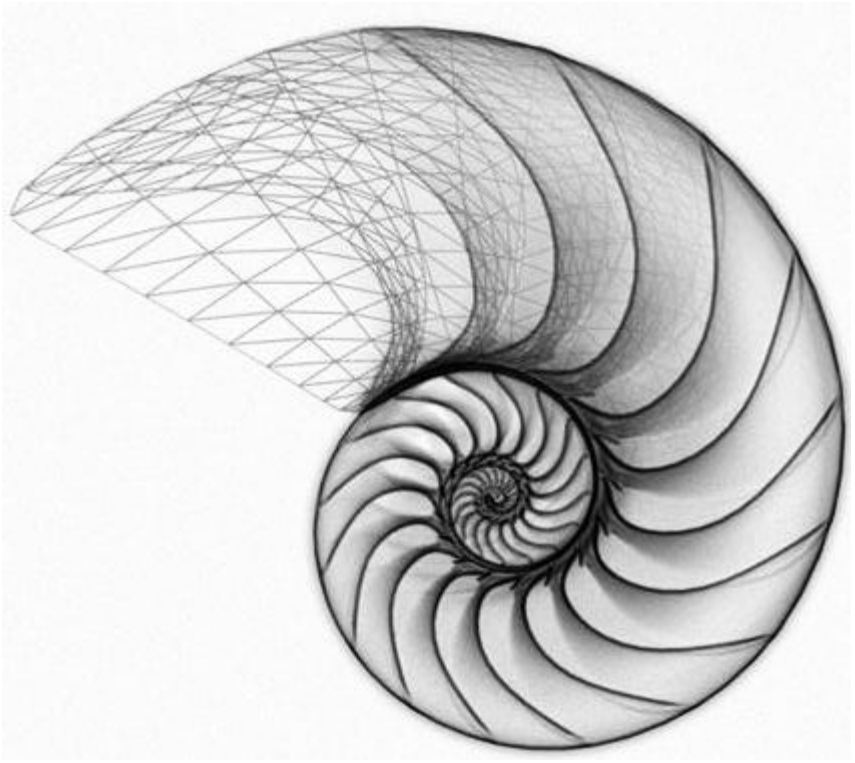
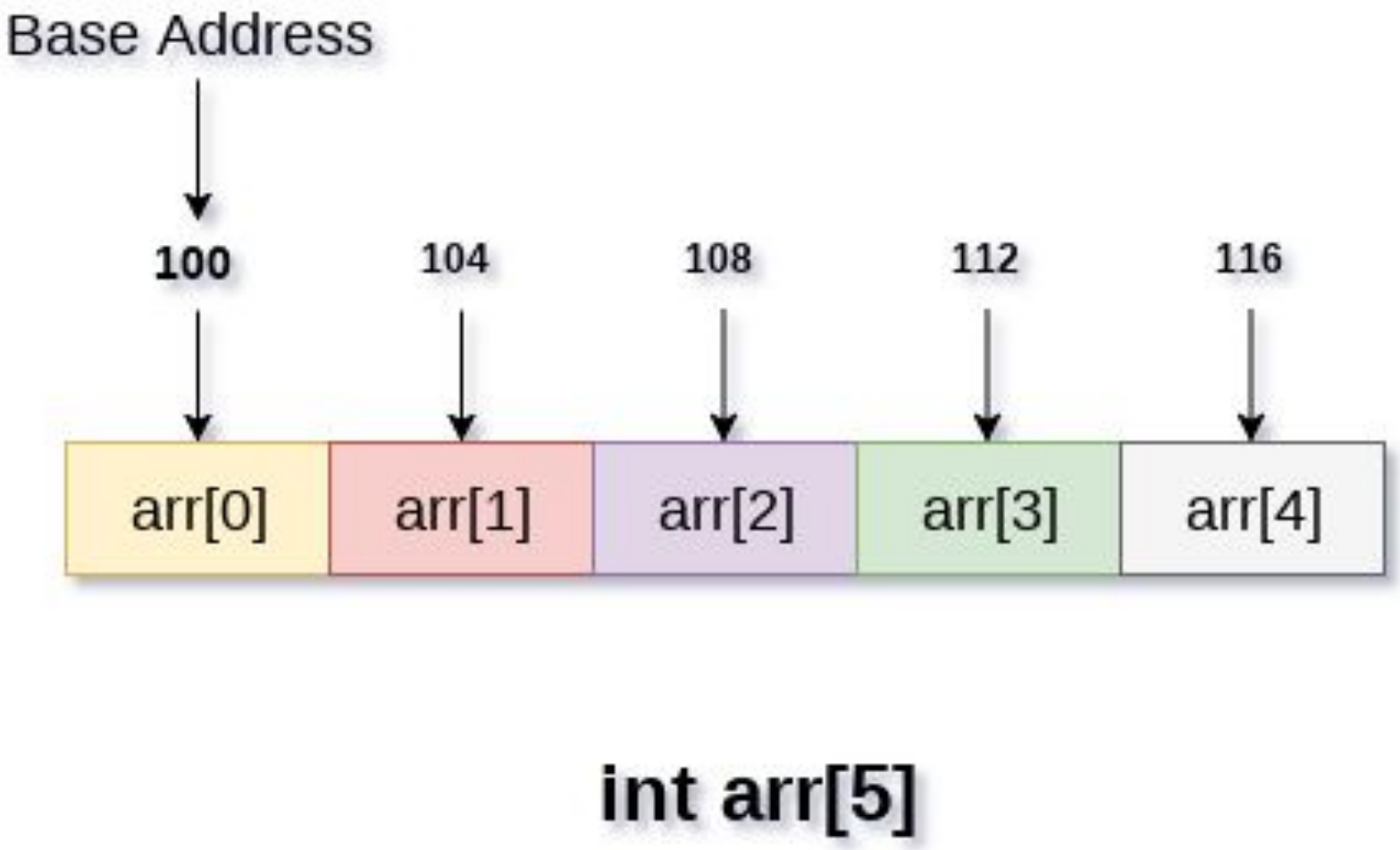


Lecture 10

1D Array

A One-Dimensional Array is the simplest form of an Array in which the elements are stored linearly and can be accessed individually by specifying the index value of each element stored in the array.

