CSE 332: Data Abstractions
Sorting I

Spring 2016

Announcements

Sorting

- Input
  - an array A of data records
  - a key value in each data record
  - a comparison function which imposes a consistent ordering on the keys
- Output
  - "sorted" array A such that
    - For any i and j, if $i < j$ then $A[i] \leq A[j]$

Consistent Ordering

- The comparison function must provide a ***consistent ordering*** on the set of possible keys
  - You can compare any two keys and get back an indication of a < b, a > b, or a = b (trichotomy)
  - The comparison functions must be consistent
    - If `compare(a,b)` says a<b, then `compare(b,a)` must say b>a
    - If `compare(a,b)` says a=b, then `compare(b,a)` must say b=a
    - If `compare(a,b)` says a=b, then `equals(a,b) and equals(b,a)` must say a=b

Why Sort?

- Provides fast search:
- Find *k*th largest element in:

Space

- How much space does the sorting algorithm require?
  - In-place: no more than the array or at most O(1) addition space
  - out-of-place: use separate data structures, copy back
  - External memory sorting – data so large that does not fit in memory

Stability

A sorting algorithm is **stable** if:

- Items in the input with the same value end up in the same order as when they began.

Input		**Unstable sort**		**Stable Sort**	
Adams	1	Adams	1	Adams	1
Black	2	Smith	1	Smith	1
Brown	4	Washington	2	Black	2
Jackson	2	Jackson	2	Jackson	2
Jones	4	Black	2	Washington	2
Smith	1	White	3	White	3
Thompson	4	Wilson	3	Wilson	3
Washington	2	Thompson	4	Brown	4
White	3	Brown	4	Jones	4
Wilson	3	Jones	4	Thompson	4

Time

How fast is the algorithm?

- requirement: for any $i<j$, $A[i] \leq A[j]$
- This means that you need to at least check on each element at the very minimum
  - Complexity is at least:
- And you could end up checking each element against every other element
  - Complexity could be as bad as:

The big question: How close to $O(n)$ can you get?

Sorting: *The Big Picture*

Simple algorithms: O(n^2)

Insertion sort
Selection sort
…

Fancier algorithms: O($n \log n$)

Heap sort
Merge sort
Quick sort (avg)
…

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: O(n)

