (recurse !!!)
  - Termination case of recursion: arrays of size 1
- Merge sorted sub-arrays, each of size  $n/2$ 
  - One pass over each sorted sub-array

# Merge Sort In C++



```
int main() {  
    int i, n, A[100]; // Variable declarations  
    cout << "Give count of integers: "; cin >> n; // Read and validate inputs  
    if ((n <= 0) || (n > 100)) { cout << "Invalid input!" << endl; return -1; }  
    cout << "Give " << n << " integers to sort." << endl;  
    for (i = 0; i < n; i++) { cin >> A[i]; } // Read elements of array A  
    mergeSort(A, 0, n); // Sort elements A[0] ... A[n-1]  
    ... Rest of code ...  
    return 0;  
}
```

# Merge Sort In C++



```
int main() {  
    int i, n, A[100]; // Variable declarations  
    cout << "Give count of integers: "; cin >> n;// Read and validate inputs  
    if ((n <= 0) || (n > 100)) { cout << "Invalid input!" << endl; return -1; }  
    cout << "Give " << n << " integers to sort." << endl;  
    for (i = 0; i < n; i++) { cin >> A[i]; } // Read elements of array A  
    mergeSort(A, 0, n); // Sort elements A[0] ... A[n-1]  
    ... Rest of code ...  
    return 0;  
}
```

# Merge Sort In C++



```
int main() {  
    int i, n, A[100]; // Variable declarations  
    cout << "Give count of integers: "; cin >> n;// Read and validate inputs  
    if ((n <= 0) || (n > 100)) { cout << "Invalid input!" << endl; return -1; }  
    cout << "Give " << n << " integers to sort." << endl;  
    for (i = 0; i < n; i++) { cin >> A[i]; } // Read elements of array A  
mergeSort(A, 0, n); // Sort elements A[0] ... A[n-1]  
    ... Rest of code ...  
    return 0;  
}
```

# Merge Sort In C++

---



```
// PRECONDITION: start < end, both within array bounds of A
void mergeSort(int A[], int start, int end) {
    if (end == start + 1) { return; } //Subarray of interest has 1 element
    int mid = (start + end)/2; //Get mid-index of subarray of interest
    mergeSort(A, start, mid); // Sort subarray A[start] ... A[mid-1]
    mergeSort(A, mid, end); // Sort subarray A[mid] ... A[end-1]
    // Merge sorted subarrays A[start] ... A[mid-1] and A[mid] ... A[end-1]
    mergeSortedSubarrays(A, start, mid, end);
    return;
}
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

# Merge Sort In C++



```
// PRECONDITION: start < end, both within array bounds of A
void mergeSort(int A[], int start, int end) {
    if (end == start + 1) { return; } //Subarray of interest has 1 element
    int mid = (start + end)/2; //Get mid-index of subarray of interest
    mergeSort(A, start, mid); // Sort subarray A[start] ... A[mid-1]
    mergeSort(A, mid, end); // Sort subarray A[mid] ... A[end-1]
    // Merge sorted subarrays
    mergeSortedSubarrays(A, start, mid, end);
    return;
}
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

**Termination case of recursion**

# Merge Sort In C++



```
// PRECONDITION: start < end, both within array bounds of A
void mergeSort(int A[], int start, int end) {
    if (end == start +1) { return; } //Subarray of interest has 1 element
    int mid = (start + end)/2; //Get mid-index of subarray of interest
    mergeSort(A, start, mid); // Sort subarray A[start] ... A[mid-1]
    mergeSort(A, mid, end); // Sort subarray A[mid] ... A[end-1]
    // Merge sorted subarrays A[start] ... A[mid-1] and A[mid] ... A[end-1]
    mergeSortedSubarrays(A, start, mid, end);
    return;
}
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

# Merge Sort In C++



