EE2331 Data Structures and Algorithms

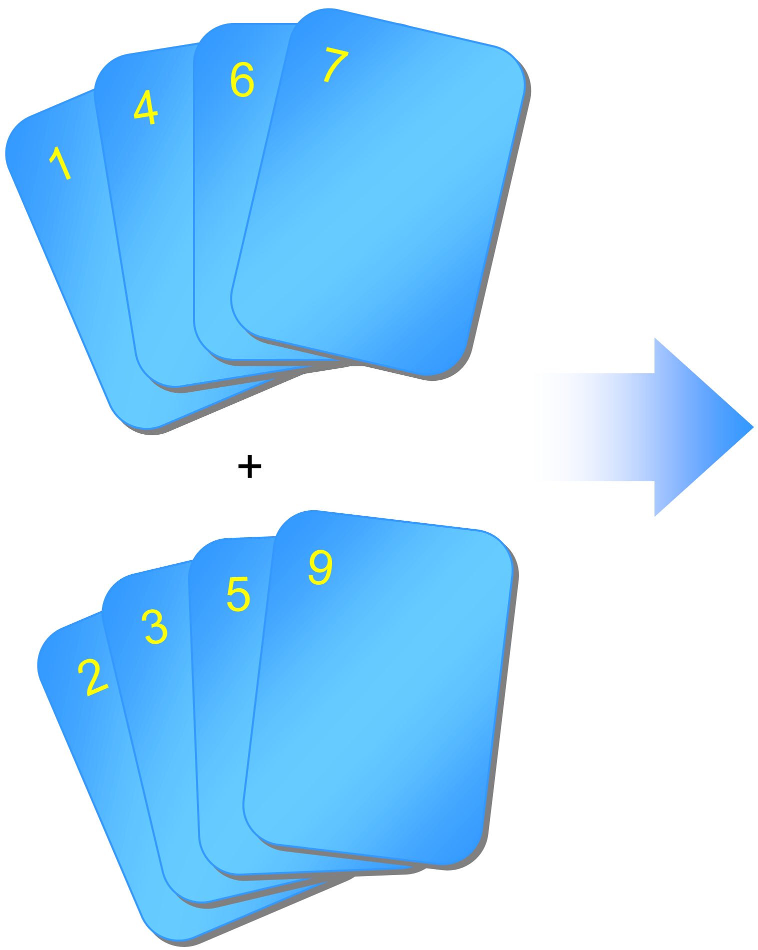
Merge sort

Merge Sort

Is it faster or slower than insertion sort?