1D Array

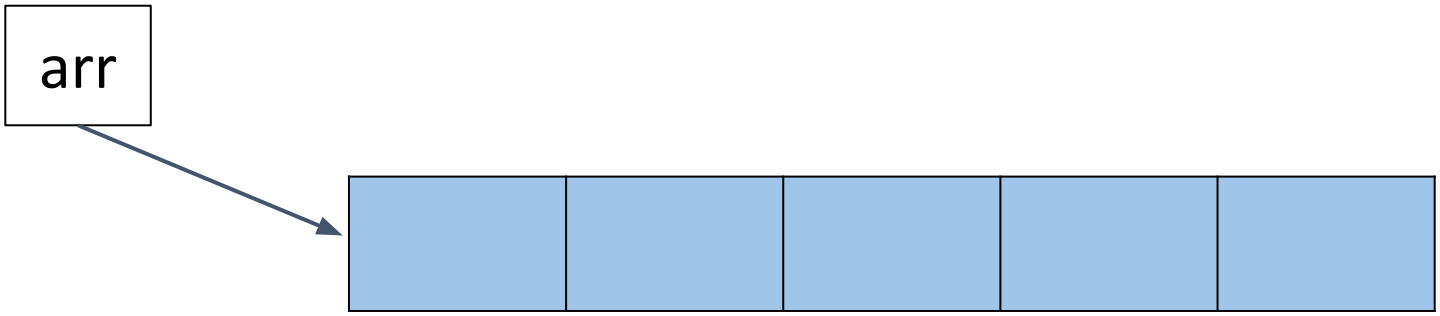

Base Address
100
104
108
112
116
arr[0]
arr[1]
arr[2]
arr[3]
arr[4]
int arr[5]

1D Array

int arr[5];

1D Array

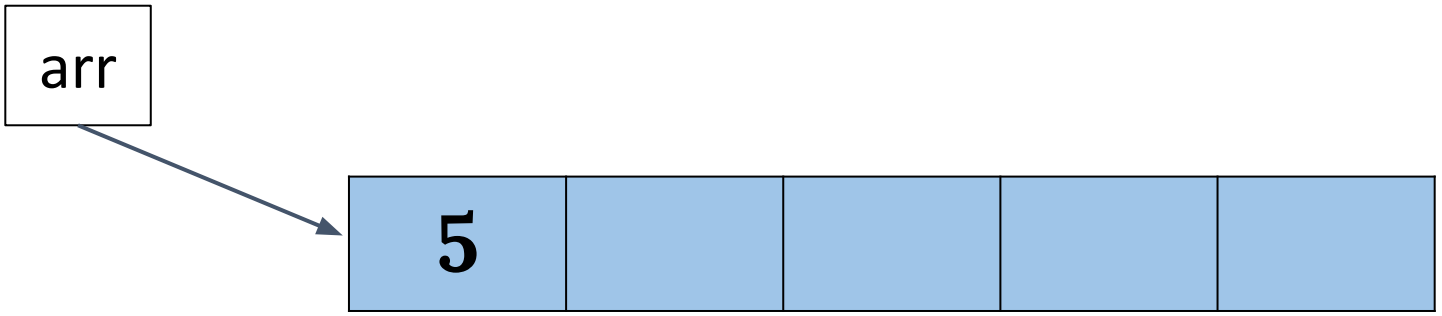
int arr[5];



1D Array

int arr[5];

arr[0] = 5;

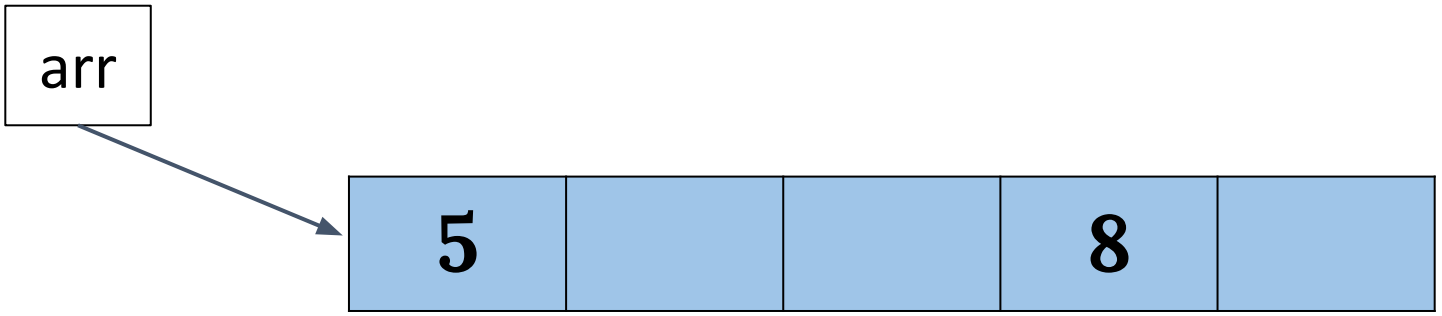


1D Array

int arr[5];

arr[0] = 5;

arr[3] = 8;

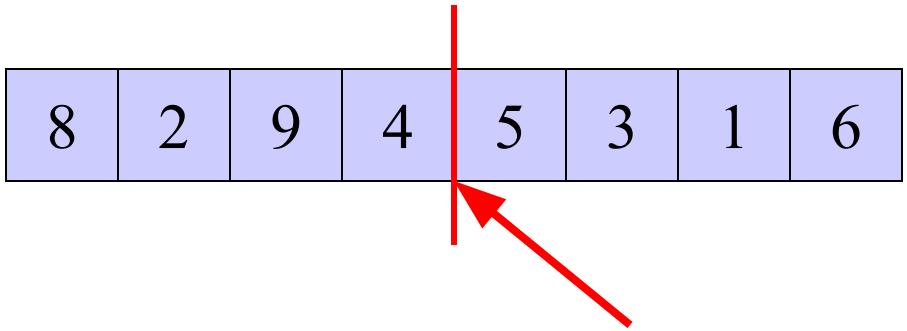


“Divide and Conquer”

“Divide and Conquer”

