Chapter 7 Sorting - part 1

South China University of Technology

College of Software Engineering

Huang Min

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 (e.g., integers)
  - › Measures of cost: Comparisons & Swaps

- Output
  - › reorganize the elements of A such that
    - For any i and j, if $i < j$ then $A[i] \leq A[j]$

Space

- How much space does the sorting algorithm require in order to sort the collection of items?
  - Is copying needed? O(n) additional space
  - In-place sorting – no copying – O(1) additional space
  - Somewhere in between for “temporary”, e.g. O(logn) space
  - External memory sorting – data so large that does not fit in memory

Time

- How fast is the algorithm?
  - The definition of a sorted array A says that for any i<j, A[i] < A[j]
  - This means that you need to at least check on each element at the very minimum, I.e., at least O(N)
  - And you could end up checking each element against every other element, which is $O(N^2)$
  - The big question is: How close to O(N) can you get?

Stability

- Stability: Does it rearrange the order of input data records which have the same key value (duplicates)?
  - E.g. Phone book sorted by name. Now sort by county – is the list still sorted by name within each county?
  - Extremely important property for databases
  - A stable sorting algorithm is one which does not rearrange the order of duplicate keys

Three Θ (n^2) Sorting Algorithms

Three $\Theta(n^2)$ Sorting Algorithms

Three simple sorting algorithms. While easy to understand and implement, they are unacceptably slow when there are many records to sort.

Nonetheless, there are situations where one of these simple algorithms is the best tool for the job.

Insertion Sort

	i=1	2	3	4	5	6	7
42	20	17	13	13	13	13	13
20	42	20	17	17	14	14	14
17	17	42	20	20	17	17	15
13	13	13	42	28	20	20	17
28	28	28	28	42	28	23	20
14	14	14	14	14	42	28	23
23	23	23	23	23	23	42	28
15	15	15	15	15	15	15	42

Insertion Sort

```
template <typename E, typename Comp>
void inssort(E A[], int n) { // Insertion Sort
  for (int i=1; i<n; i++) // Insert i'th record
    for (int j=i; (j>0) && (Comp::prior(A[j],
         A[j-1])); j--)
          swap(A, j, j-1);
}
```

Best Case: 0 swaps, n - 1 comparisons

Worst Case: $n(n-1)/2$ swaps and comparisons

Average Case: $n(n-1)/4$ swaps and comparisons

Bubble Sort

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	42	14	14	14	14	14	14
17	20	42	15	15	15	15	15
13	17	20	42	17	17	17	17
28	14	17	20	42	20	20	20
14	28	15	17	20	42	23	23
23	15	28	23	23	23	42	28
15	23	23	28	28	28	28	42

Bubble Sort

```
template <typename E, typename Comp>
void bubsort(E A[], int n) { // Bubble Sort
  for (int i=0; i<n-1; i++) // Bubble up i'th record
    for (int j=n-1; j>i; j--)
      if (Comp::prior(A[j], A[j-1]))
        swap(A, j, j-1);
}
```

Best Case: 0 swaps, n(n-1)/2 comparisons

Worst Case:

n(n-1)/2 swaps and comparisons

Average Case:

n(n-1) /4 swaps and n(n-1) /2 comparisons.

Selection Sort

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

Selection Sort

```
template <typename E, typename Comp>
void selsort(E A[], int n) { // Selection Sort
  for (int i=0; i<n-1; i++) { // Select i'th record
    int lowindex = i; // Remember its index
    for (int j=n-1; j>i; j--) // Find the least value
      if (Comp::prior(A[j], A[lowindex]))
        lowindex = j; // Put it in place
          swap(A, i, lowindex);
  }
}
```

Best Case: 0 swaps (n-1 as written), *n(n-1)*/2 comparisons.