Daily Life Example

- The idea of merging

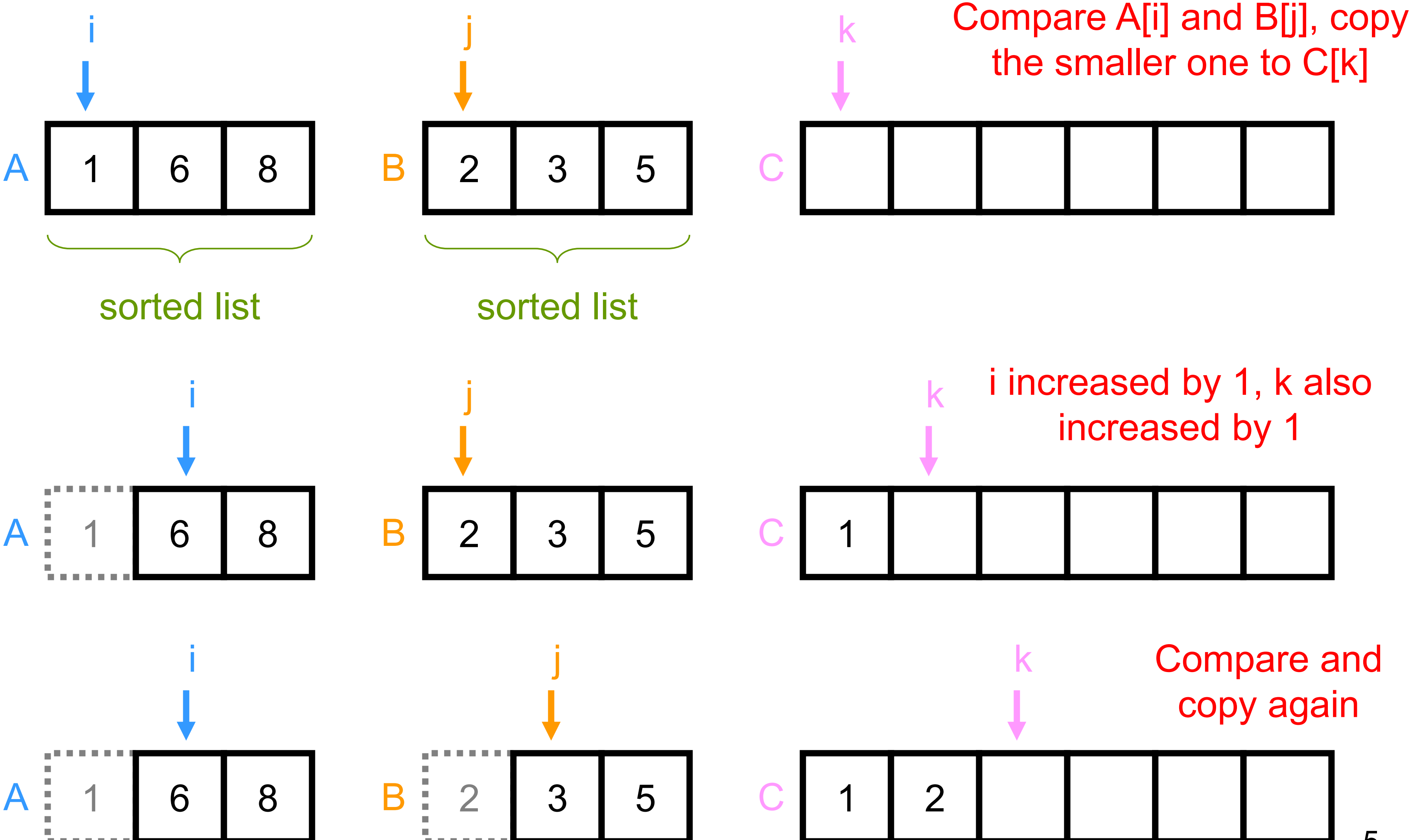


Any pre-condition?