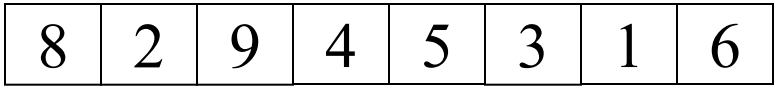
- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves Mergesort
- **Idea 2 :** Partition array into items that are “small” and items that are “large”, then recursively sort the two sets Quicksort

Mergesort

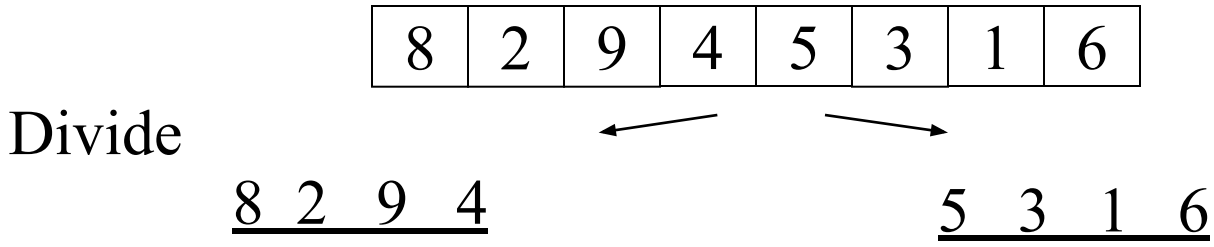


- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

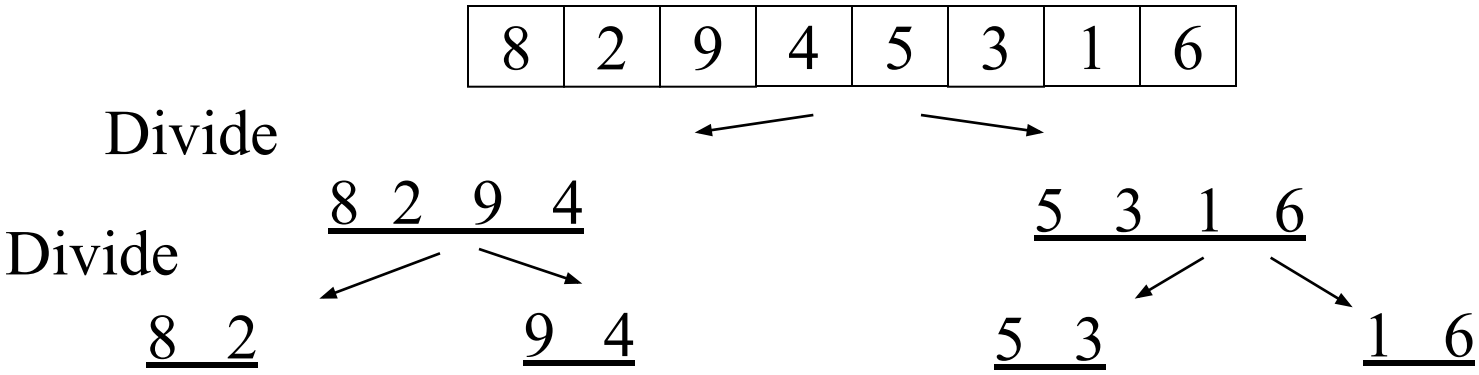
Mergesort Example



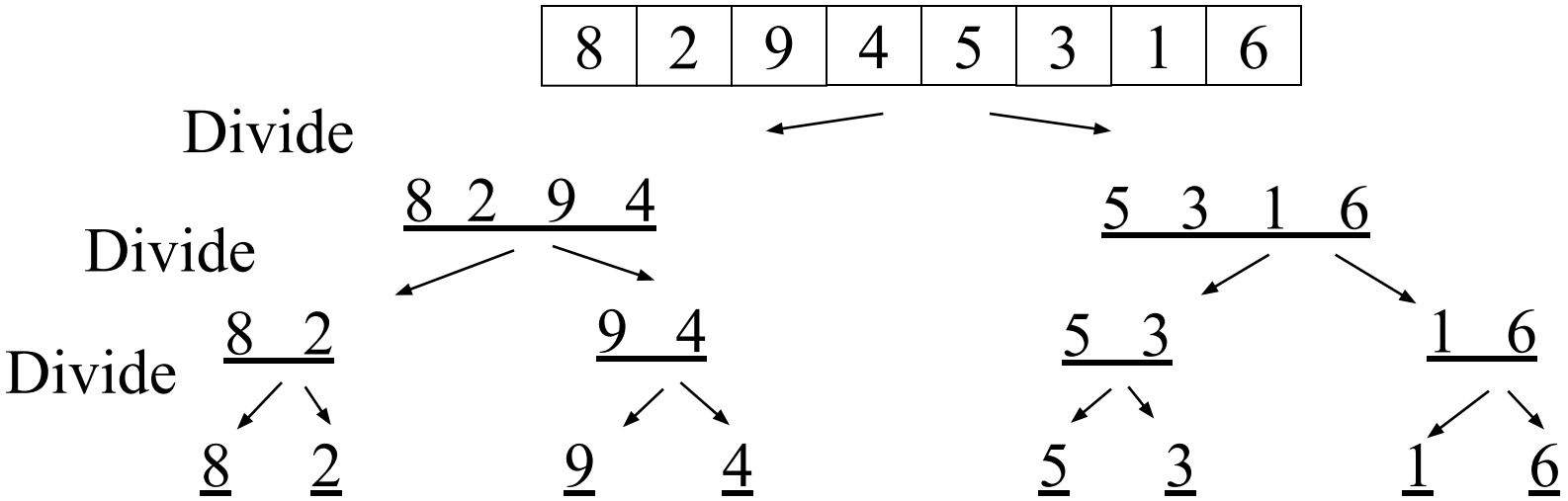
Mergesort Example



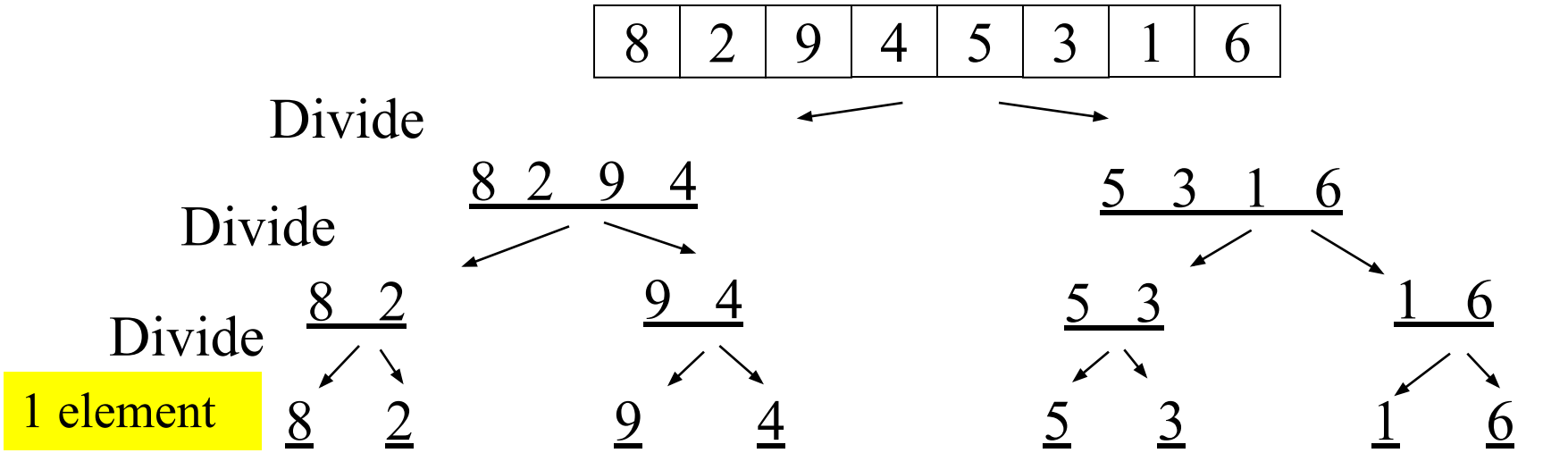
Mergesort Example