Worst Case: *n* - 1 swaps and *n(n-1)*/2 comparisons

Average Case: O(n) swaps and *n(n-1)*/2 comparisons

Summary

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

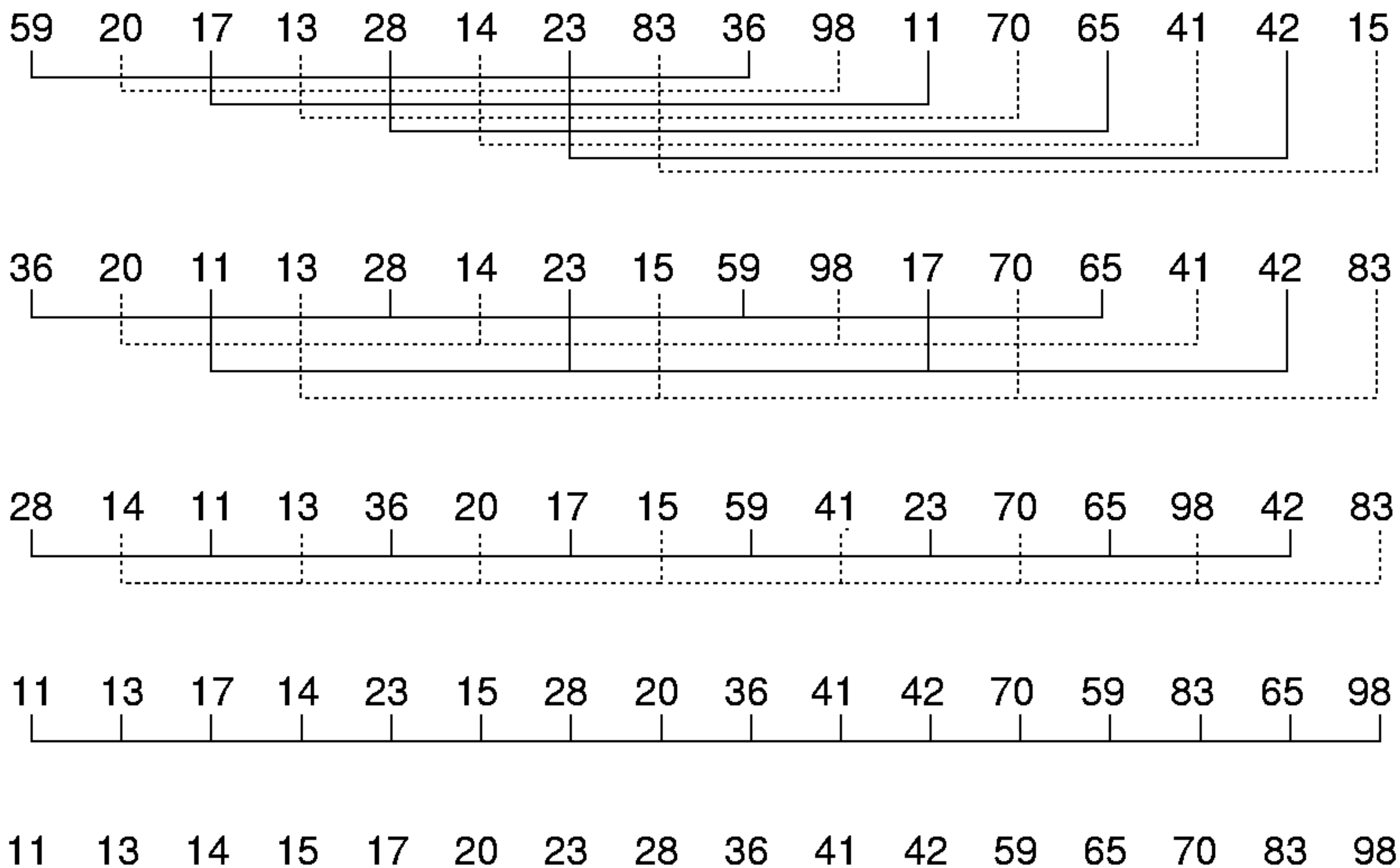
How about the stabilities of these algorithms?

Insert sort and bubble sort are stable.

Selection sort is unstable.

Shellsort

Shellsort


59 20 17 13 28 14 23 83 36 98 11 70 65 41 42 15
36 20 11 13 28 14 23 15 59 98 17 70 65 41 42 83
28 14 11 13 36 20 17 15 59 41 23 70 65 98 42 83
11 13 17 14 23 15 28 20 36 41 42 70 59 83 65 98
11 13 14 15 17 20 23 28 36 41 42 59 65 70 83 98

Shellsort

```
// Modified version of Insertion Sort for varying increments
template <typename E, typename Comp>
void inssort2(E A[], int n, int incr) {
  for (int i=incr; i<n; i+=incr)
    for (int j=i; (j>=incr) &&
        (Comp::prior(A[j], A[j-incr])); j-=incr)
          swap(A, j, j-incr);
}
template <typename E, typename Comp>
void shellsort(E A[], int n) { // Shellsort
  for (int i=n/2; i>2; i/=2) // For each increment
    for (int j=0; j<i; j++) // Sort each sublist
      inssort2<E,Comp>(&A[j], n-j, i);
      inssort2<E,Comp>(A, n, 1);
}
```

Shellsort

Shelsort cost:

$O(n^{1.25}) \sim O(1.6\, n^{1.25})$ -- Empirical formula

经验公式一般由拟合得到，没有完整的理论推导过程。 经验公式更趋向于应用，重要看其是否精确。

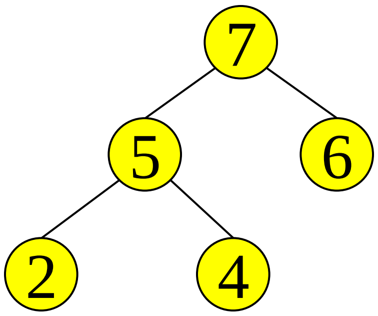
Shellsort algorithm is unstable.