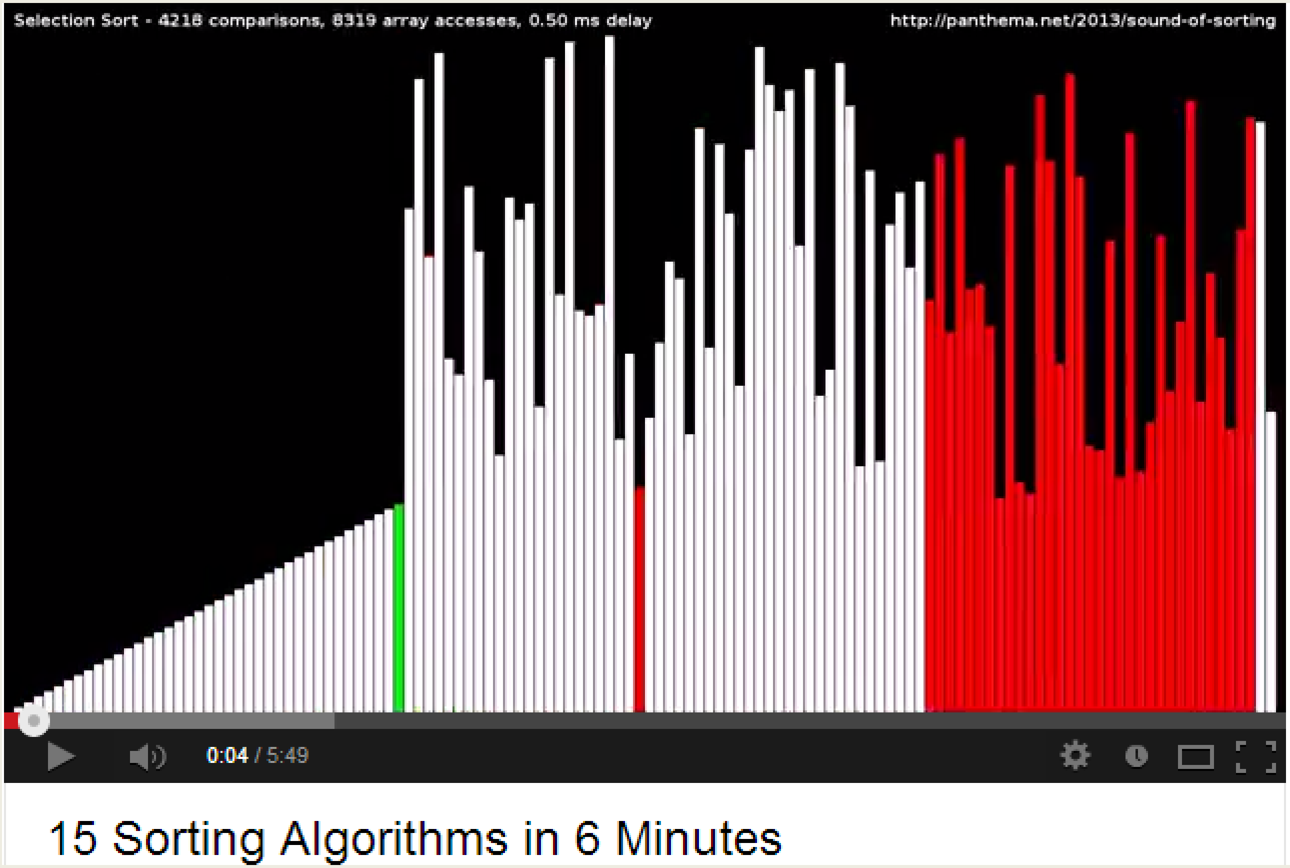
Bucket sort
Radix sort

Handling huge data sets

External sorting

Demo (with sound!)

- http://www.youtube.com/watch?v=kPRA0W1kECg



15 Sorting Algorithms in 6 Minutes

Selection Sort: idea

1. Find the smallest element, put it 1st
2. Find the next smallest element, put it 2nd
3. Find the next smallest, put it 3rd
4. And so on …

Try it out: Selection Sort

- 31, 16, 54, 4, 2, 17, 6

Selection Sort: Code

```
void SelectionSort (Array a[0..n-1]) {
    for (i=0; i<n; ++i) {
        j = Find index of
                smallest entry in a[i..n-1]
        Swap(a[i],a[j])
    }
}
```

Runtime:

worst case :

best case :

average case :

Bubble Sort

- Take a pass through the array
  - If neighboring elements are out of order, swap them.
- Repeat until no swaps needed

- Wost & avg case: $O(n^2)$
  - pretty much no reason to ever use this algorithm

Insertion Sort

1. Sort first 2 elements.
2. Insert 3rd element in order.
    - (First 3 elements are now sorted.)
3. Insert 4th element in order
    - (First 4 elements are now sorted.)
4. And so on…

How to do the insertion?

Suppose my sequence is:

16, 31, 54, 78, 32, 17, 6

And I’ve already sorted up to 78. How to insert 32?

Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6

Insertion Sort: Code

```
void InsertionSort (Array a[0..n-1]) {
    for (i=1; i<n; i++) {
        for (j=i; j>0; j--) {
            if (a[j] < a[j-1])
                Swap(a[j],a[j-1])
            else
                break
      }

}
```

Note: can instead move the “hole” to minimize copying, as with a binary heap.

Runtime:

worst case :

best case :

average case :

Insertion Sort vs. Selection Sort

- Same worst case, avg case complexity
- Insertion better best-case
  - preferable when input is “almost sorted”
    - one of the best sorting algs for almost sorted case (also for small arrays)