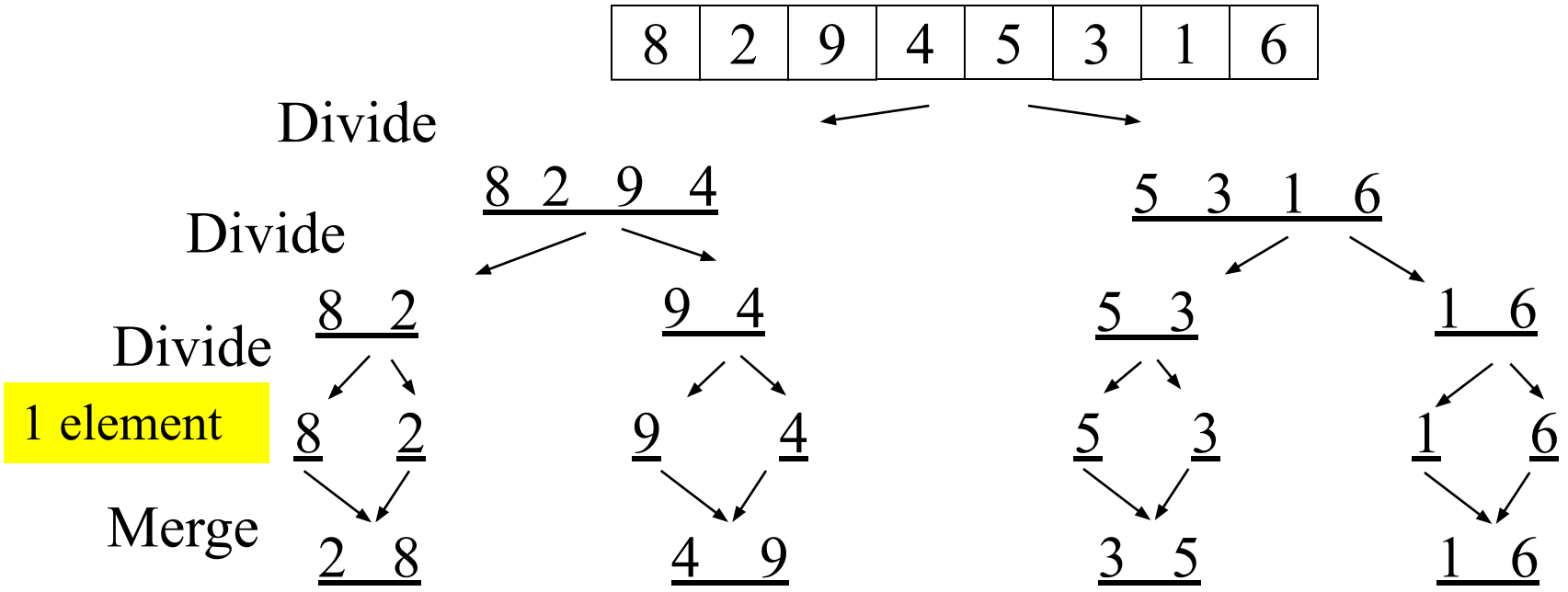
Mergesort Example



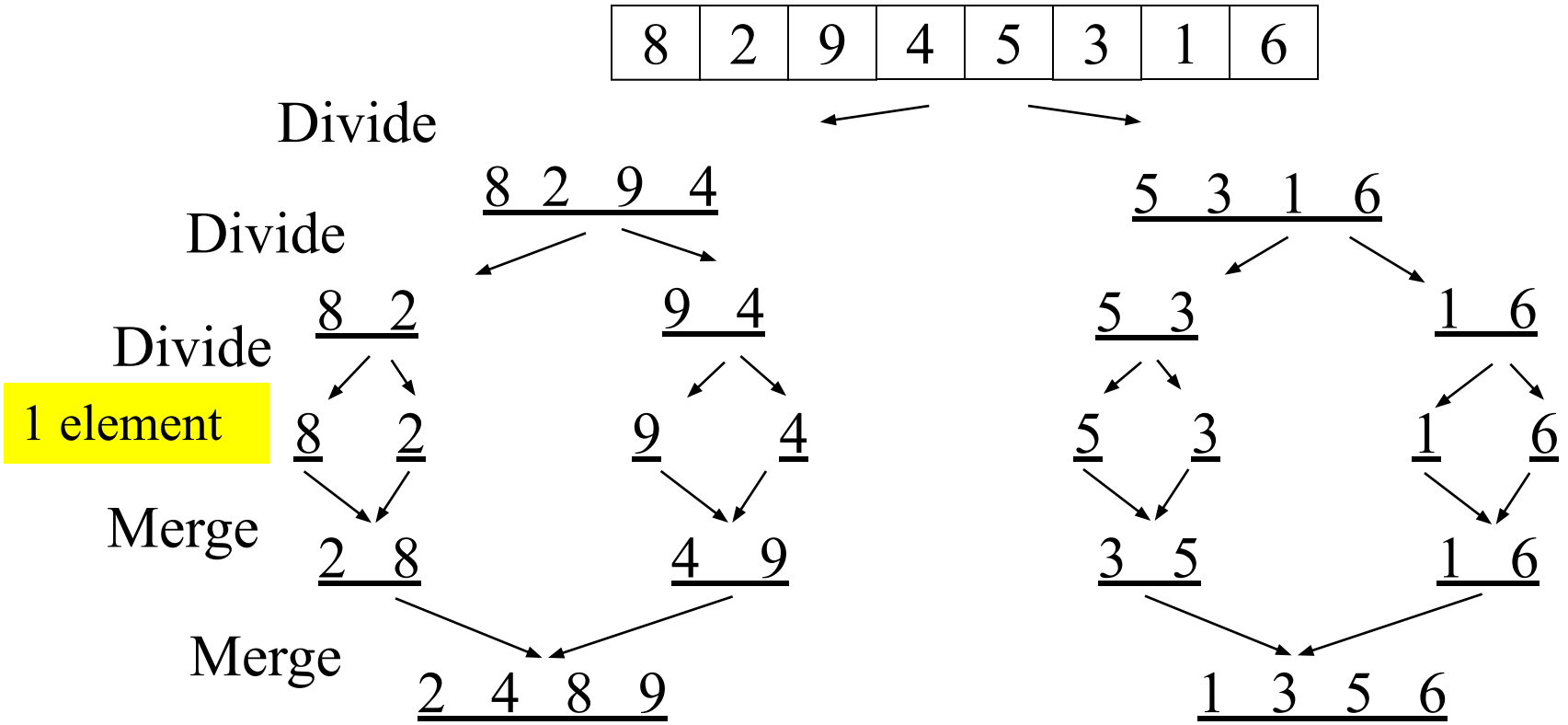
Mergesort Example



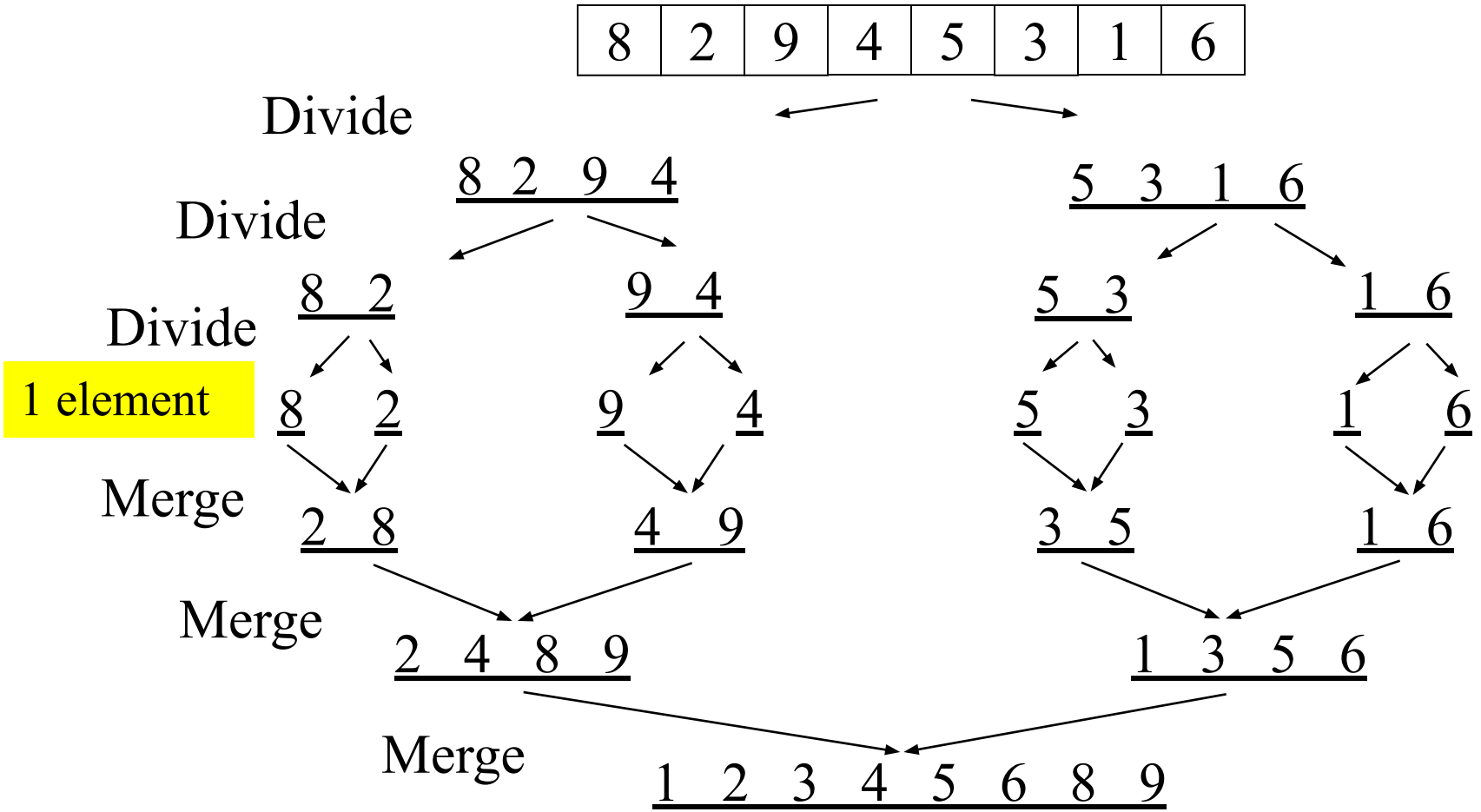
Mergesort Example



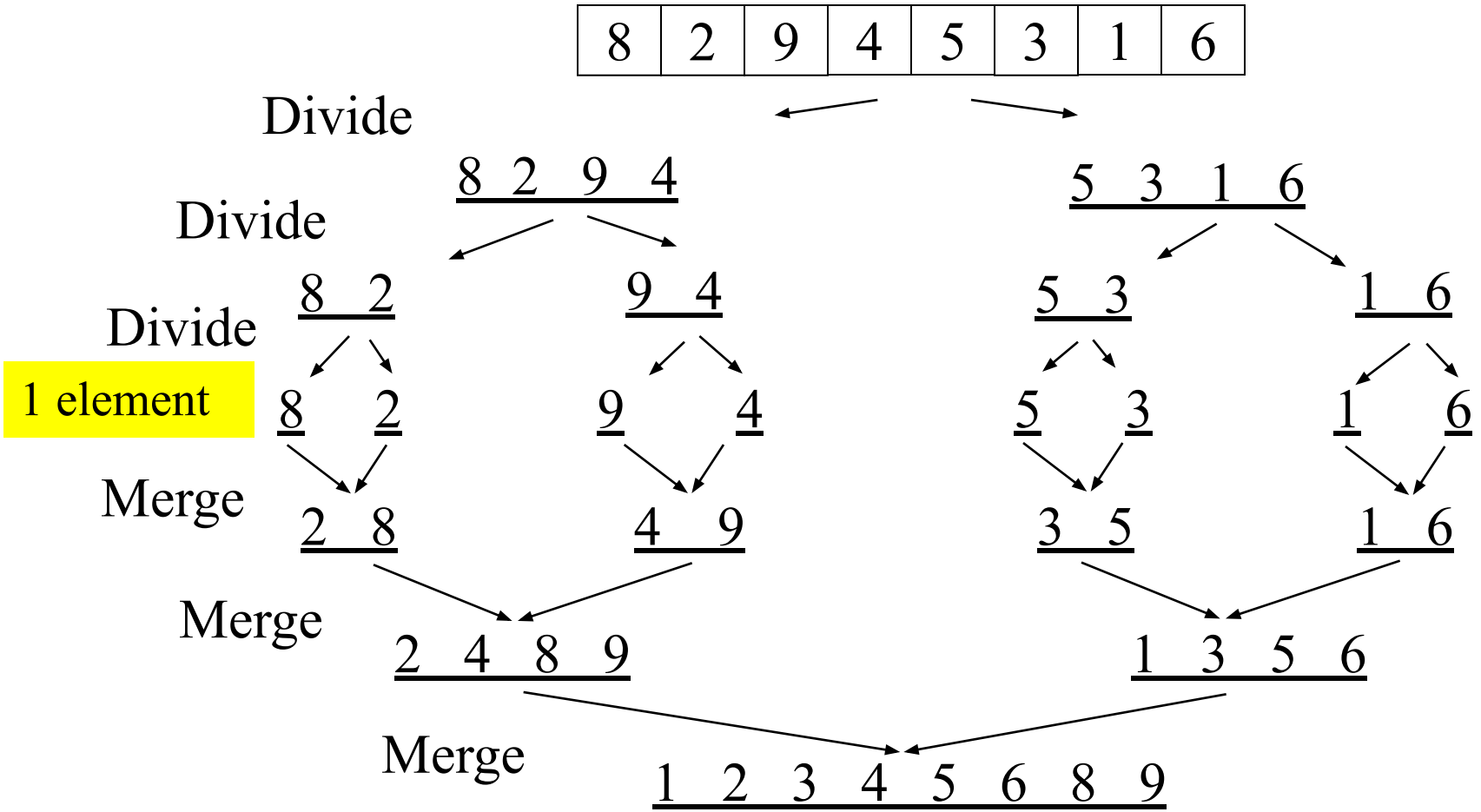
Mergesort Example



Mergesort Example