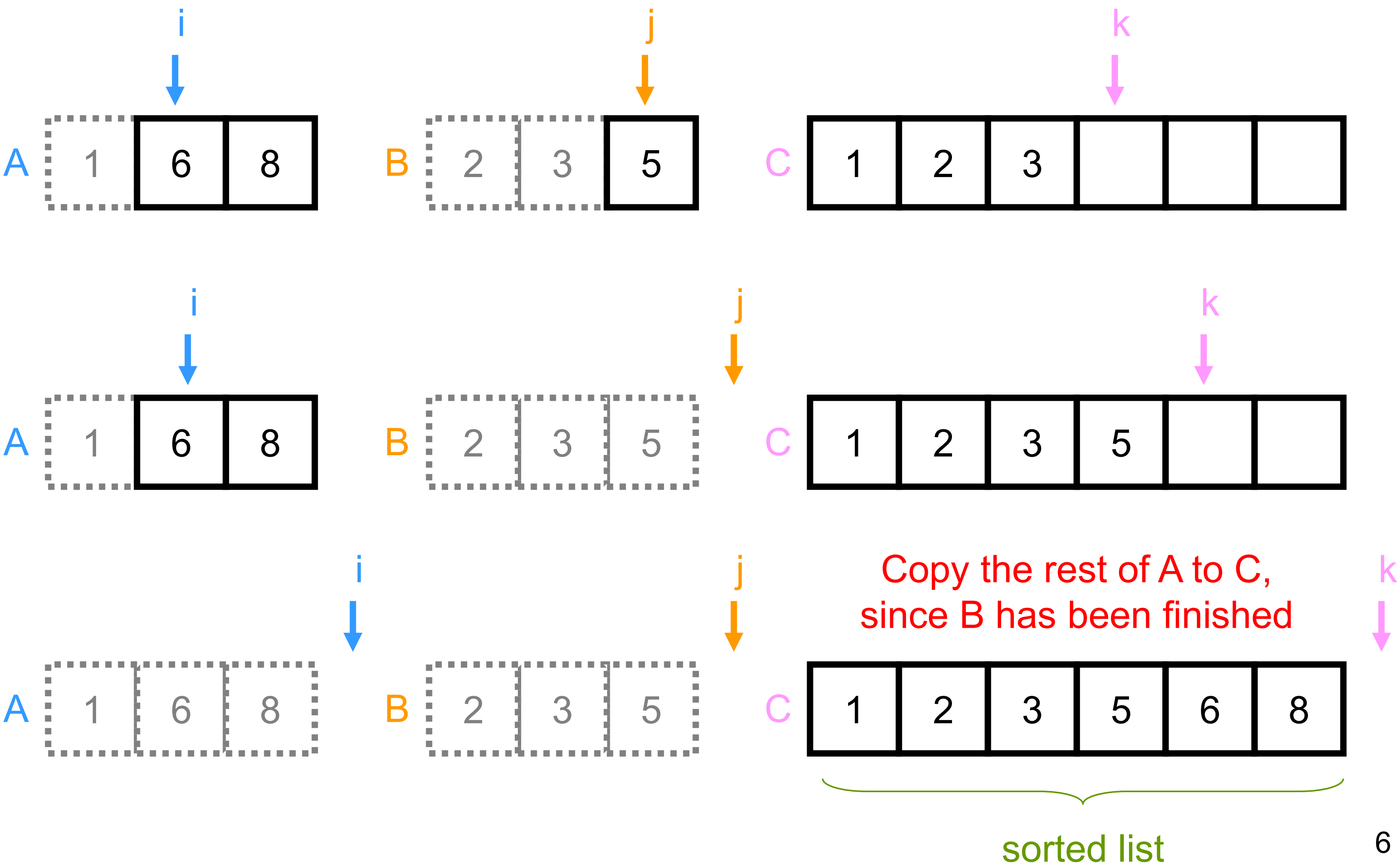
Merging

- To merge 2 **sorted** lists
- It takes 2 input arrays *A*[] & *B*[], 1 output array *C*[] and 3 counters (i, j, k) for the arrays respectively
- The smaller of *A*[i] and *B*[j] is copied to *C*[k], then the counters are advanced
- If either *A*[] or *B*[] finishes first, the reminder of the other array is copied to *C*[]