Heap Sort

Heap Sort

- We use a Max-Heap
- Root node = A[1]
- Children of A[i] = A[2i], A[2i+1]
- Keep track of current size N (number of nodes)

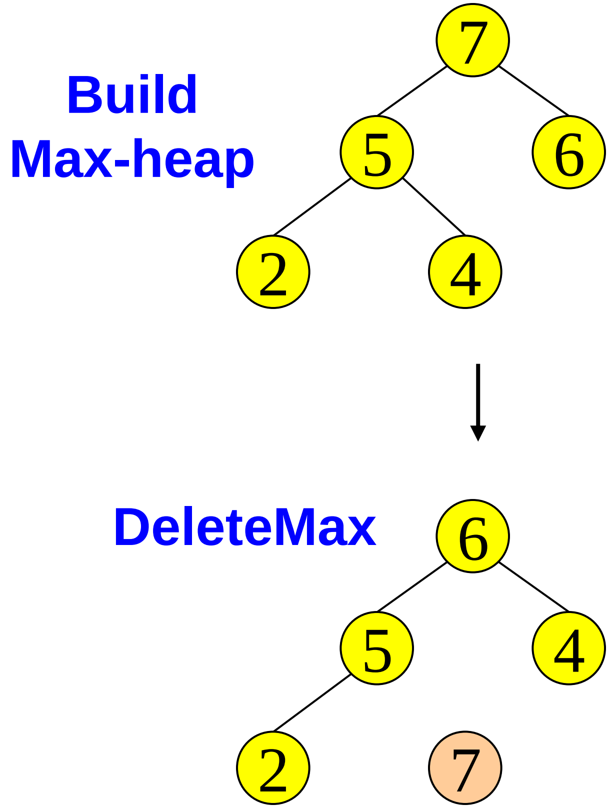
value	7	5	6	2	4			
index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**



N = 5

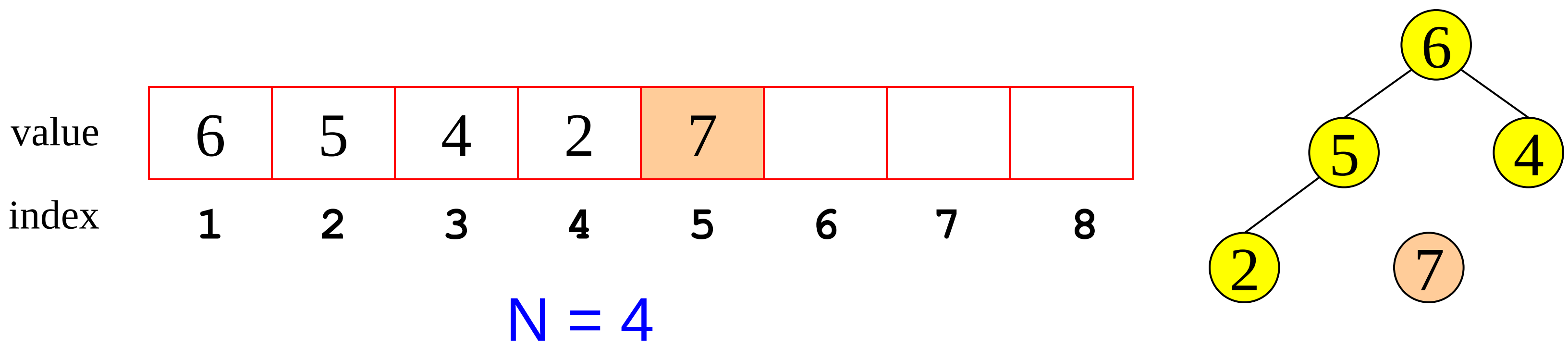
Using Binary Heaps for Sorting

- Build a max-heap
- Do N DeleteMax operations and store each Max element as it comes out of the heap
- Data comes out in largest to smallest order
- Where can we put the elements as they are removed from the heap?



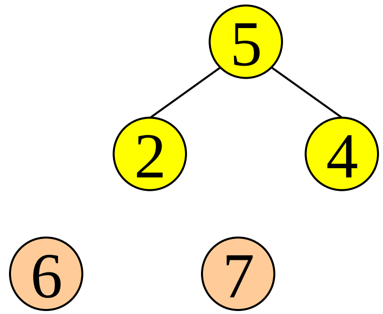
1 Removal = 1 Addition

- Every time we do a DeleteMax, the heap gets smaller by one node, and we have one more node to store
  - Store the data at the end of the heap array
  - Not "in the heap" but it is in the heap array



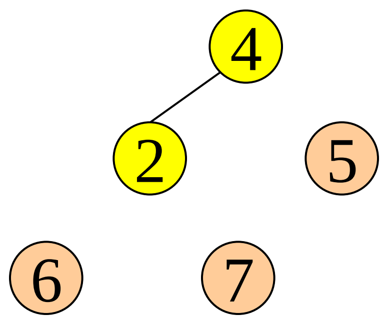
Repeated DeleteMax

5	2	4	6	7			
1	2	3	4	5	6	7	8



N = 3

4	2	5	6	7			
1	2	3	4	5	6	7	8



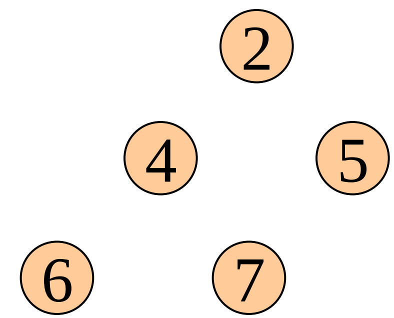
N = 2

Heap Sort is In-place

- After all the DeleteMaxs, the heap is gone but the array is full and is in sorted order

value	2	4	5	6	7			
index	1	2	3	4	5	6	7	8

N = 0



Heapsort: Analysis

- Running time
  - › time to build max-heap is $O(N)$
  - › time for N DeleteMax operations is N $O(\log N)$
  - › total time is **$O(N \log N)$**
- Can also show that running time is $\Omega(N \log N)$ for some inputs,
  - › so *worst case* is **$\Theta(N \log N)$**
  - › *Average case* running time is also $O(N \log N)$
- Heapsort is in-place but not stable (why?)

