Search

- Binary search

```
Bool find(int a[], int n, int t){
int l = 0;
Int r = n-1;
Int m;

While (l <= r){
        m = l + (r-l)/2;
        if (a[m] == t) return true;
        if (a[m] < t) l = m + 1;
        else r=m-1;
}
Return false;}
```

$l+(r-l)/2$

$= l + \frac{r-l}{2}$

$= \frac{2l}{2} + \frac{r-l}{2}$

$= \frac{2l+r-l}{2}$

$= \frac{l+r}{2}$

Sorting

- Insertion Sort
- Merge Sort
- Quick Sort

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 0: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	**7.42**	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 1: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	0.56	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 2: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	2.78	1.12	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Iteration 3: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	2.78	1.17	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

Iteration 4: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	2.78	0.32	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	1.17	0.32	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	1.12	0.32	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.56	0.32	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 3.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 4.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Iteration 5: step 5.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	`0.32`	`0.56`	`1.12`	`1.17`	`2.78`	`6.21`	`7.42`	`4.42`	`3.14`	`7.71`

Iteration 6: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

Iteration 6: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	6.21	4.42	7.42	3.14	7.71

Iteration 7: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Iteration 7: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Iteration 7: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	6.21	3.14	7.42	7.71

Iteration 8: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	4.42	3.14	6.21	7.42	7.71

Iteration 8: step 1.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 8: step 2.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 8: step 3.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.
- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 9: step 0.

Insertion Sort

- Iteration i. Repeatedly swap element i with the one to its left if smaller.

- Property. After ith iteration, `a[0]` through `a[i]` contain first i+1 elements in ascending order.

Array index	0	1	2	3	4	5	6	7	8	9
Value	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Iteration 10: DONE.

Insertion sort – Pseudo code

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

```
n = length(A)

for i = 1 to n - 1
        j = i
        while j > 0 and A[j-1] > A[j]
                swap(A[j], A[j-1])
                j = j - 1
```

i = 1,

```
        j = 1:
        while j > 0 and A[0] > A[1]
                swap
                j = j - 1
```

Insertion sort – Pseudo code

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

```
n = length(A)

for i = 1 to n - 1
        j = i
        while j > 0 and A[j-1] > A[j]
                swap(A[j], A[j-1])
                j = j - 1
```

i = 2,

```
        j = 2:
        while j > 0 and A[1] > A[2]
                swap
                j = j - 1
```

Insertion sort – Pseudo code

Array index	0	1	2	3	4	5	6	7	8	9
Value	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

```
n = length(A)

for i = 1 to n - 1
        j = i
        while j > 0 and A[j-1] > A[j]
                swap(A[j], A[j-1])
                j = j - 1
```

i = 3,

```
        j = 3:
        while j > 0 and A[2] > A[3]
                swap
                j = j - 1
```

Insertion Sort

```
void insertion_sort(int arr[], int n)
{
        int i, temp, j;
        for (i = 1; i < n; i++)
        {
                temp = arr[i];
                j = i - 1;

                while (j >= 0 && arr[j] > temp)
                {
                        arr[j + 1] = arr[j];
                        j = j - 1;
                }
                arr[j + 1] = temp;
        }
}
```

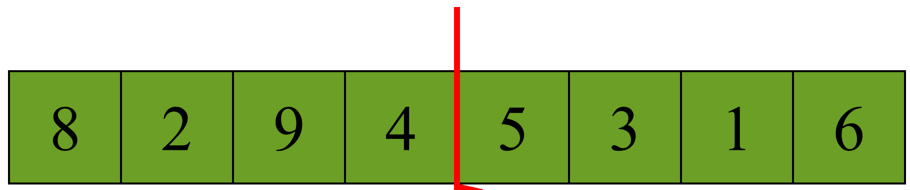
Sorting

- Insertion Sort
- Merge Sort
- Quick Sort

“Divide and Conquer”