Sorting: *The Big Picture*

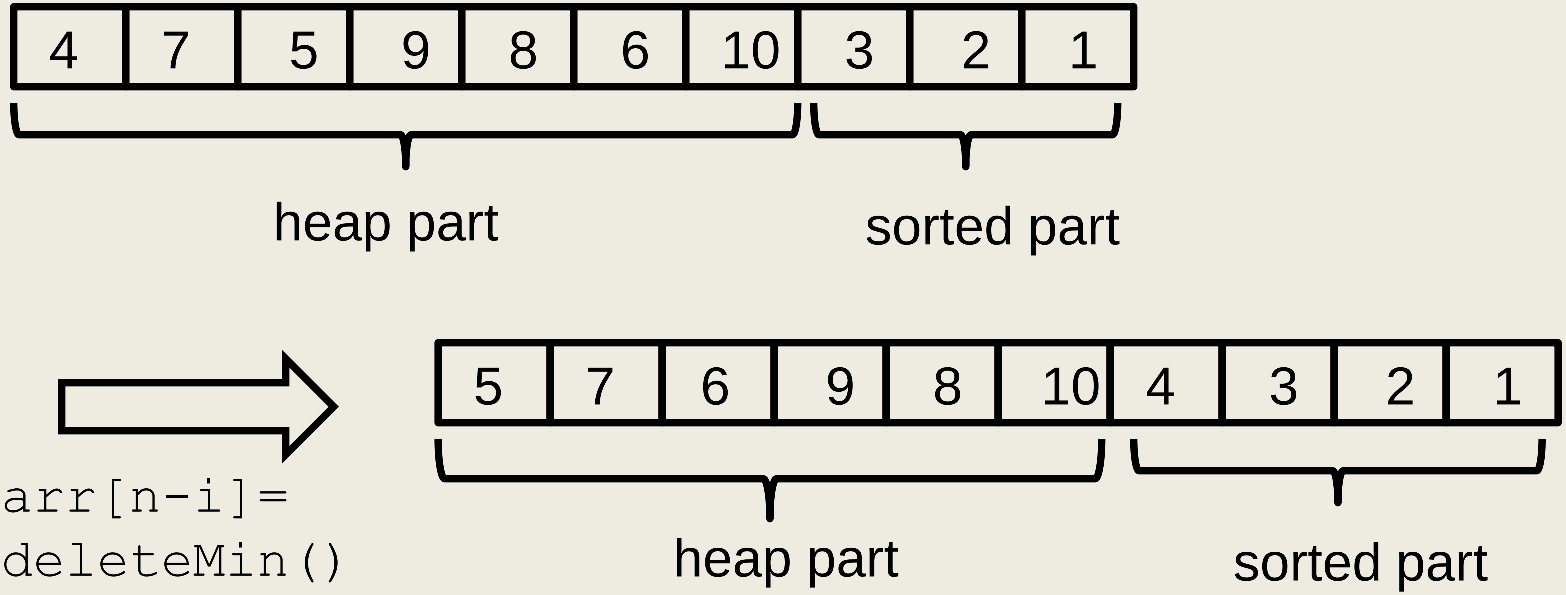
- Simple algorithms: $O(n^2)$
  - Insertion sort
  - Selection sort
  - …
- Fancier algorithms: $O(n \log n)$
  - Heap sort
  - Merge sort
  - Quick sort (avg)
  - …
- Comparison lower bound: $\Omega(n \log n)$
- Specialized algorithms: $O(n)$
  - Bucket sort
  - Radix sort
- Handling huge data sets
  - External sorting

Heap Sort: Sort with a Binary Heap

Worst Case Runtime:

In-place heap sort

- Treat the initial array as a heap (via **buildHeap**)
- When you delete the **i**th element, put it at **arr[n-i]**
  - It’s not part of the heap anymore!



4	7	5	9	8	6	10	3	2	1

heap part

sorted part

```
arr[n-i]=
deleteMin()
```

5	7	6	9	8	10	4	3	2	1

heap part

sorted part

AVL Sort

Insert nodes into an AVL Tree
Conduct an In-order traversal to extract nodes in sorted order

Worst Case Runtime:

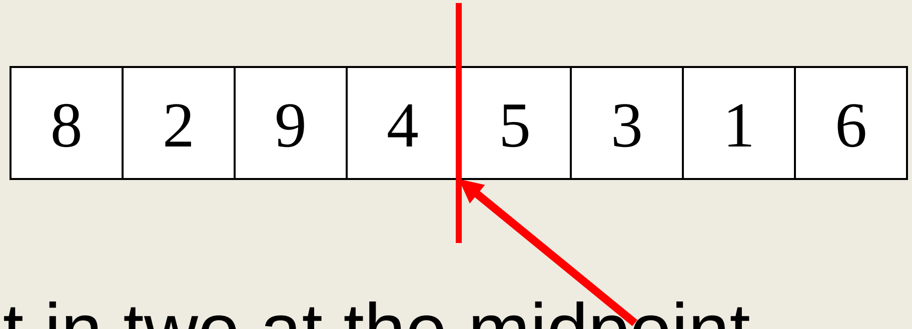
“Divide and Conquer”

- 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 in half, *recursively* sort left and right halves, then *merge* two halves
  → known as Mergesort

- **Idea 2 :** Partition array into small items and large items, then recursively sort the two sets
  → known as Quicksort