Merge Sort Example

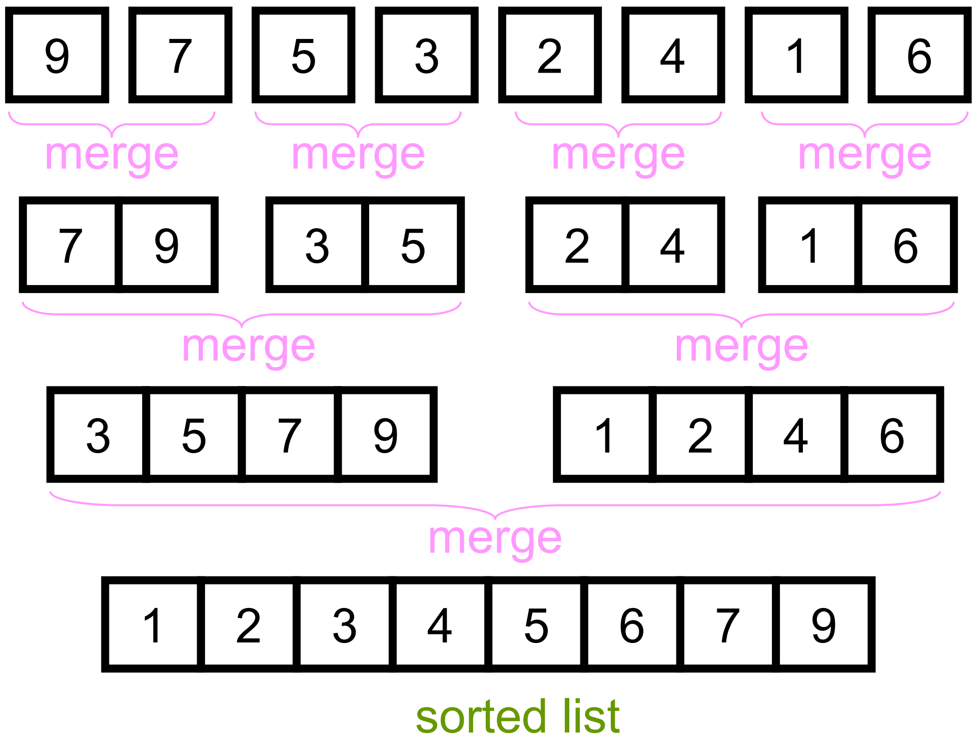


Merge Sort Example

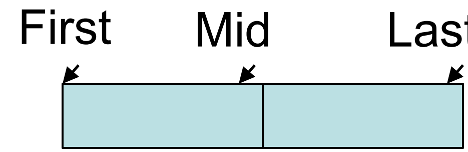


The Algorithm

- Initially the input list is divided into *N* sublists of size 1
- Adjacent pairs of lists are merged to form larger sorted sublists
- The merging process is repeated until there is only one list



Merge Adjacent Lists




```
void merge(int data[], int first, int mid, int last) {
    int temp[SIZE], i = first, j = mid + 1, k = 0;
    while (i <= mid && j <= last) {
        if (data[i] <= data[j])
            temp[k++] = data[i++];
        else
            temp[k++] = data[j++];
    }
    while (i <= mid) temp[k++] = data[i++];