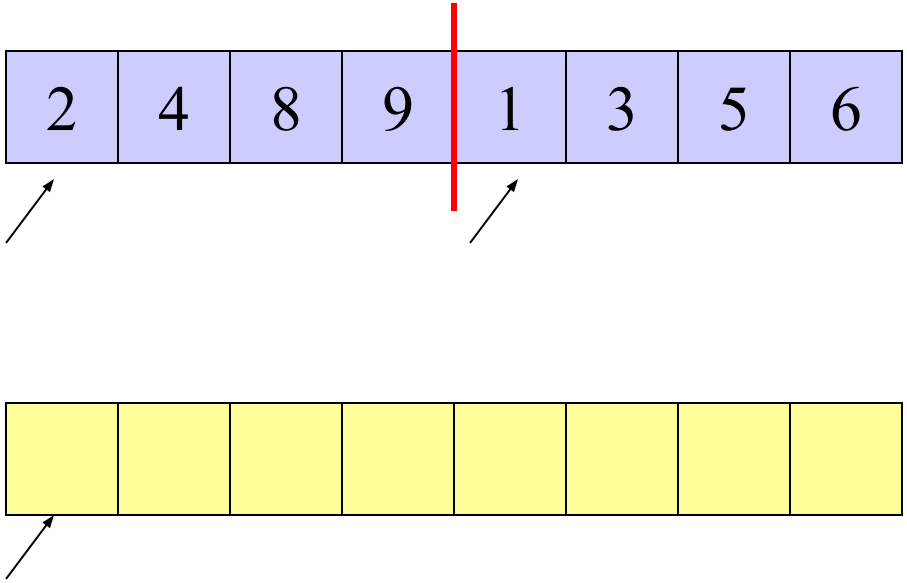
Mergesort Example



Exercise

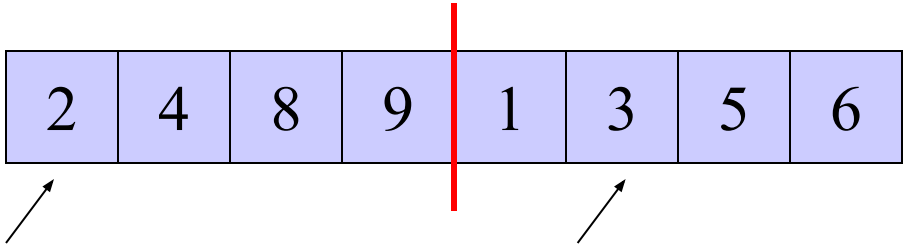
Auxiliary Array

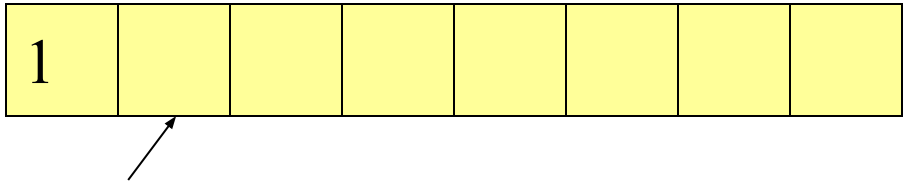
- The merging requires an auxiliary array.



Auxiliary Array

- The merging requires an auxiliary array.