```
// PRECONDITION: start < end, both within array bounds of A
void mergeSort(int A[], int start, int end) {
    if (end == start + 1) { return; } //Subarray of interest has 1 element
    int mid = (start + end)/2; //Get mid-index of subarray of interest
    mergeSort(A, start, mid); // Sort subarray A[start] ... A[mid-1]
    mergeSort(A, mid, end); // Sort subarray A[mid] ... A[end-1]
    // Merge sorted subarrays A[start] ... A[mid-1] and A[mid] ... A[end-1]
    mergeSortedSubarrays(A, start, mid, end);
    return;
}
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

# Merge Sort In C++



```
// PRECONDITION: A[start] ... A[mid-1] and A[mid] ... A[end-1] sorted in
//                  decreasing order
void mergeSortedSubarrays(int A[], int start, int mid, int end) {
    int i, j; int tempA[100], index = start;
    for (i = start, j = mid; ((i < mid) || (j < end)); ) { // Merging loop
        // Determine whether A[i] or A[j] should appear next in sorted order
        // Update tempA[index] accordingly
        index++;
    } // end of merging loop
    // Copy tempA[start] ... tempA[end-1] to A[start] ... A[end-1]
    return;
}
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

# Merge Sort In C++



```
// PRECONDITION: A[start] ... A[mid-1] and A[mid] ... A[end-1] sorted in  
// decreasing order
```

```
void mergeSortedSubarrays(int A[], int start, int mid, int end) {
```

```
    int i, j; int tempA[100], index = start;
```

```
    for (i = start, j = mid; i < end; i++) {
```

```
        // Determine whether to take from first or second subarray
```

```
        // Update temporary array
```

```
        index ++;
```

```
} // end of merge
```

```
// O(n)
```

```
return index;
```

```
}
```

Running index for second subarray  
 $A[mid] \dots A[end-1]$

Running index for first subarray  $A[start] \dots A[mid-1]$

```
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

# Merge Sort In C++

```
// PRECONDITION: A[start] ... A[mid-1] and A[mid] ... A[end-1] sorted in
// decreasing order
void mergeSortedSubarrays(int A[], int start, int mid, int end) {
    int i, j; int tempA[100], index = start;
    for (i = start, j = mid; ((i < mid) && (j < end)); index++) { // Merging loop
        // Determining the smaller element
        // Update the temporary array
        index++;
    } // End of merging loop
}

// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

**Index in tempA where next element of merged-and-sorted subarray is stored**

**Merged-and-sorted subarray temporarily stored in tempA[start] ... tempA[end-1]**

# Merge Sort In C++



```
// PRECONDITION: A[start] ... A[mid-1] and A[mid] ... A[end-1] sorted in
//                  decreasing order
void mergeSortedSubarrays(int A[], int start, int mid, int end) {
    int i, j; int tempA[100], index = start;
    for (i = start, j = mid; ((i < mid) || (j < end)); ) { // Merging loop
        // Determine whether A[i] or A[j] should appear next in sorted order
        // Update tempA[index] accordingly
        index++;
    } // end of merging loop
    // Copy tempA[start] ... tempA[end-1] to A[start] ... A[end-1]
    return;
}
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

# Merging Loop In C++



```
for (i = start, j = mid; ((i < mid) || (j < end)); ) { // Merging loop
    if ((i < mid) && (j < end)) { // None of the two subarrays fully seen yet
        }
    else { // One of the two subarrays fully seen.
        }
    index++;
} // end of merging loop
```

# Merging Loop In C++



```
for (i = start, j = mid; ((i < mid) || (j < end)); ) { // Merging loop
    if ((i < mid) && (j < end)) { // None of the two subarrays fully seen yet
        }
    else {
        // One of the two subarrays fully seen.
        // Copy elements from the other subarray to tempA.
    }
    index++;
} // end of merging loop
```

# Merging Loop In C++



```
for (i = start, j = mid; ((i < mid) || (j < end)); ) { // Merging loop
    if ((i < mid) && (j < end)) { // None of the two subarrays fully seen yet
        }
    else { if (i < mid) {tempA[index] = A[i]; i++;} // A[mid] ... A[end-1] seen
          else           {tempA[index] = A[j]; j++;} // A[start] ... A[mid-1] seen
        }
    index++;
} // end of merging loop
```