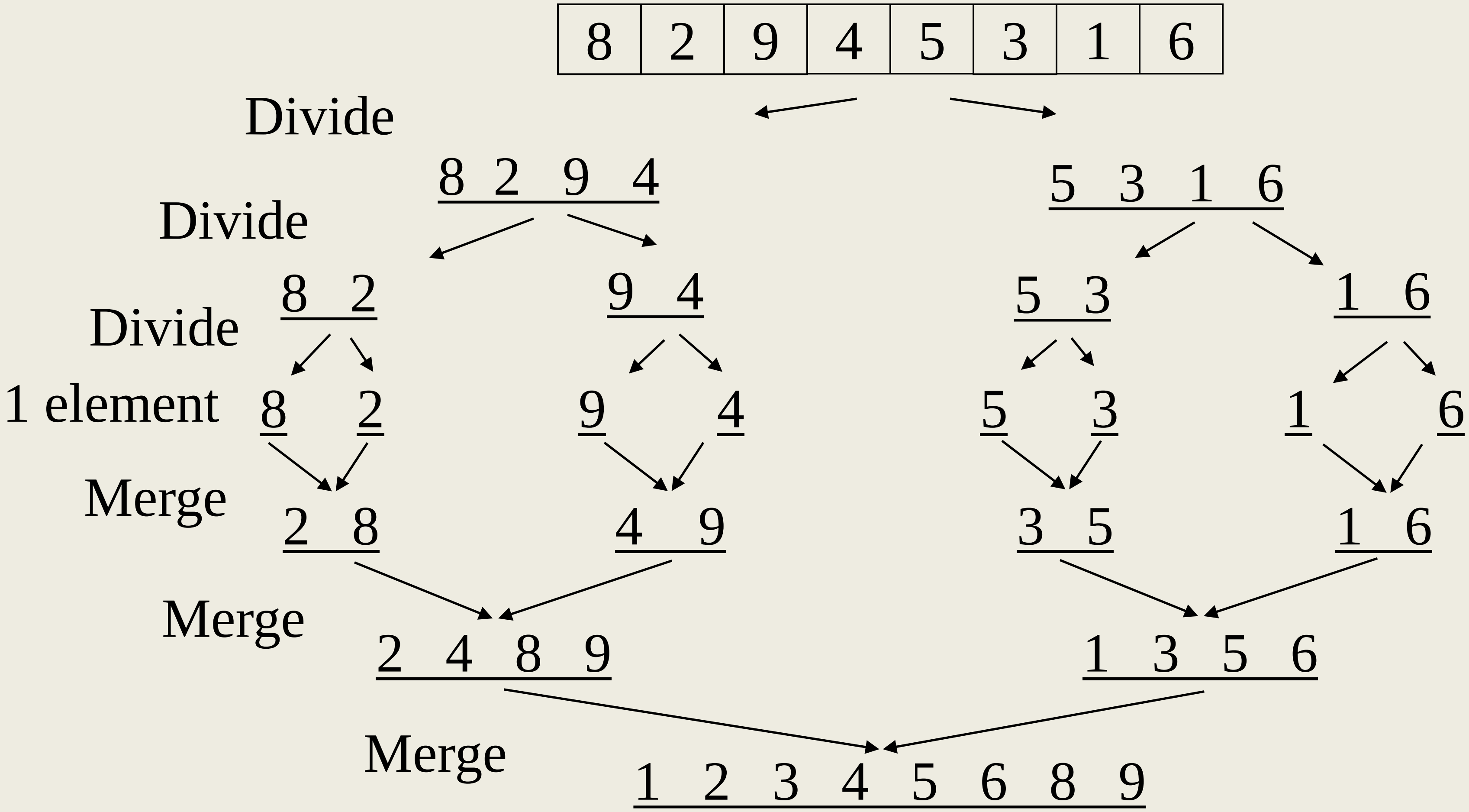
Mergesort



8	2	9	4	5	3	1	6

- Divide it in two at the midpoint
- Sort each half (recursively)
- Merge two halves together