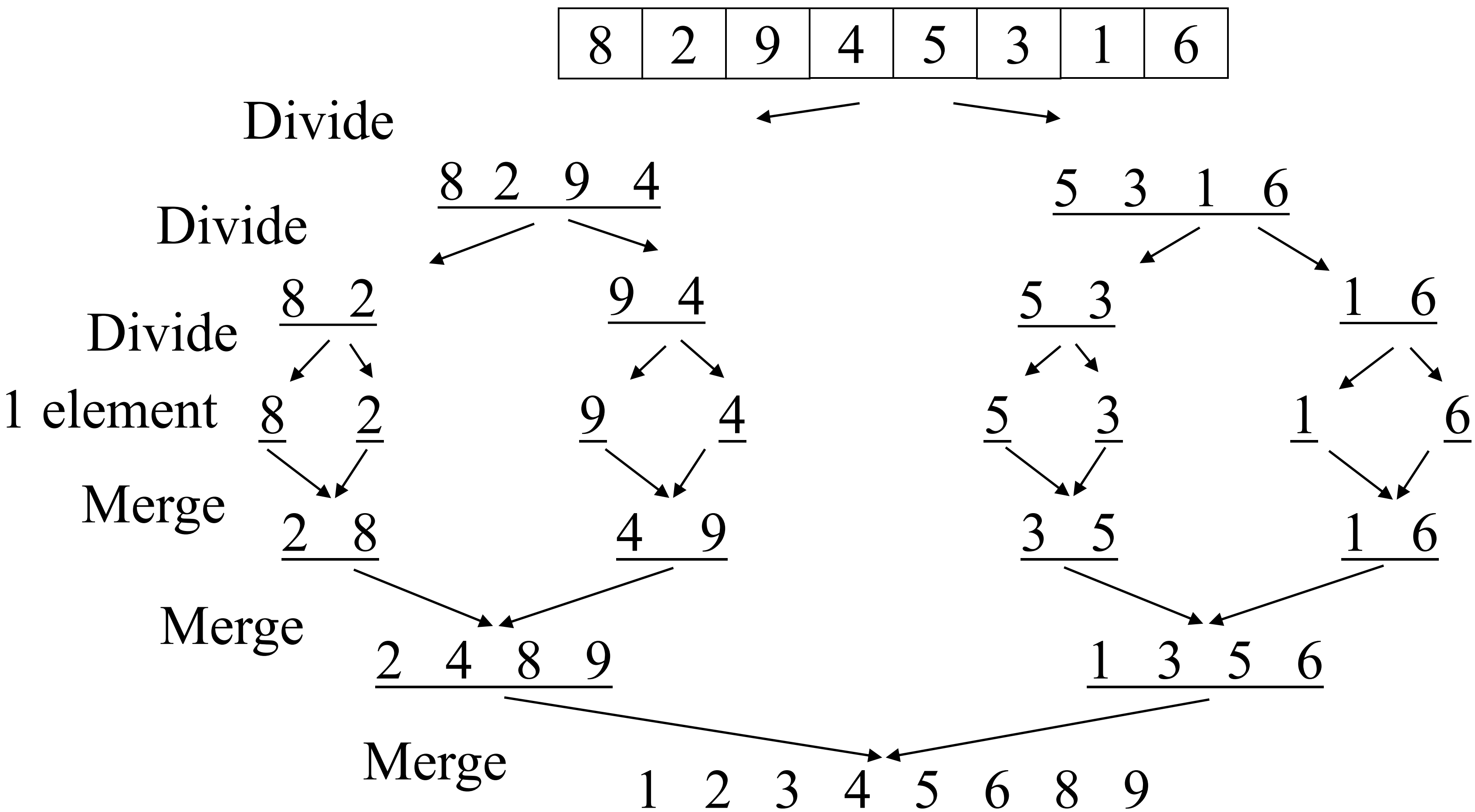
- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort
- **Idea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