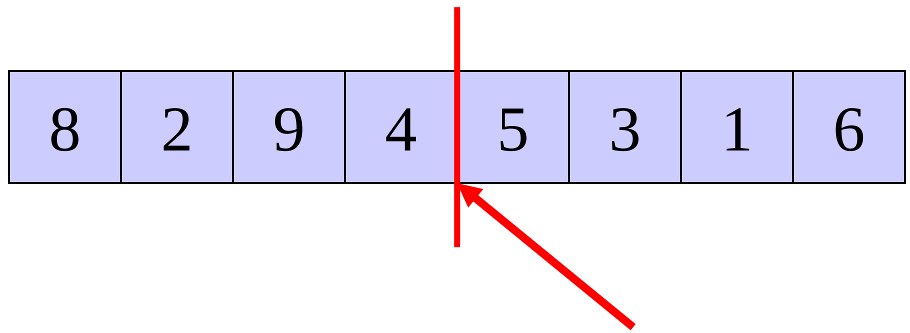
Heapsort algorithm is unstable.

“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

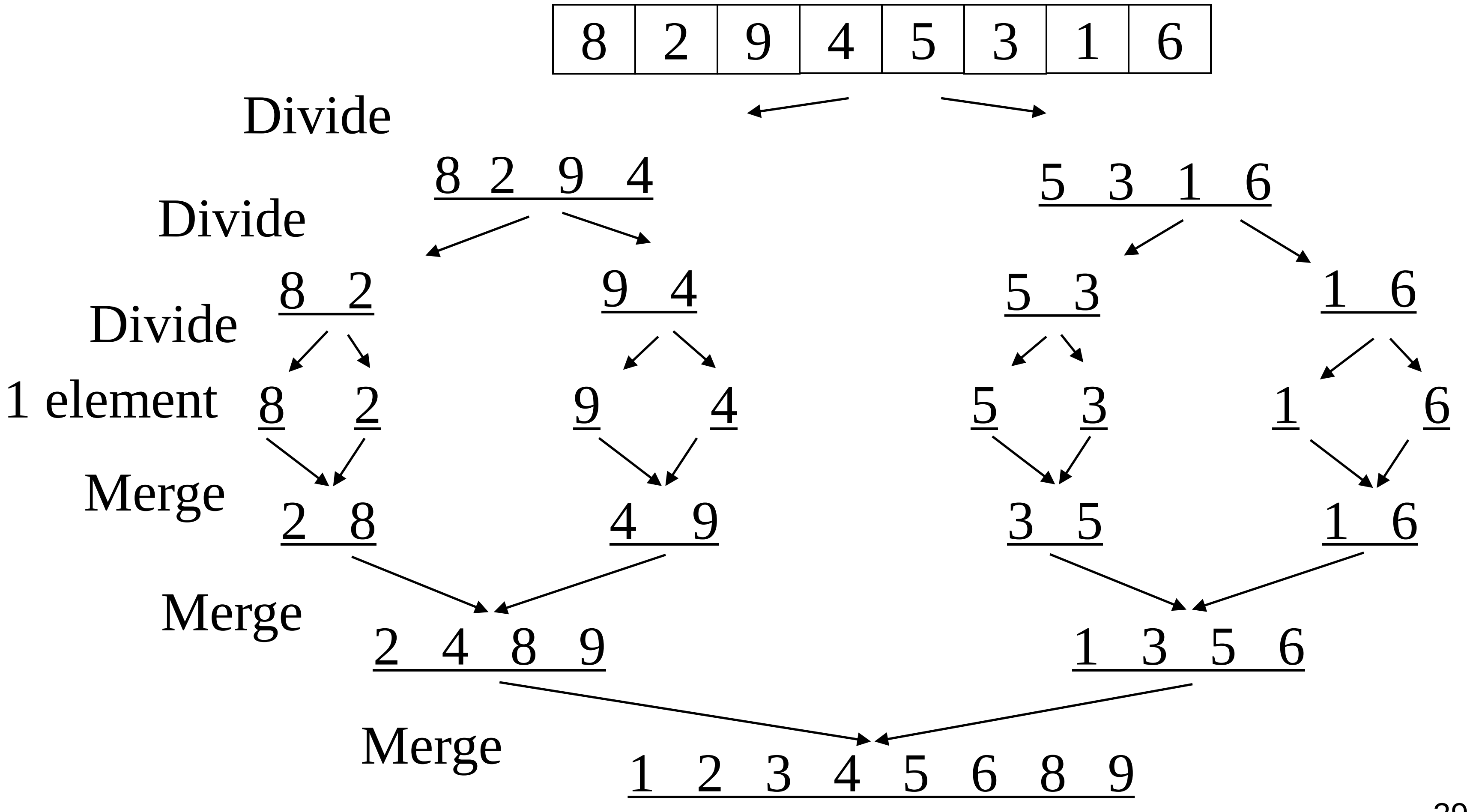
Mergesort

Mergesort