    while (j <= last) temp[k++] = data[j++];
    i = 0;
    while (i < k) data[first+i] = temp[i++];
}
```

Compare A[i] and B[j], copy the smaller one to temp[k]

A is data[first…mid]

B is data[mid+1…last]

C is temp[…]

The remaining A or B will be copied into temp

The sorted temp. array is copied back to data

What is the running time complexity?

Divide-and-Conquer

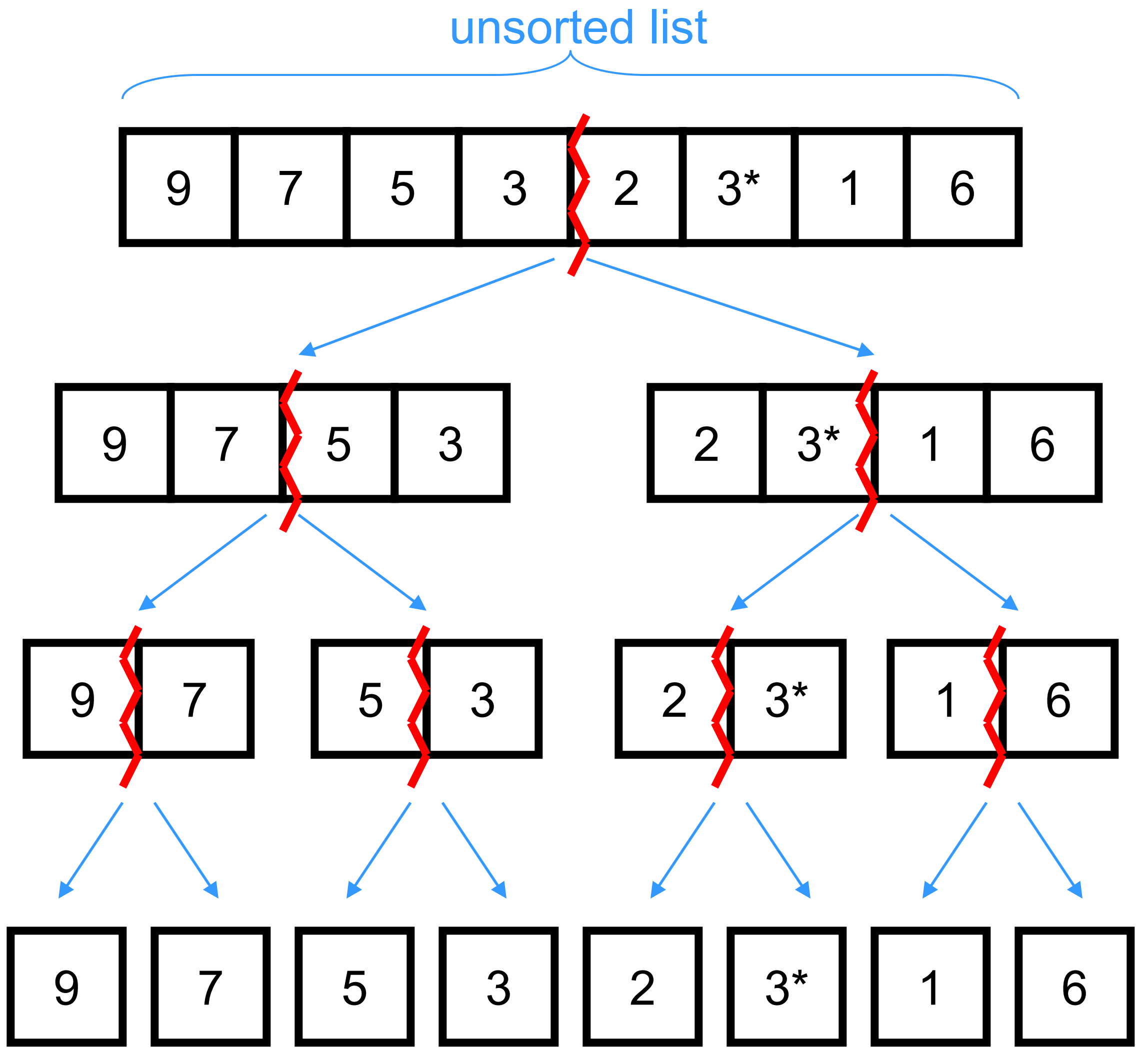
- This algorithm is a classic divide-and-conquer strategy
- Very powerful use of recursion
- The problem is divided into smaller problems and solved independently and recursively
- The conquering phase consists of merging together the sorted lists