Mergesort Example



8	2	9	4	5	3	1	6

Divide

8 2 9 4

5 3 1 6

Divide

8 2

9 4

5 3

1 6

Divide

1 element

8

2

9

4

5

3

1

6

Merge

2 8

4 9

3 5

1 6

Merge

2 4 8 9

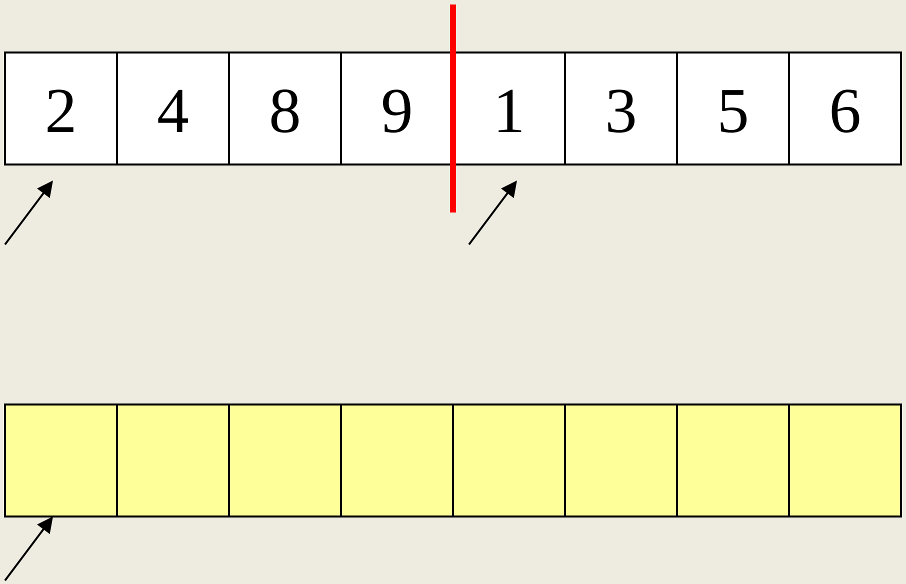
1 3 5 6

Merge

1 2 3 4 5 6 8 9

Merging: Two Pointer Method

- Perform merge using an auxiliary array

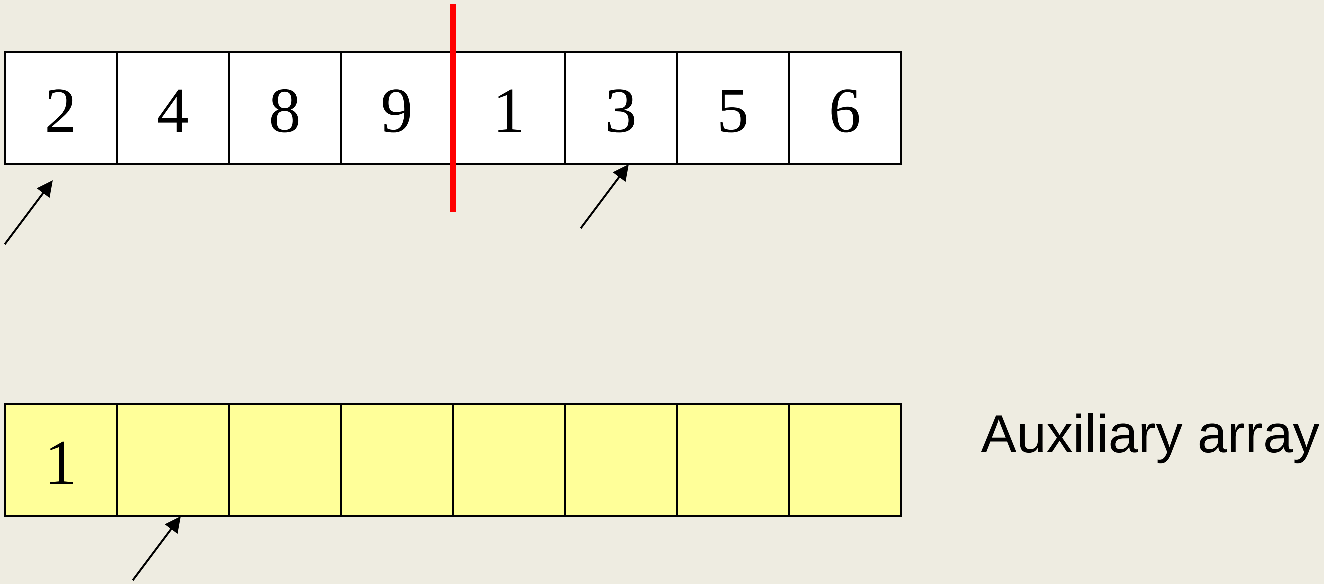


2	4	8	9	1	3	5	6

Auxiliary array

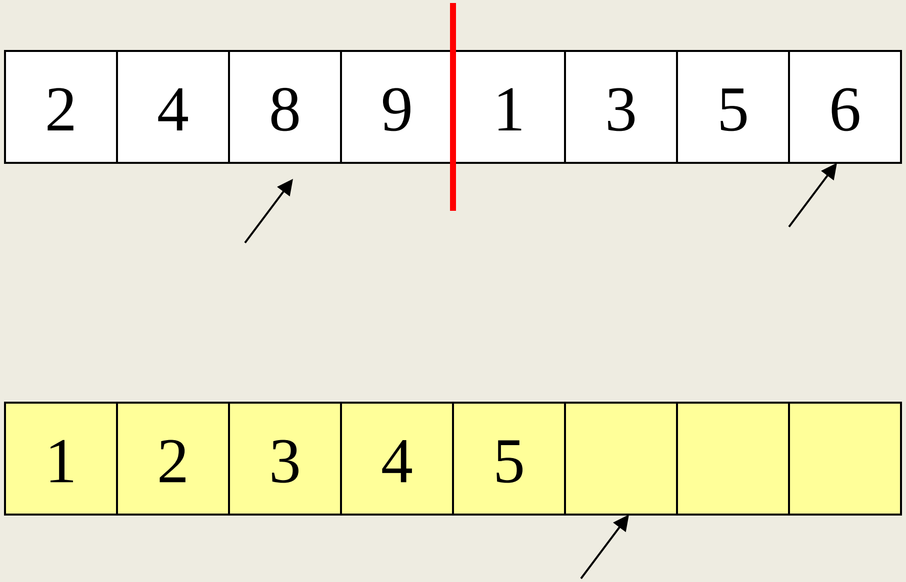
Merging: Two Pointer Method

- Perform merge using an auxiliary array



Merging: Two Pointer Method

- Perform merge using an auxiliary array



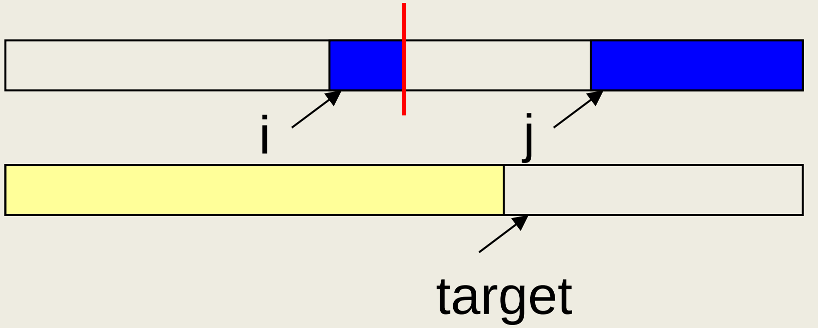