# Merging Loop In C++



```
for (i = start, j = mid; ((i < mid) || (j < end)); ) { // Merging loop
    if ((i < mid) && (j < end)) { // None of the two subarrays fully seen yet
        // Find which of A[i] (from subarray A[start] ... A[mid-1]) or
        // A[j] (from subarray A[mid] ... A[end-1]) is the next sorted element,
        } // copy that element to tempA[index]
    else { if (i < mid) {tempA[index] = A[i]; i++;} // A[mid] ... A[end-1] seen
          else           {tempA[index] = A[j]; j++;} // A[start] ... A[mid-1] seen
    }
    index++;
} // end of merging loop
```

# Merging Loop In C++



```
for (i = start, j = mid; ((i < mid) || (j < end)); ) { // Merging loop
    if ((i < mid) && (j < end)) { // None of the two subarrays fully seen yet
        if (A[j] > A[i]) {tempA[index] = A[j]; j++;}
        else {tempA[index] = A[i]; i++;}
    }
    else { if (i < mid) {tempA[index] = A[i]; i++;} // A[mid] ... A[end-1] seen
           else {tempA[index] = A[j]; j++;} // A[start] ... A[mid-1] seen
    }
    index++;
} // end of merging loop
```

# Merging Loop In C++: The Whole Story



```
for (i = start, j = mid; ((i < mid) || (j < end)); ) { // Merging loop
    if ((i < mid) && (j < end)) { // None of the two subarrays fully seen yet
        if (A[j] > A[i]) {tempA[index] = A[j]; j++;}
        else {tempA[index] = A[i]; i++;}
    }
    else { if (i < mid) {tempA[index] = A[i]; i++;} // A[mid] ... A[end-1] seen
           else {tempA[index] = A[j]; j++;} // A[start] ... A[mid-1] seen
    }
    index++;
} // end of merging loop
```