Divide-and-Conquer

- Recursive structure
  - Step1: Divide the problem into a number of sub-problems
  - Step 2: Conquer the sub-problem by solving them recursively. If the subproblem sizes are small enough, solve the subproblem in a straightforward way
  - Step 3: Combine the solutions to the sub-problems into the solution for the original problem.

Dividing Phase

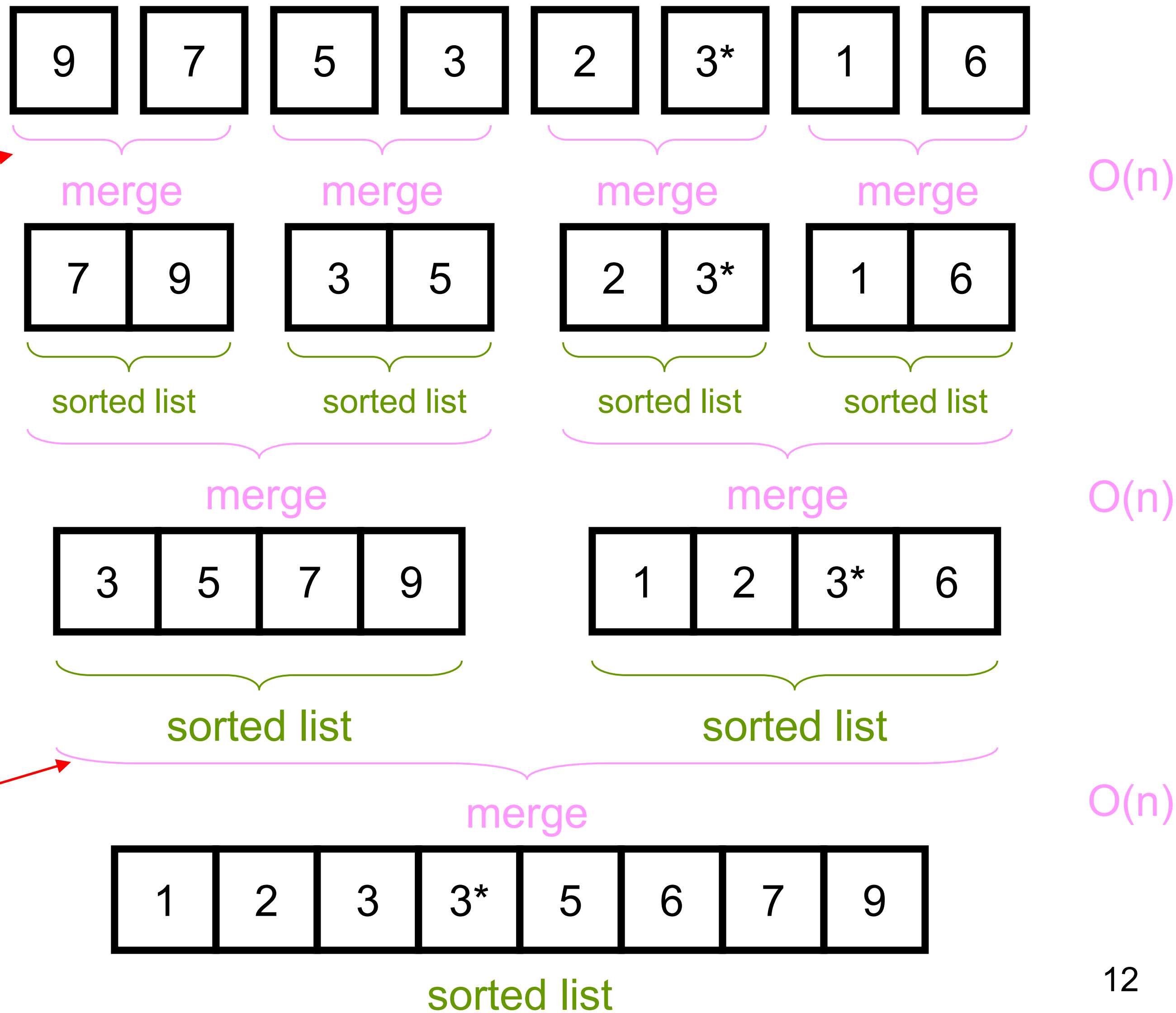
Divide the list into halves



Divide again and again until only one single element left in the list

Conquering Phase

Merge the small sorted lists into larger sorted lists



Merge again and again until one single list left

Merge Sort (Using Recursion)

```
void mergesort(int data[], int first, int last) {
    int mid = (first + last ) / 2;
    if (first >= last) return;          //base case: size = 1
    mergesort(data, first, mid);    //recursion: divide the list into halves
    mergesort(data, mid+1, last);//recursion: divide the list into halves
    merge(data, first, mid, last);  //start merging the list: conquer
}
int main(...) {
    int data[] = {8, 5, 9, 6, 3};
    mergesort(data, 0, 4);
    return 0;
}
```

In-class exercise:
Write all the called mergesort() functions and the parameters of mergesort.

For example, the first called mergesort function is mergesort(data, 0, 4).

Merge Sort (Using Recursion)

```
void mergesort(int data[], int first, int last) {
    int mid = (first + last ) / 2;
    if (first >= last) return;          //base case: size = 1
    mergesort(data, first, mid);        //recursion: divide the list into halves
    mergesort(data, mid+1, last);//recursion: divide the list into halves
    merge(data, first, mid, last);  //start merging the list: conquer
}
int main(…) {
    int data[] = {8, 5, 9, 6, 3};
    mergesort(data, 0, 4);
    return 0;
}
```

In-class exercise:
Write all the called mergesort() functions and the parameters of mergesort.

For example, the first called mergesort function is mergesort(data, 0, 4).