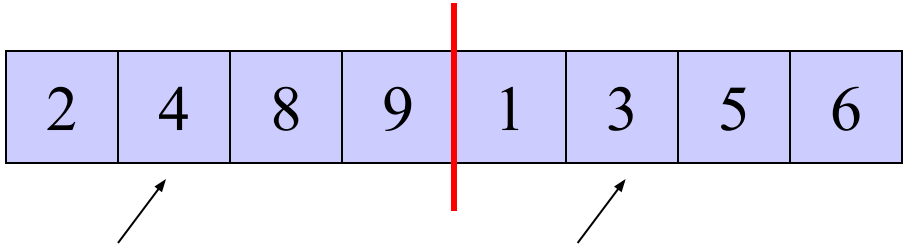


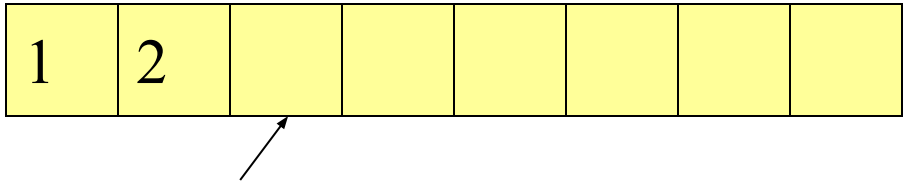


Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

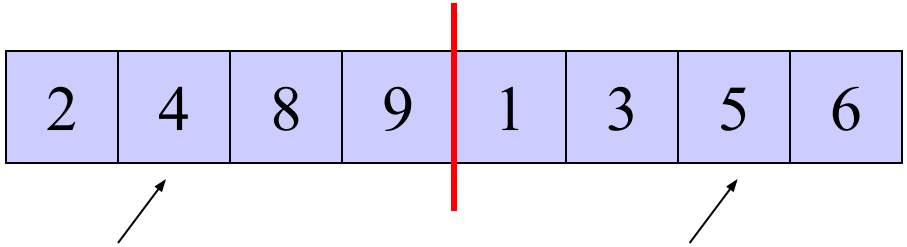


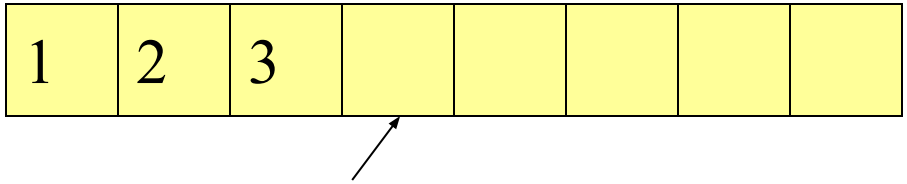


Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

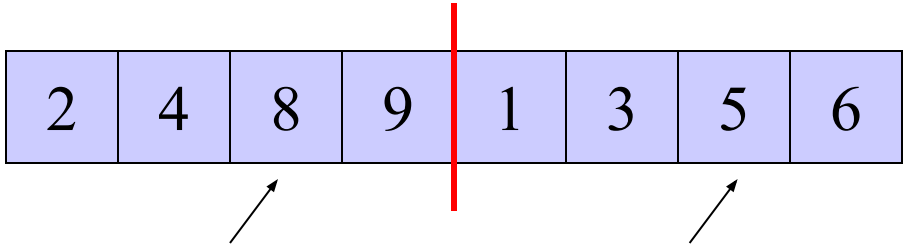


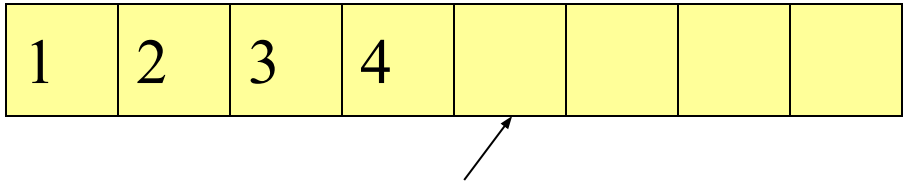


Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

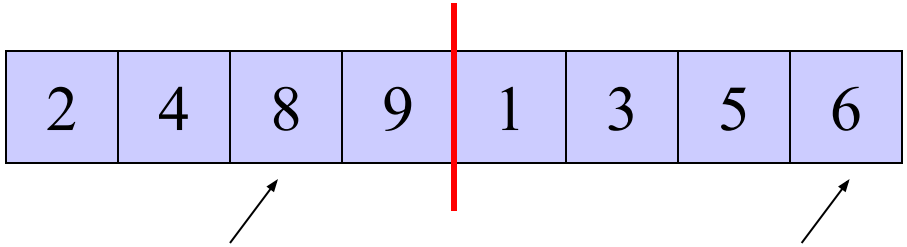


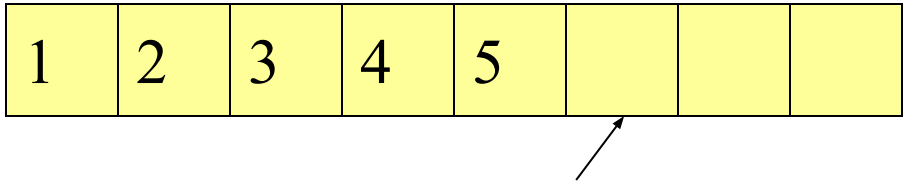


Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

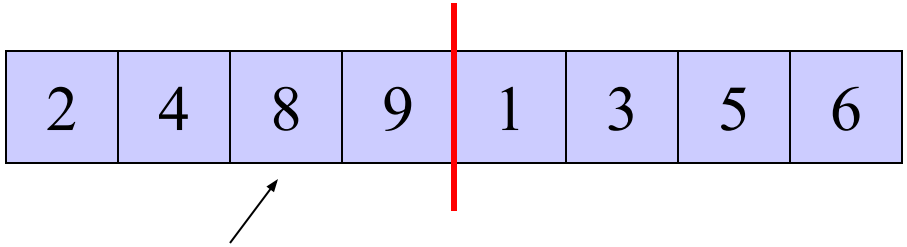


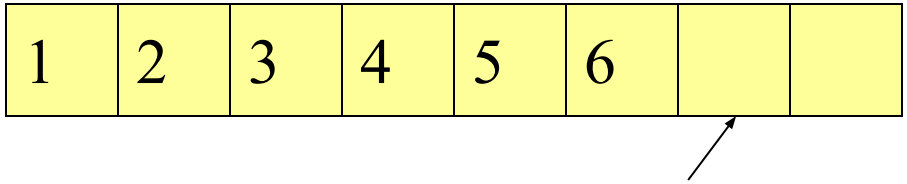


Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

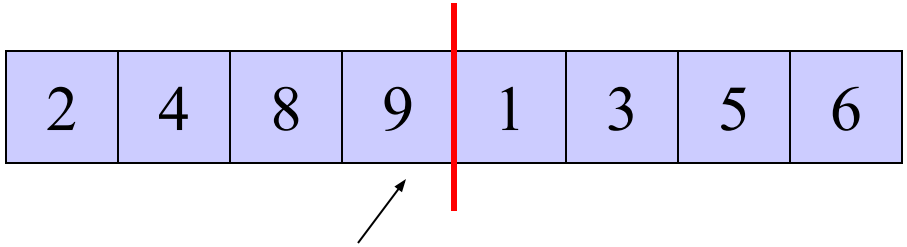


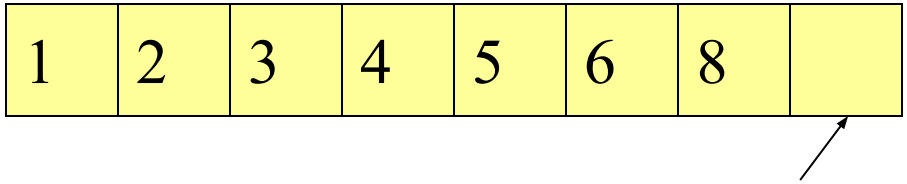


Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.

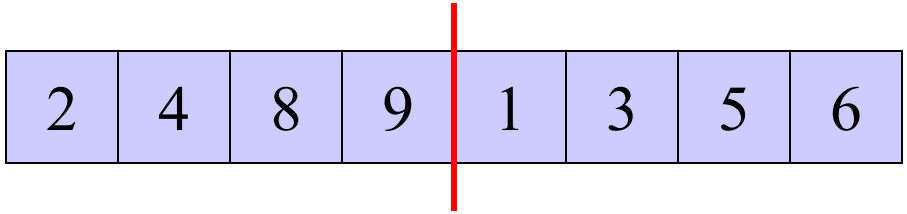


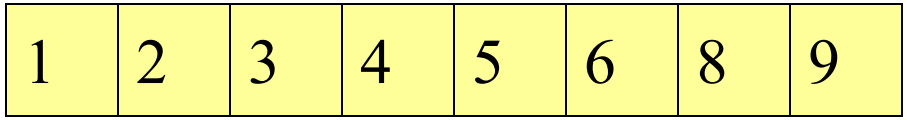


Auxiliary array

Auxiliary Array

- The merging requires an auxiliary array.





Auxiliary array

Exercise

Mergesort Analysis

- Let T(N) be the running time for an array of N elements
- Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array
- Each recursive call takes T(N/2) and merging takes O(N)

Mergesort Recurrence Relation

- The recurrence relation for T(N) is:
  - › $T(1) \leq a$
    - base case: 1 element array constant time
  - › $T(N) \leq 2T(N/2) + bN$
    - Sorting N elements takes
      - the time to sort the left half
      - plus the time to sort the right half
      - plus an O(N) time to merge the two halves
- $T(N) = O(n \log n)$ (recall from previous lecture)