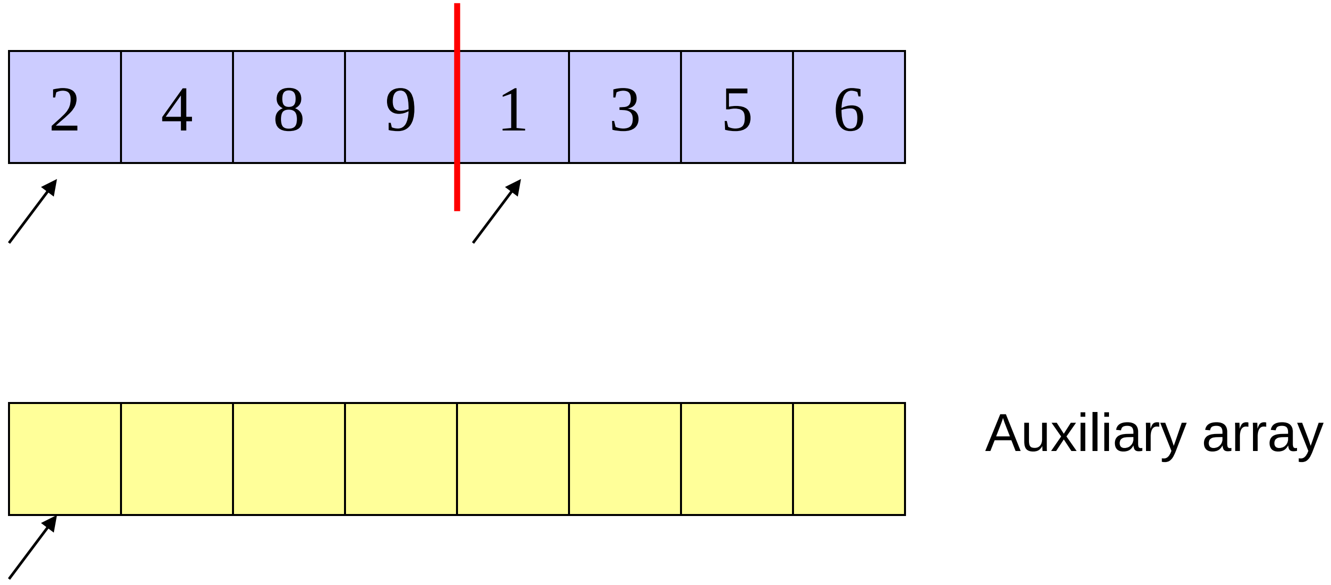
- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Mergesort Example



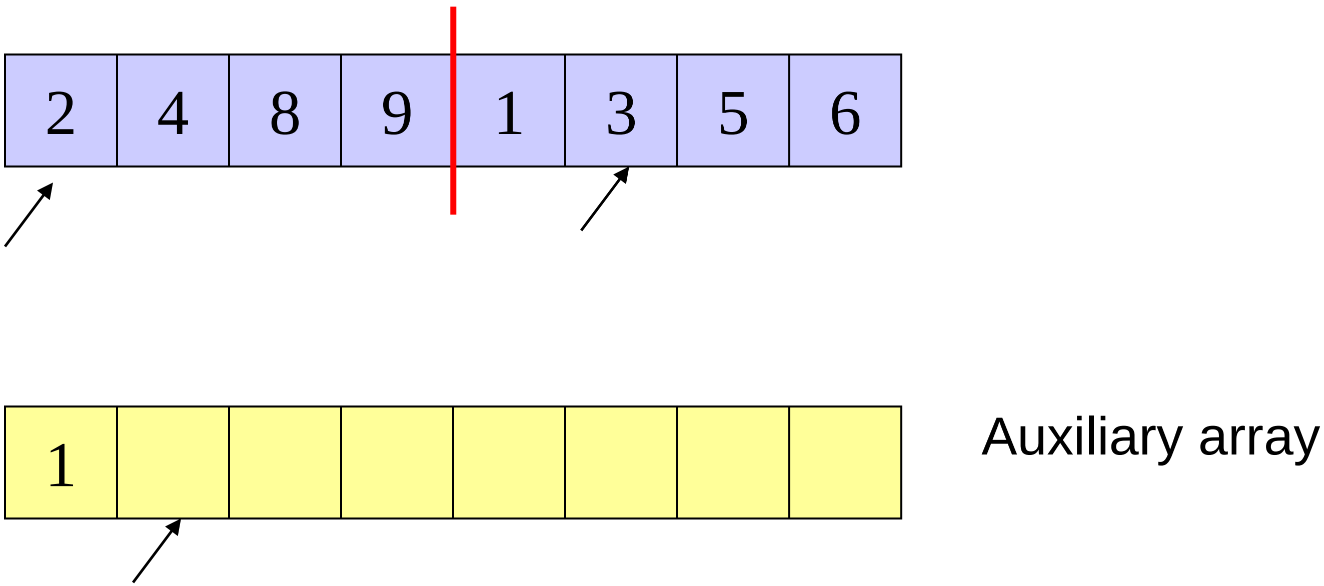
Auxiliary Array

- The merging requires an auxiliary array.