2	4	8	9	1	3	5	6

1	2	3	4	5			

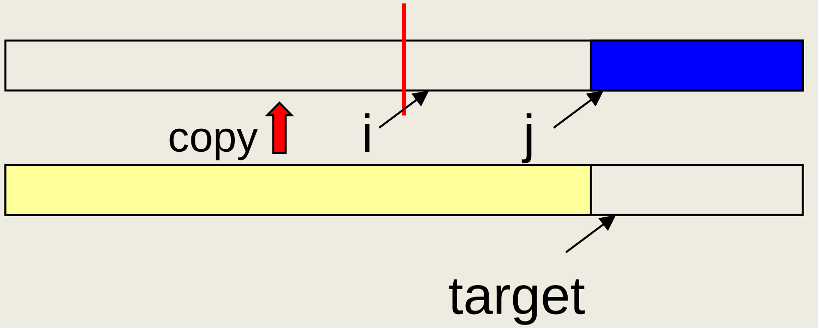
Auxiliary array

Merging: Finishing Up

Starting from here…

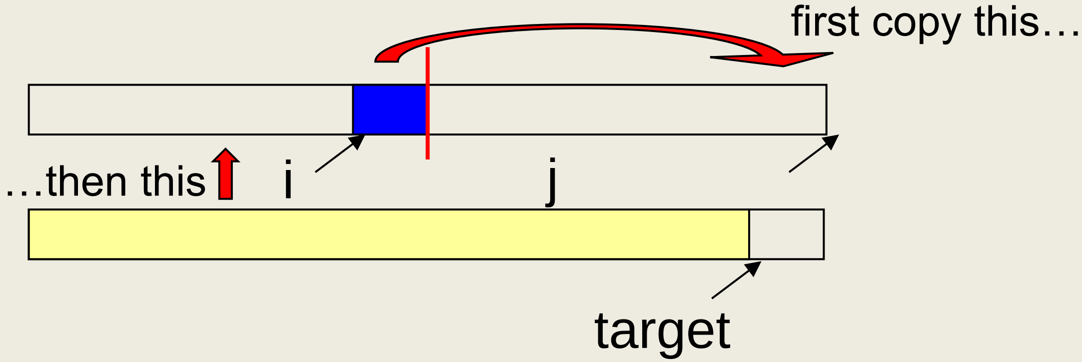


Left finishes up



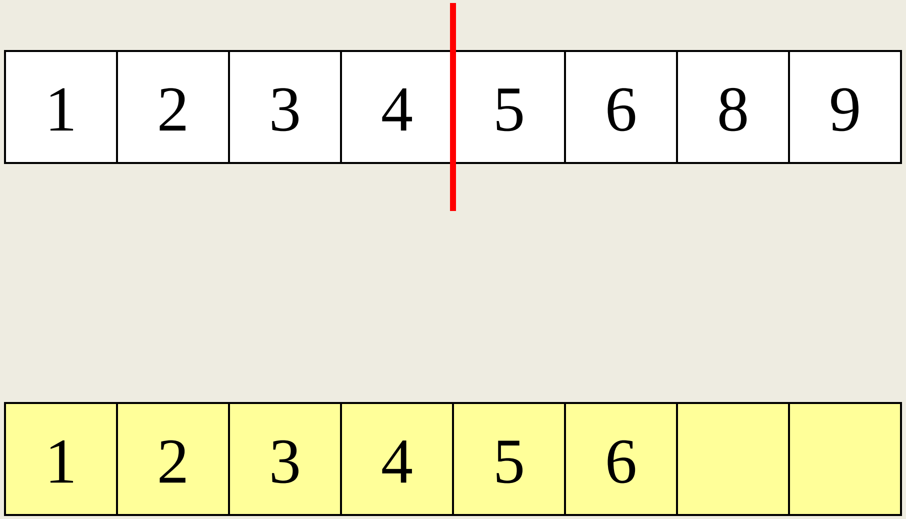
or

Right finishes up



Merging: Two Pointer Method

- Final result



1	2	3	4	5	6	8	9

1	2	3	4	5	6		

Auxiliary array

Complexity?

Stability?

Merging

```
Merge(A[], Temp[], left, mid, right)  {
  Int i, j, k, l, target
  i = left
  j = mid + 1
  target = left
  while (i ≤ mid && j ≤ right) {
    if (A[i] ≤ A[j])
      Temp[target] = A[i++]
    else
      Temp[target] = A[j++]
    target++
  }