Mergesort



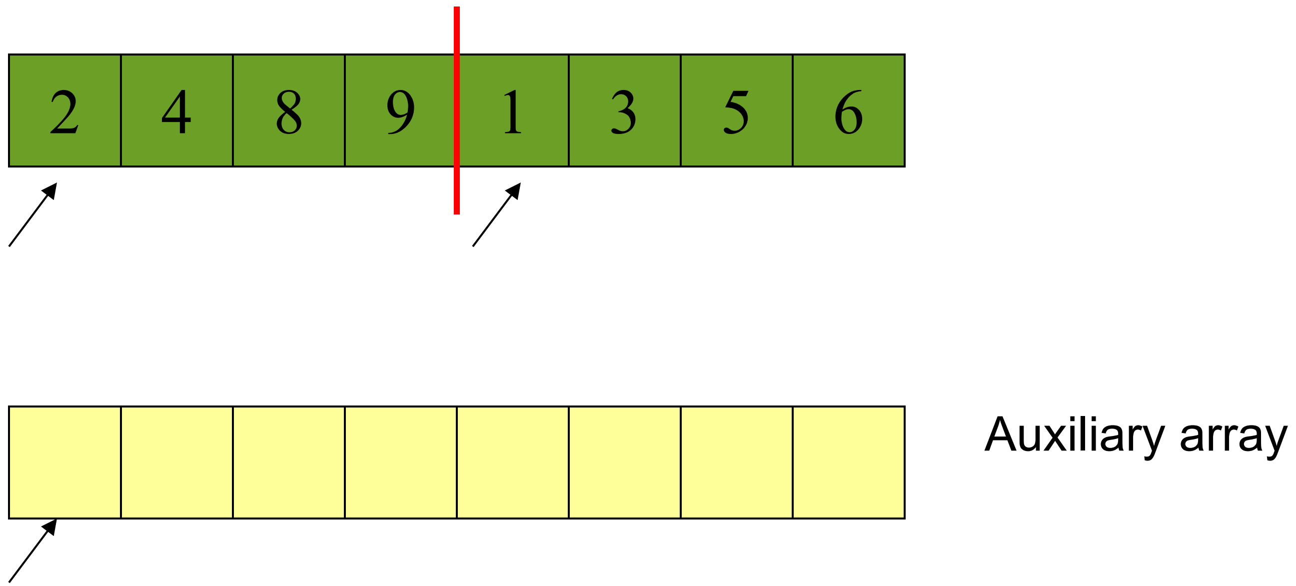
- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Mergesort Example