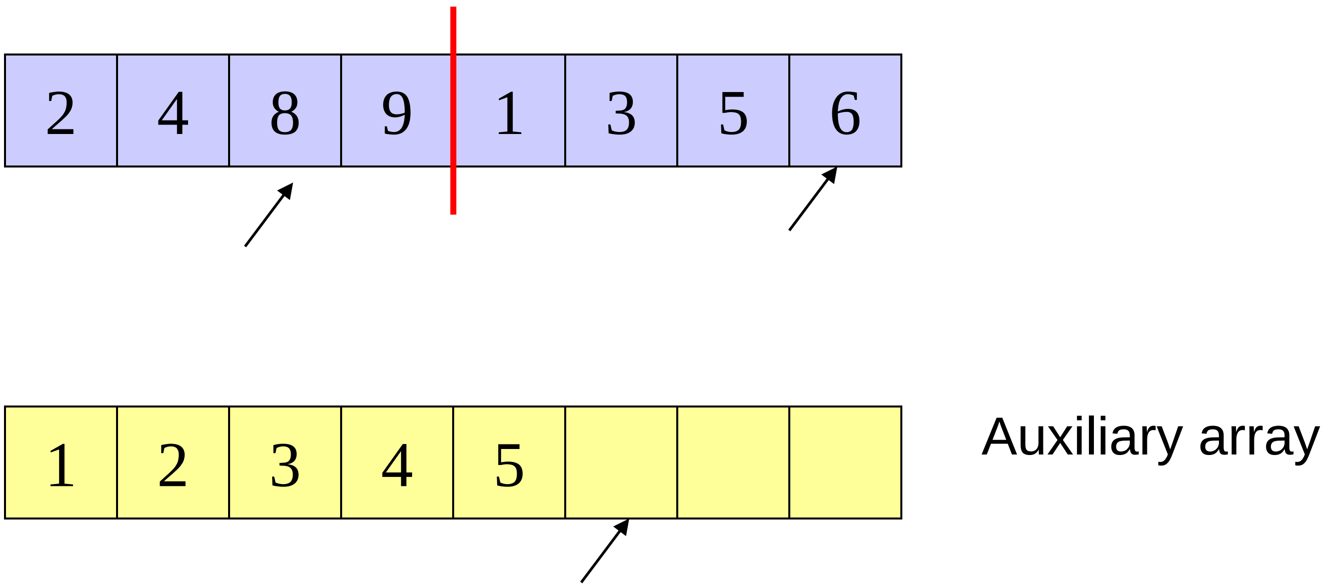
Auxiliary Array

- The merging requires an auxiliary array.