  if (i > mid) //left completed//
    for (k = left to target-1)
      A[k] = Temp[k];
  if (j > right) //right completed//
    k = mid
    l = right
    while (k ≥ i)
      A[l--] = A[k--]
    for (k = left to target-1)
      A[k] = Temp[k]
}
```

Recursive Mergesort

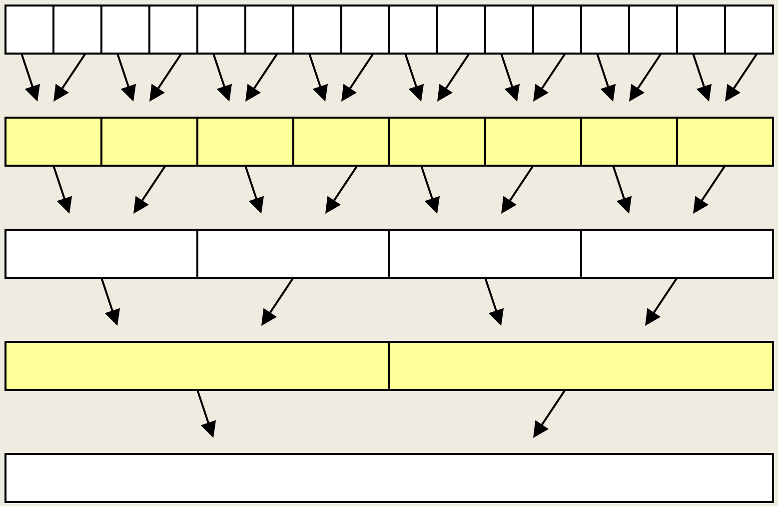
```
MainMergesort(A[1..n], n) {
  Array Temp[1..n]
  Mergesort[A, Temp, 1, n]
}

Mergesort(A[], Temp[], left, right) {
  if (left < right) {
    mid = (left + right)/2
    Mergesort(A, Temp, left, mid)
    Mergesort(A, Temp, mid+1, right)
    Merge(A, Temp, left, mid, right)
  }
}
```

What is the recurrence relation?