# Merge Sort In C++



```
// PRECONDITION: A[start] ... A[mid-1] and A[mid] ... A[end-1] sorted in  
// decreasing order
```

```
void mergeSortedSubarrays(int A[], int start, int mid, int end) {
```

```
    int i, j; int tempA[100], index = start;
```

```
    for (i = s
```

```
        // Det
```

```
for (i = start; i < end; i++) {A[i] = tempA[i];}
```

```
        // Update temp
```

```
        index ++;
```

```
} // end of merging loop
```

```
    // Copy tempA[start] ... tempA[end-1] to A[start] ... A[end-1]
```

```
    return;
```

```
}
```

```
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

# “Basic” Steps in mergeSortedSubarrays

---



- Reading two array elements, comparing them, writing an element in tempA, incrementing indices
- Copying an element from tempA to A

# Counting “Basic” Steps In mergeSortedSubarrays



- Suppose mergeSortedSubarrays called with sub-arrays  $A[\text{start}] \dots A[\text{mid}-1]$  and  $A[\text{mid}] \dots A[\text{end}-1]$ , each of size  $n/2$ 
  - At most  $n/2$  “basic” steps to iterate over  $A[\text{start}] \dots A[\text{mid}-1]$
  - At most  $n/2$  “basic” steps to iterate over  $A[\text{mid}] \dots A[\text{end}-1]$
  - In all, at most  $(n/2 + n/2)$  or  $n$  “basic” steps to get merged sorted subarray in  $\text{tempA}$
  - $n$  “basic” steps to copy  $\text{tempA}[\text{start}] \dots \text{tempA}[\text{end}-1]$  to  $A[\text{start}] \dots A[\text{end}-1]$