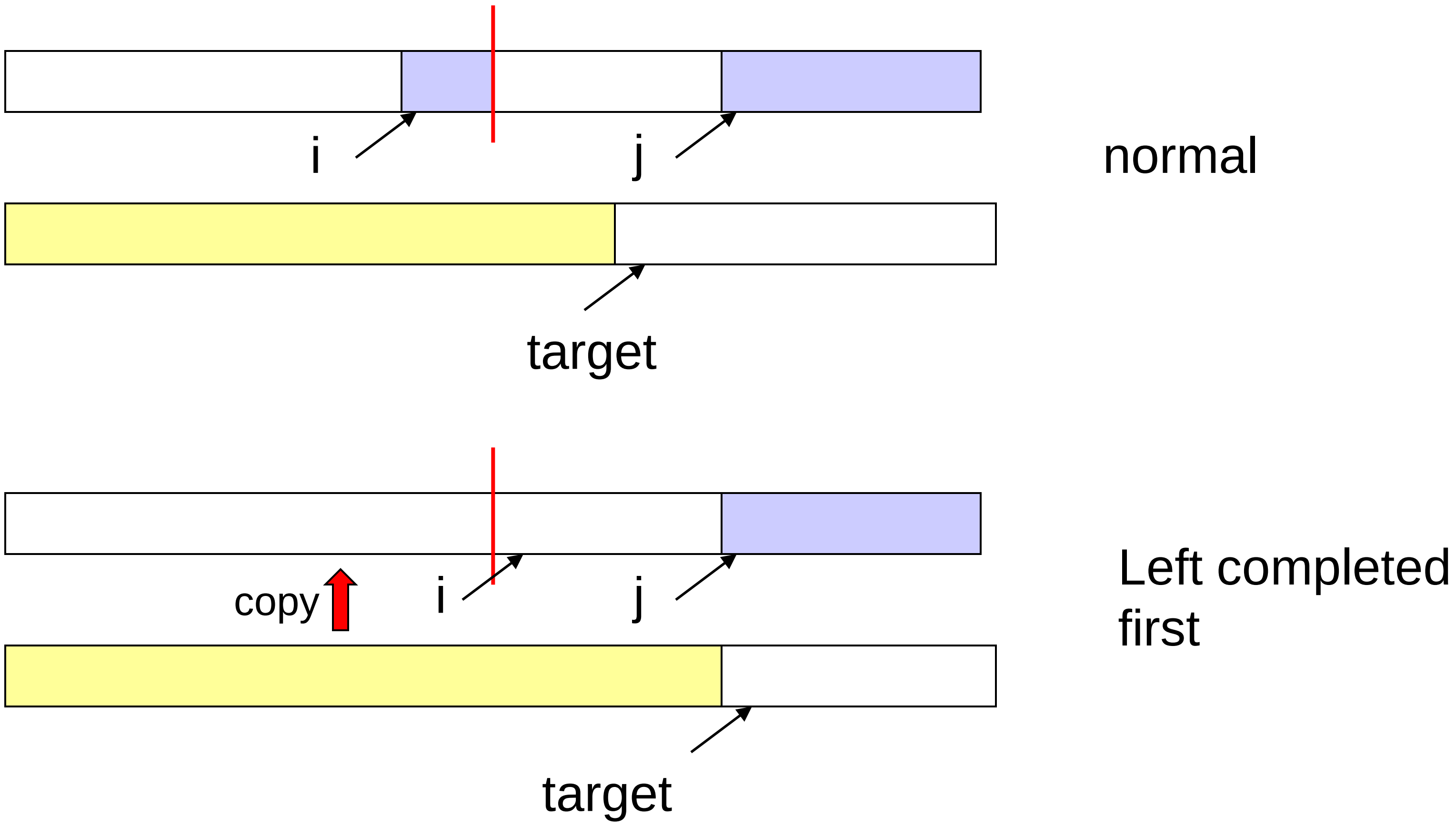


Auxiliary Array

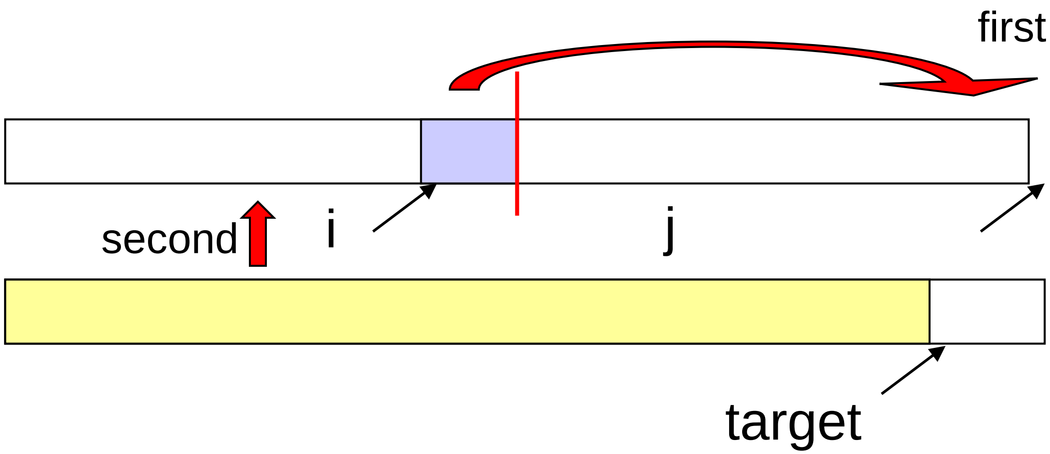
- The merging requires an auxiliary array.



Merging



Merging



Right completed first

Merging Algorithm

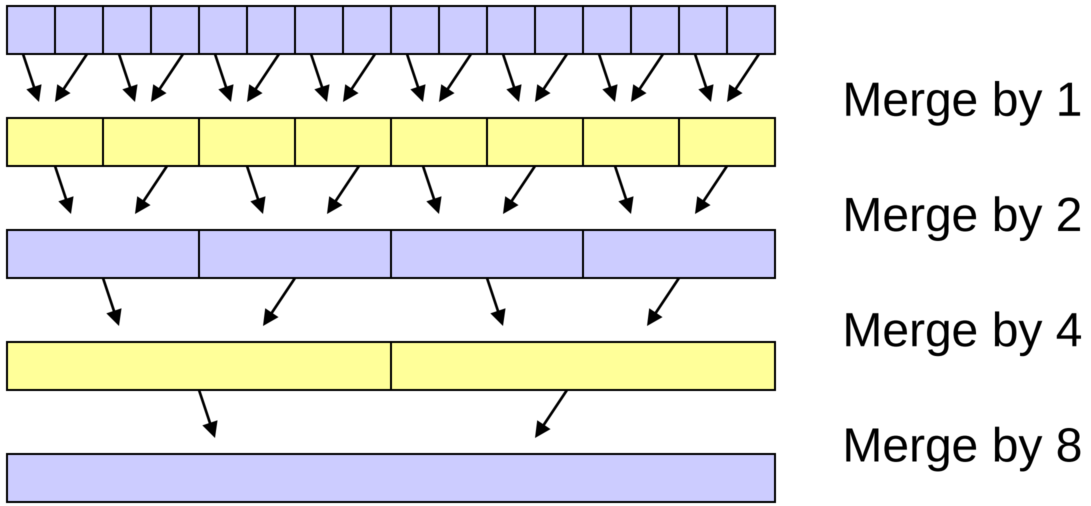
```
Merge(A[], T[] : integer array, left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;
  i := left; j := mid + 1; target := left;
  while i ≤ mid and j ≤ right do
    if A[i] ≤ A[j] then T[target] := A[i] ; i:= i + 1;
      else T[target] := A[j]; j := j + 1;
    target := target + 1;
  if i > mid then //left completed//
    for k := left to target-1 do A[k] := T[k];
  if j > right then //right completed//
    k : = mid; l := right;
    while k ≥ i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
} // Only stored elements are stored in target[]
```