Mergesort: Complexity

Iterative Mergesort



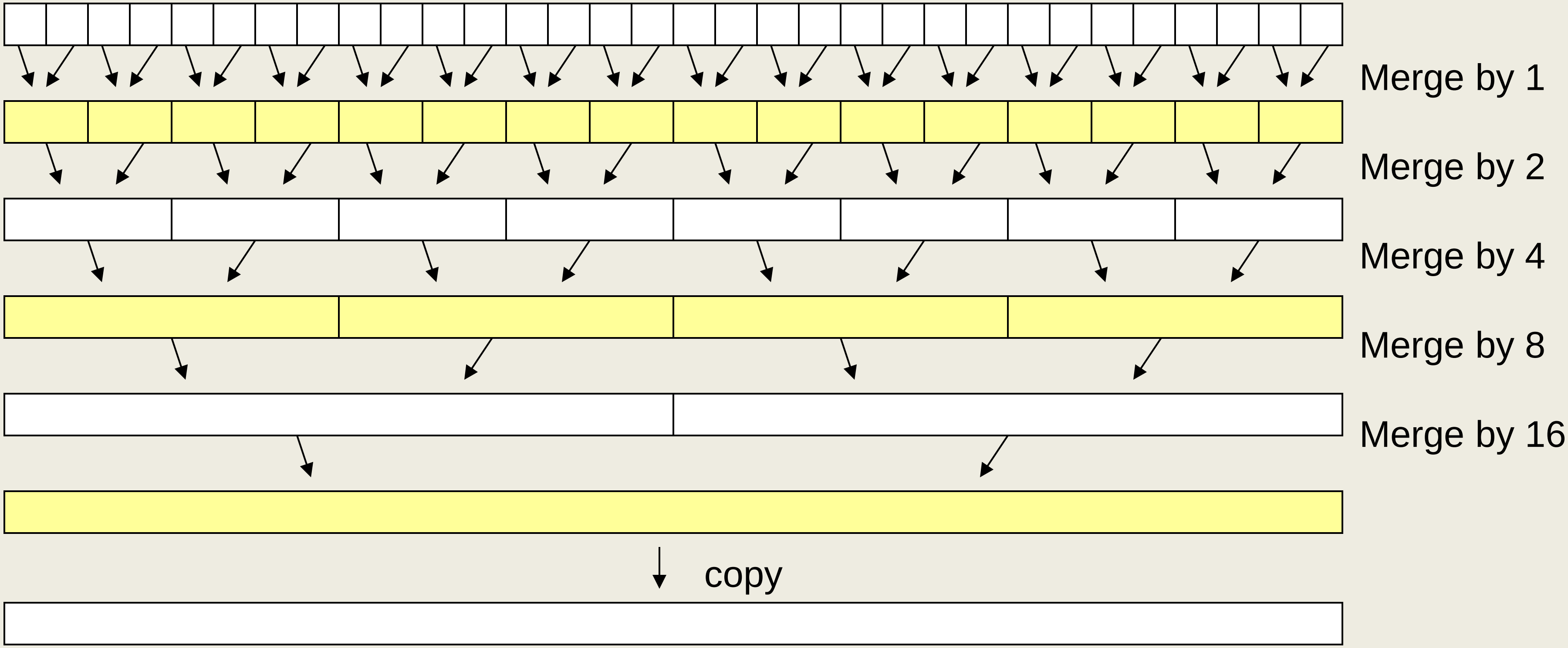
Merge by 1

Merge by 2

Merge by 4

Merge by 8

Iterative Mergesort



Iterative Mergesort reduces copying

Complexity?

Properties of Mergesort

- In-place?
- Stable?
- Sorted list complexity?
- Nicely extends to handle linked lists.
- Multi-way merge is basis of big data sorting.
- Java uses Mergesort on Collections and on Arrays of Objects.