The following functions are called by the given order.
Mergesort(data, 0,4), mergesort(data,0,2), mergesort(data,0,1), mergesort(data,0,0), mergesort(data,1,1), merge(data,0,0,1),mergesort(2,2),merge(data,0,1,2),mergesort(data,3,4),mergesort(data,3,3),mergesort(data,4,4),merge(data,3,3,4),merge(data,0,2,4)

```
void mergesort(int data[], int first, int last) {
    int mid = (first + last ) / 2;
    if (first >= last) return;
    mergesort(data, first, mid);
    mergesort(data, mid+1, last);
    merge(data, first, mid, last);
}
int main(…) {
    int data[] = {8, 5, 9, 6, 3};
    mergesort(data, 0, 4);
    return 0;
}
```

```
void merge(int data[], int first, int mid, int last) {
    int temp[SIZE], i = first, j = mid + 1, k = 0;
    while (i <= mid && j <= last) {
        if (data[i] <= data[j])
            temp[k++] = data[i++];
        else
            temp[k++] = data[j++];
    }
    while (i <= mid) temp[k++] = data[i++];
    while (j <= last) temp[k++] = data[j++];
    i = 0;
    while (i < k) data[first+i] = temp[i++];
}
```

Running Time Analysis

statement		Times
	void mergesort(int data[], int first, int last) {	T(n)
1	if (first >= last) return;	$\Theta(1)$
2	mergesort(data, first, (first + last) / 2);	$T(\frac{n}{2})$
	// (first + last) / 2 is the mid-point	
3	mergesort(data, (first + last) / 2+1, last);	$T(\frac{n}{2})$
4	merge(data, first, mid, last);	$\Theta(n)$
	}	

$$T(n)=\begin{cases}\Theta(1) + 2\cdot T(\frac{n}{2}) + \Theta(n) & , \ otherwise \\ 1, & , \quad when\ n = 1\end{cases}$$

Running Time Analysis

$$T(n) = 1 + 2\cdot T(\frac{n}{2}) + n$$
$$= 1+2+ 4\cdot T(\frac{n}{4})+n+n$$
$$= 4\cdot T(\frac{n}{4})+2n+1+2$$
$$= 4+ 8\cdot T(\frac{n}{8})+n+2n+1+2$$
$$= 8\cdot T(\frac{n}{8})+3n+1+2+4$$

… …

$$= 2^k\cdot T(\frac{n}{2^k})+kn+(2^0 + 2^1 + 2^2 + \cdots + 2^{k-1})$$

… …

until $\frac{n}{2^k} = 1 \Rightarrow$ k=$\log_2 n$

$$T(x) = 1 + 2\cdot T(\frac{x}{2}) + x$$
$$T(\frac{x}{2}) = 1 + 2\cdot T(\frac{x}{4}) + \frac{x}{2}$$
$$2T(\frac{x}{2}) = 2 + 4\cdot T(\frac{x}{4}) + x$$
$$T(\frac{x}{4}) = 1 + 2\cdot T(\frac{x}{8}) + \frac{x}{4}$$
$$4T(\frac{x}{4}) = 4 + 8\cdot T(\frac{x}{8}) + x$$

… …

Running Time Analysis

When k=$\log_2 n$,

T(n)=$2^k \cdot T(\frac{n}{2^k})$+kn+$(2^0 + 2^1 + 2^2 + \cdots + 2^{k-1})$

=$2^{\log_2 n} \cdot T(1)$+n $\log_2 n$ +$(2^0 + 2^1 + \cdots + 2^{\log_2 n - 1})$

=n+ n $\log_2 n$ + n – 1 = $\underbrace{n \log_2 n}_{\text{merge}} + \underbrace{2n - 1}_{\text{\# of statement 1 (testing whether the input size=1)}}$