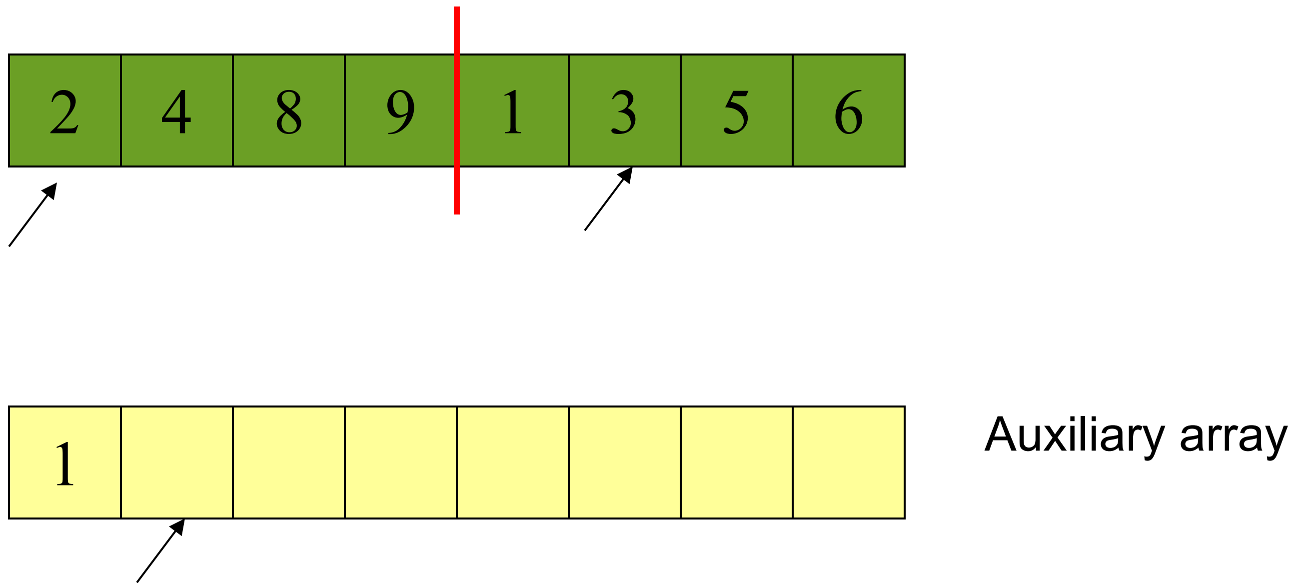
Auxiliary Array

- The merging requires an auxiliary array.



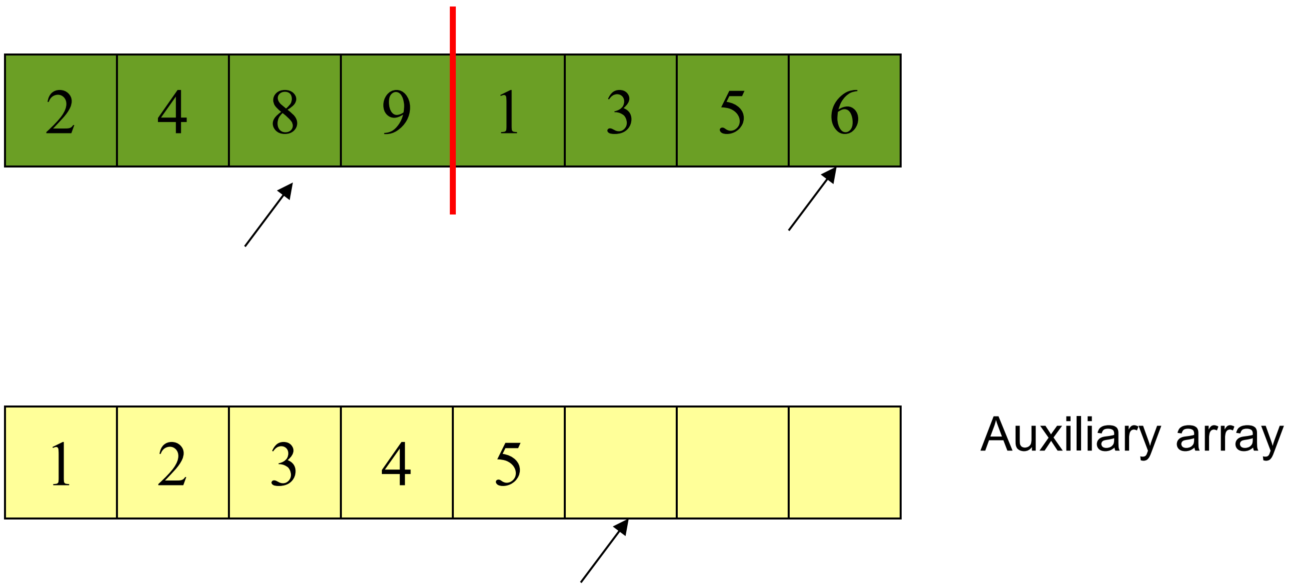
Auxiliary Array

- The merging requires an auxiliary array.