**Overall, at most  $2n$  “basic” steps**

# Counting “Basic” Steps in Merge Sort

---



- Let  $T_n$  denote maximum count of “basic” steps to merge sort an array of size  $n$

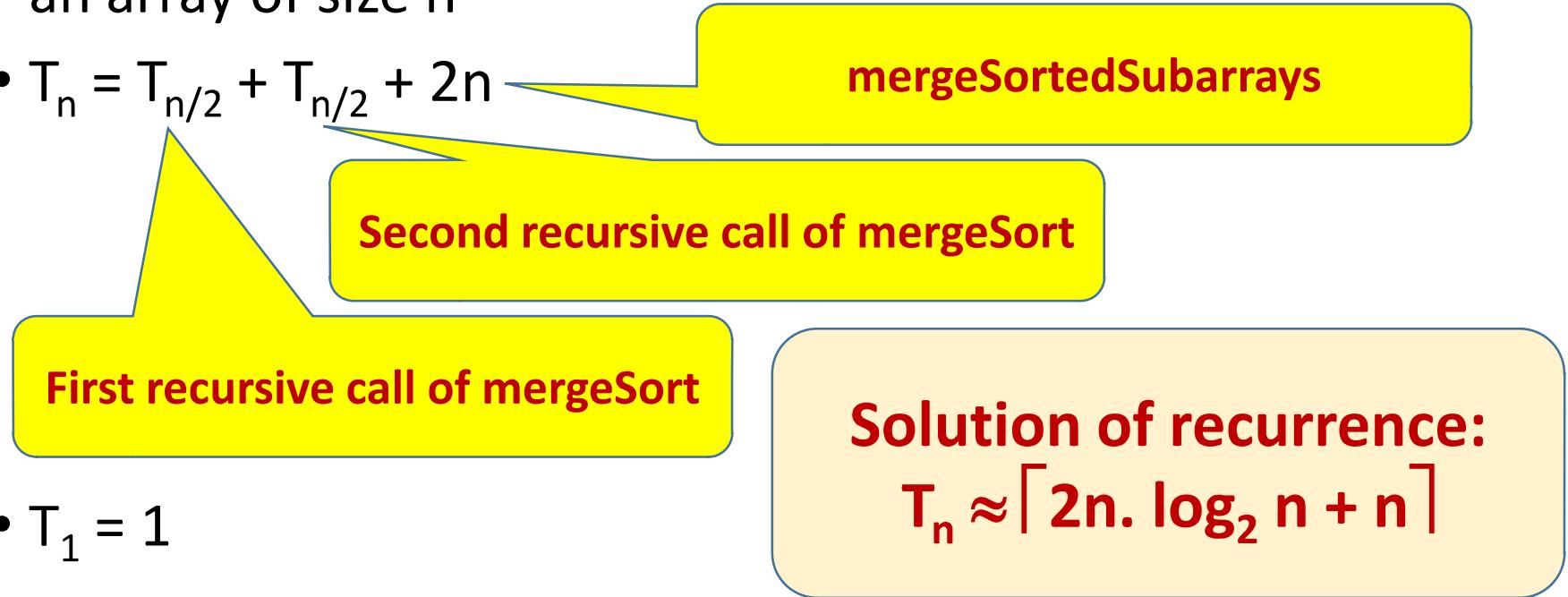
# Merge Sort: Looking Back



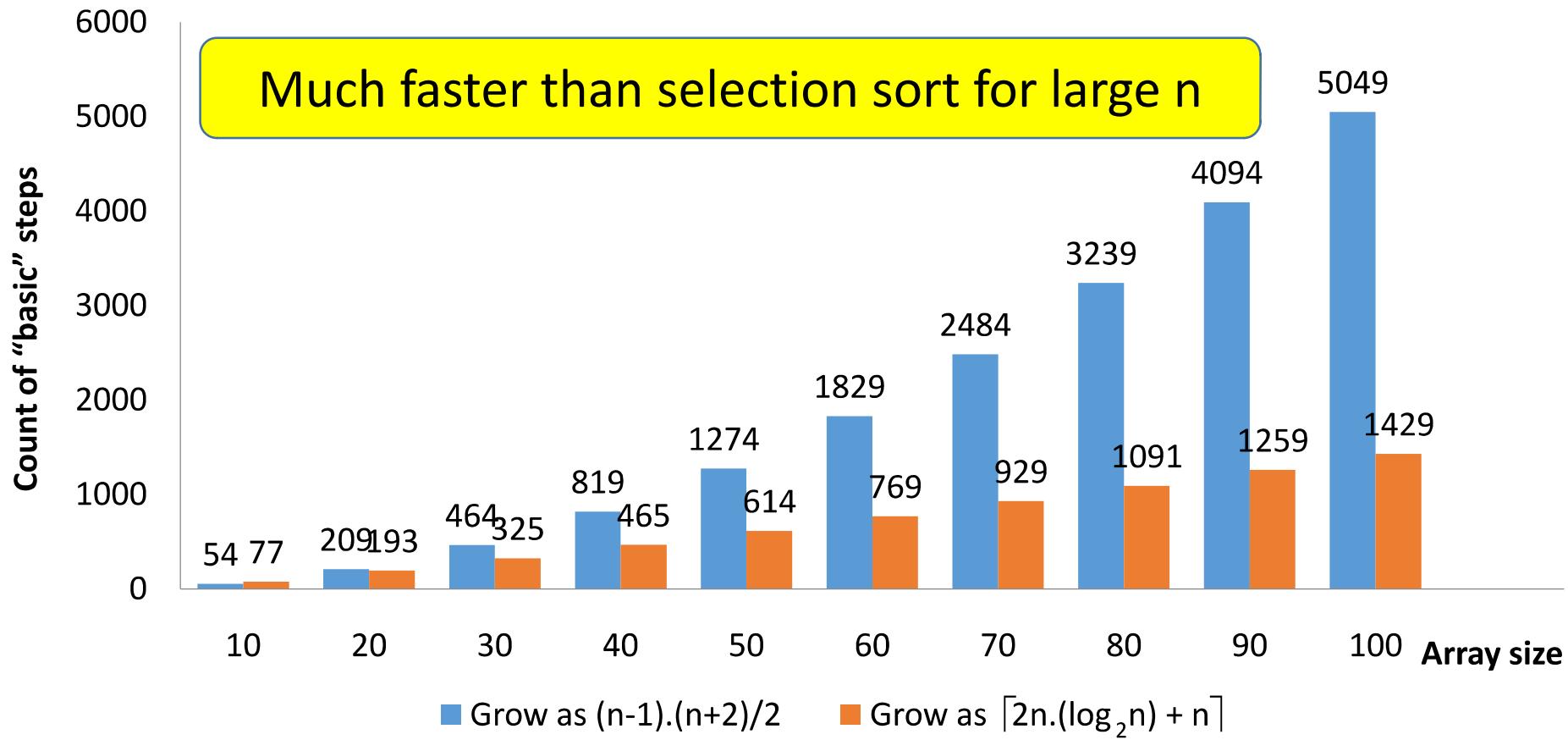
```
// PRECONDITION: start < end, both within array bounds of A
void mergeSort(int A[], int start, int end) {
    if (end == start +1) { return; } //Subarray of interest has 1 element
    int mid = (start + end)/2; //Get mid-index of subarray of interest
    mergeSort(A, start, mid); // Sort subarray A[start] ... A[mid-1]
    mergeSort(A, mid, end); // Sort subarray A[mid] ... A[end-1]
    // Merge sorted subarrays A[start] ... A[mid-1] and A[mid] ... A[end-1]
    mergeSortedSubarrays(A, start, mid, end);
    return;
}
// POSTCONDITION: A[start] ... A[end-1] sorted in decreasing order
```

# Counting “Basic” Steps in Merge Sort



- Let  $T_n$  denote maximum count of “basic” steps to merge sort an array of size  $n$
  - $T_n = T_{n/2} + T_{n/2} + 2n$
- 
- mergeSortedSubarrays**
- Second recursive call of mergeSort**
- First recursive call of mergeSort**
- Solution of recurrence:**  
 $T_n \approx \lceil 2n \cdot \log_2 n + n \rceil$

# Comparing “Basic” Steps With Selection Sort



# Summary

---



- Merge sort
  - C++ implementation using recursion
  - Analysis of performance
    - Counting “basic” steps in sorting an array of size  $n$
  - Clear winner performance-wise over selection sort