Properties of Mergesort

- Not in-place
  - › Requires an auxiliary array (O(n) extra space)

Properties of Mergesort

- Not in-place
  - › Requires an auxiliary array (O(n) extra space)

Properties of Mergesort

- Not in-place
  - › Requires an auxiliary array (O(n) extra space)

Algorithm of Mergesort

```
step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
          if left > right
              return
          mid= (left+right)/2
          mergesort(array, left, mid)
          mergesort(array, mid+1, right)
          merge(array, left, mid, right)
step 4: Stop
```

Mergesort

```
void mergeSort(int *arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}



void show(int *arr, int size) {
    for (int i = 0; i < size; i++)
        std::cout << arr[i] << " ";
    std::cout << "\n";
}
```

Mergesort

```
void merge(int *arr, int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1, n2 = r - m;
    int *L = new int[n1], *R = new int[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0; j = 0; k = l;
    while (i < n1 || j < n2) {
        if (j >= n2 || (i < n1 && L[i] <= R[j])) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    delete[] L;
    delete[] R;
}
```

Mergesort

```
/** Main function */
int main() {
   int size;
   std::cout << "Enter the number of elements : ";
   std::cin >> size;
   int *arr = new int[size];
   std::cout << "Enter the unsorted elements : ";
   for (int i = 0; i < size; ++i) {
       std::cin >> arr[i];
   }
   mergeSort(arr, 0, size - 1);
   std::cout << "Sorted array : ";
   show(arr, size);
   delete[] arr;
   return 0;
}
```