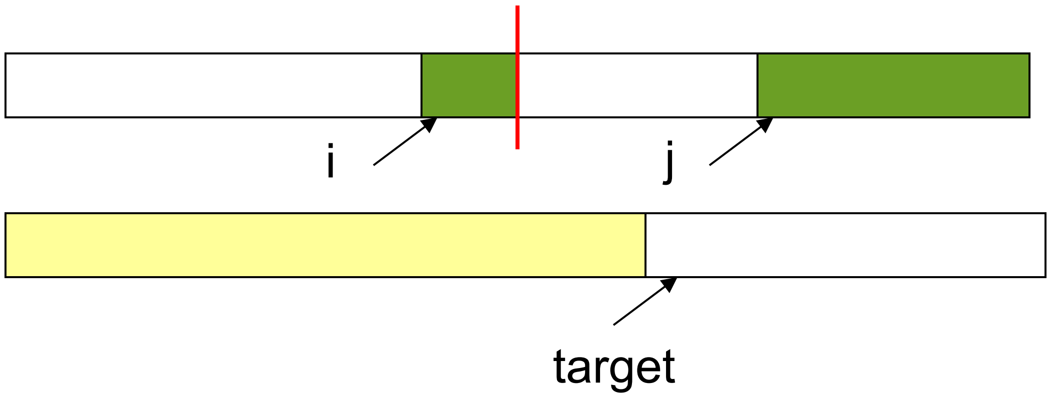


Auxiliary Array

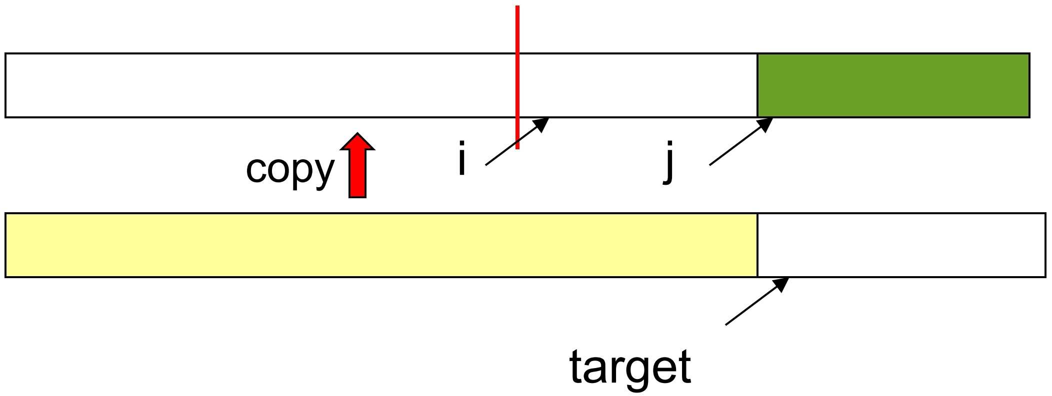
- The merging requires an auxiliary array.



Merging

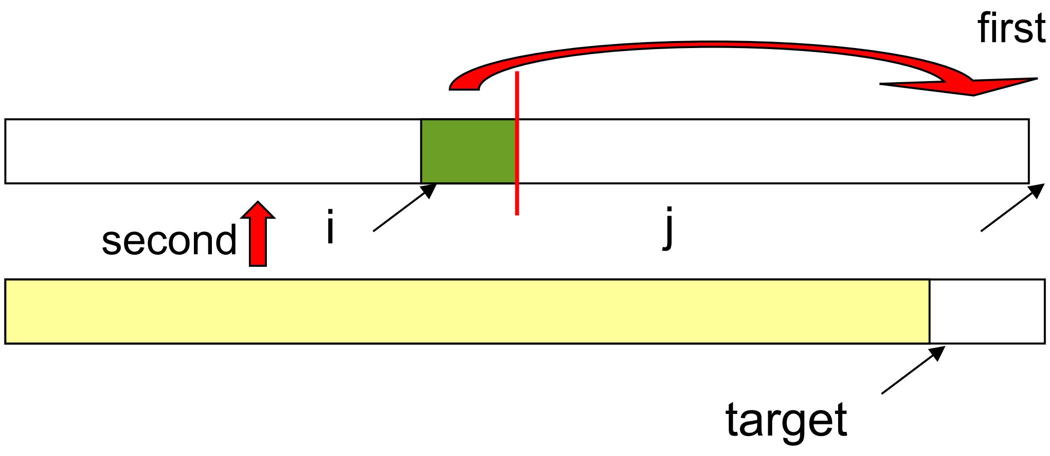


normal



Left completed first

Merging



Right completed first

Merging

MergeSort(arr[], l, r)