= $\mathbf{O(n\boldsymbol{log_2 n})}$

merge

# of statement 1 (testing whether the input size=1)

$S = 2^0 + 2^1 + \cdots + 2^{\log_2 n - 1}$

$2S = 2^1 + 2^2 + \cdots + 2^{\log_2 n}$

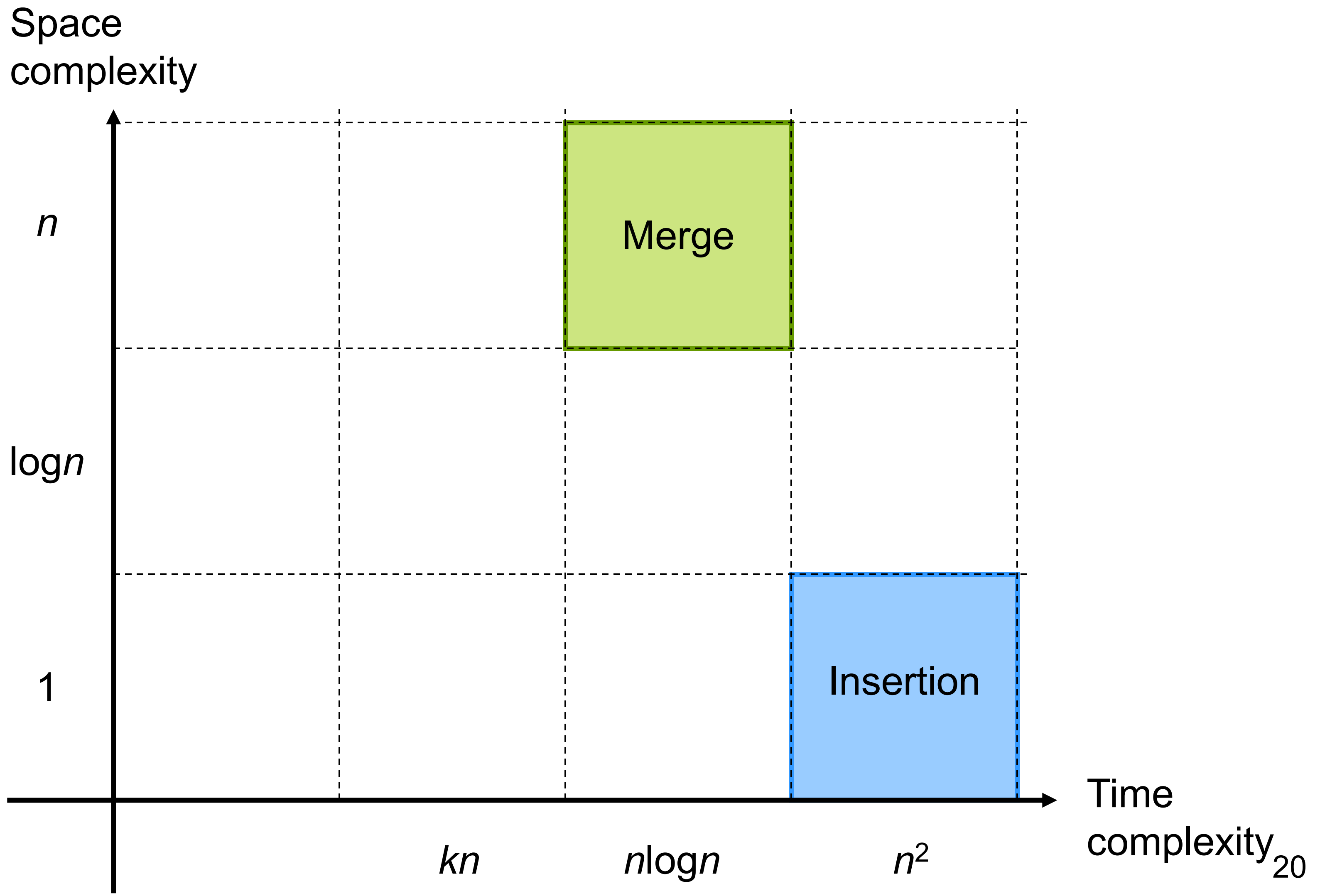
$S = 2S\text{-}S = 2^{\log_2 n} - 2^0 = n - 1$

Complexity Analysis

- Merge sort goes through the same steps - independent of the data
  - Best case = Worst case = Average case
- For each runs, it requires O(n) time to finish
- There are $\log_2 n$ runs in total
- The time complexity is O($n\log n$)
- Faster than **insertion sort**!

- The trade-off is it needs extra memory to hold the temporary sorted result
- Space complexity = O(n)
- Improvement to the merge algorithm:
  - Instead of merging each set of lists from data[] to temp[] and then copy temp[] back to data[], alternate merge passes can be performed from data[] to temp[] and from temp[] to data[].

Summary


Space
complexity
n
logn
1
Merge
Insertion
Time
complexity
kn
nlogn
n2