If r > l

- Find the middle point to divide the array into two halves:
  - middle m = l + (r – l)/2
- Call mergeSort for first half:
  - Call mergeSort(arr, l, m)
- Call mergeSort for second half:
  - Call mergeSort(arr, m + 1, r)
- Merge the two halves sorted in steps 2 and 3:
  - Call merge(arr, l, m, r)

Merging

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

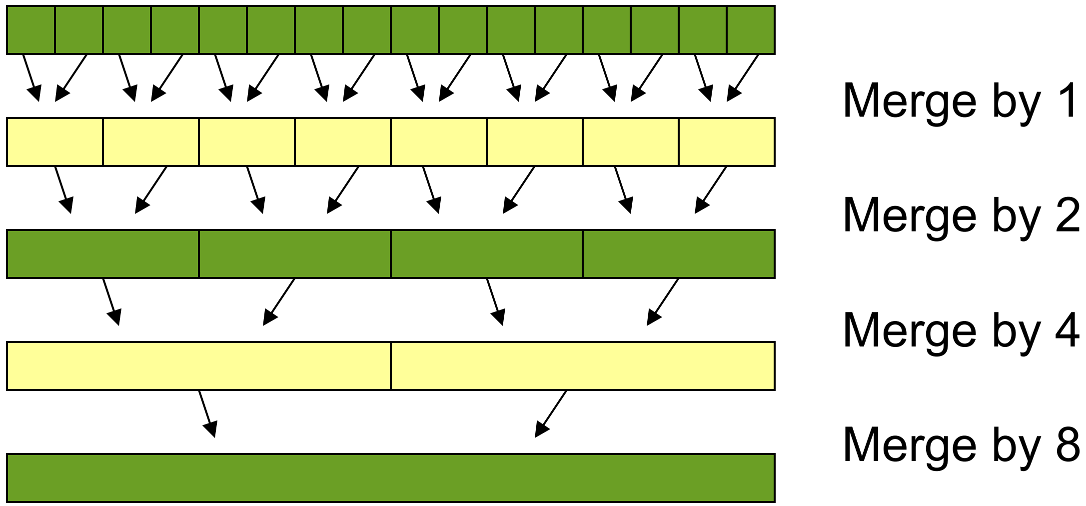
```
if left > right
    return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

step 4: Stop

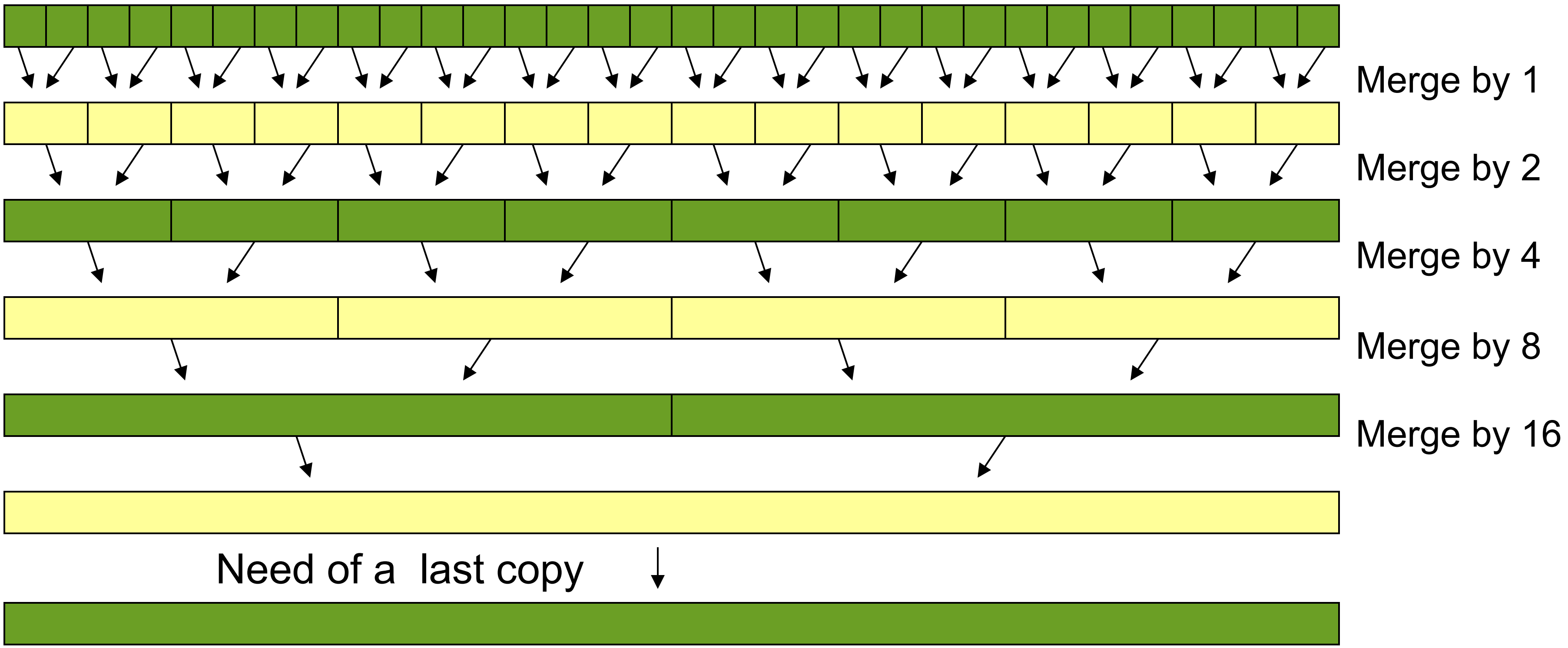
Merging

Code

Iterative Mergesort



Iterative Mergesort



Iterative Mergesort

Iterative Merge Sort Algorithm
Consider an array Arr[] of size N that we want to sort:

Step 1: Initialize sub_size with 1
multiply it by 2 as long as it is less than N.
And for each sub_size, do the following:

Step 2: Initialize L with 0 and add 2*sub_size as long as it is less than N.
Calculate Mid as min(L + sub_size - 1, N-1)
R as min(L + (2* sub_size) -1, N-1) and do the following:

Step 3: Copy sub-array [L, Mid-1] in list A and sub-array [Mid, R] in list B
merge these sorted lists to make a sorted list C using the following method:

Step 3.1: Compare the first elements of lists A and B
remove the first element from the list whose first element is smaller and append it to C.
Repeat this until either list A or B becomes empty.

Step 3.2: Copy the list(A or B), which is not empty, to C.

Step 4: Copy list C to Arr[] from index L to R.

Iterative Mergesort

```
IterativeMergesort(A[1..n]: integer array, n : integer) : {
//precondition: n is a power of 2//
  i, m, parity : integer;
  T[1..n]: integer array;
  m := 2; parity := 0;
  while m ≤ n do
    for i = 1 to n - m + 1 by m do
       if parity = 0 then Merge(A,T,i,i+m-1);
         else Merge(T,A,i,i+m-1);
    parity := 1 - parity;
    m := 2*m;
  if parity = 1 then
    for i = 1 to n do A[i] := T[i];
}
```

How do you handle non-powers of 2?
How can the final copy be avoided?

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
- $T(N) = O(n \log n)$

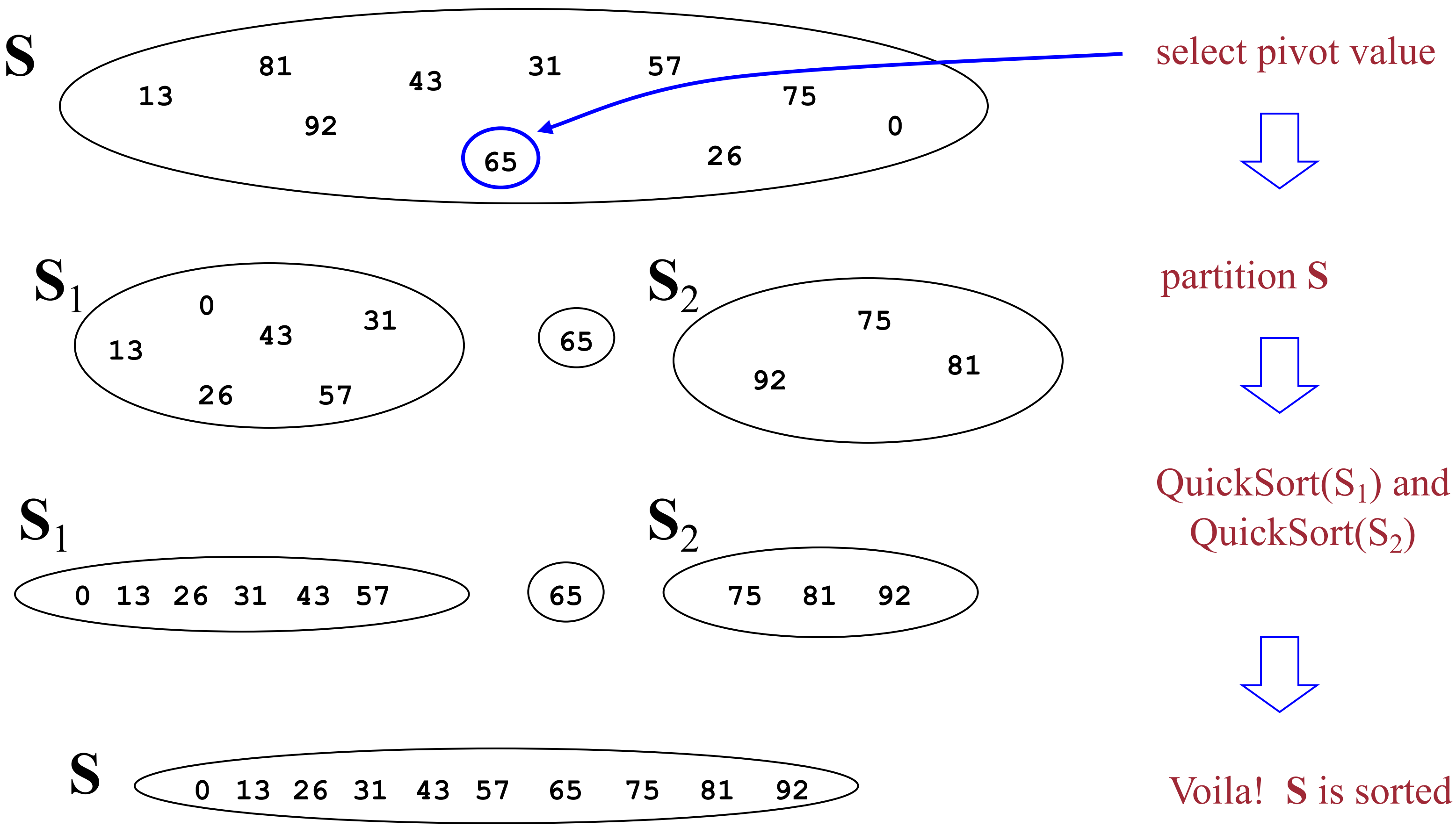
Properties of Mergesort

- Not in-place
  - Requires an auxiliary array (O(n) extra space)
- Stable
  - Make sure that left is sent to target on equal values.
- Iterative Mergesort reduces copying.

Sorting

- Insertion Sort
- Merge Sort
- Quick Sort

The steps of QuickSort



Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  - Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

"Four easy steps"

- To sort an array **S**
  1. If the number of elements in **S** is 0 or 1, then return. The array is sorted.
  2. Pick an element *v* in **S**. This is the *pivot* value.
  3. Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1$ = {all values $x \leq v$}, and $\mathbf{S}_2$ = {all values $x \geq v$}.
  4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

Details, details

- Implementing the actual partitioning
- Picking the pivot
  - want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Dealing with cases where the element equals the pivot