Recursive Mergesort

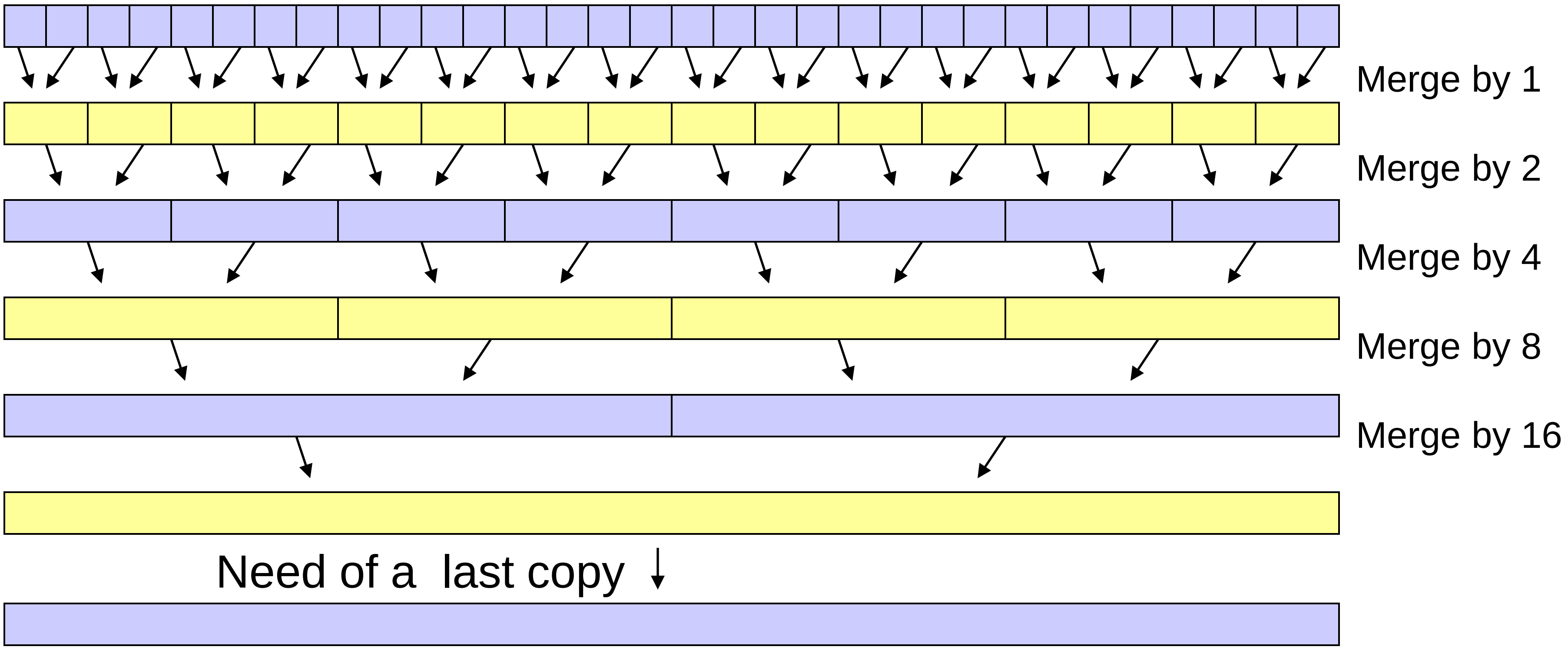
```
Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A,T,left,mid);
    Mergesort(A,T,mid+1,right);
    Merge(A,T,left,right);
}

MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort[A,T,1,n];
}
```

Iterative Mergesort

uses 2 arrays; alternates between them



Iterative Mergesort



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

Properties of Mergesort

- Not in-place
  - Requires an auxiliary array (O(n) extra space)
- Stable
  - Mergesort algorithm is stable if make sure that left is sent to target on equal values.

归并排序算法实现

\递归实现归并排序的原理如下:

递归分割:

<table>
<tr><td colspan="8">34 23 12 55 66 4 2 99 1 45 77 88 99 5</td></tr>
<tr><td colspan="4">34 23 12 55 66 4 2</td><td colspan="4">99 1 45 77 88 99 5</td></tr>
<tr><td colspan="2">34 23 12 55</td><td colspan="2">66 4 2</td><td colspan="2">99 1 45 77</td><td colspan="2">88 99 5</td></tr>
<tr><td>34 23</td><td>12 55</td><td>66 4</td><td>2</td><td>99 1</td><td>45 77</td><td>88 99</td><td>5</td></tr>
</table>

递归到达底部后排序返回:

<table>
<tr><td>23 34</td><td>12 55</td><td>4 66</td><td>2</td><td>1 99</td><td>45 77</td><td>88 99</td><td>5</td></tr>
<tr><td colspan="2">12 23 34 55</td><td colspan="2">2 4 66</td><td colspan="2">1 45 77 99</td><td colspan="2">88 99 5</td></tr>
<tr><td colspan="4">34 23 12 55 66 4 2</td><td colspan="4">99 1 45 77 88 99 5</td></tr>
<tr><td colspan="8">1 2 4 5 12 23 34 45 55 66 77 88 99 99</td></tr>
</table>

Quicksort

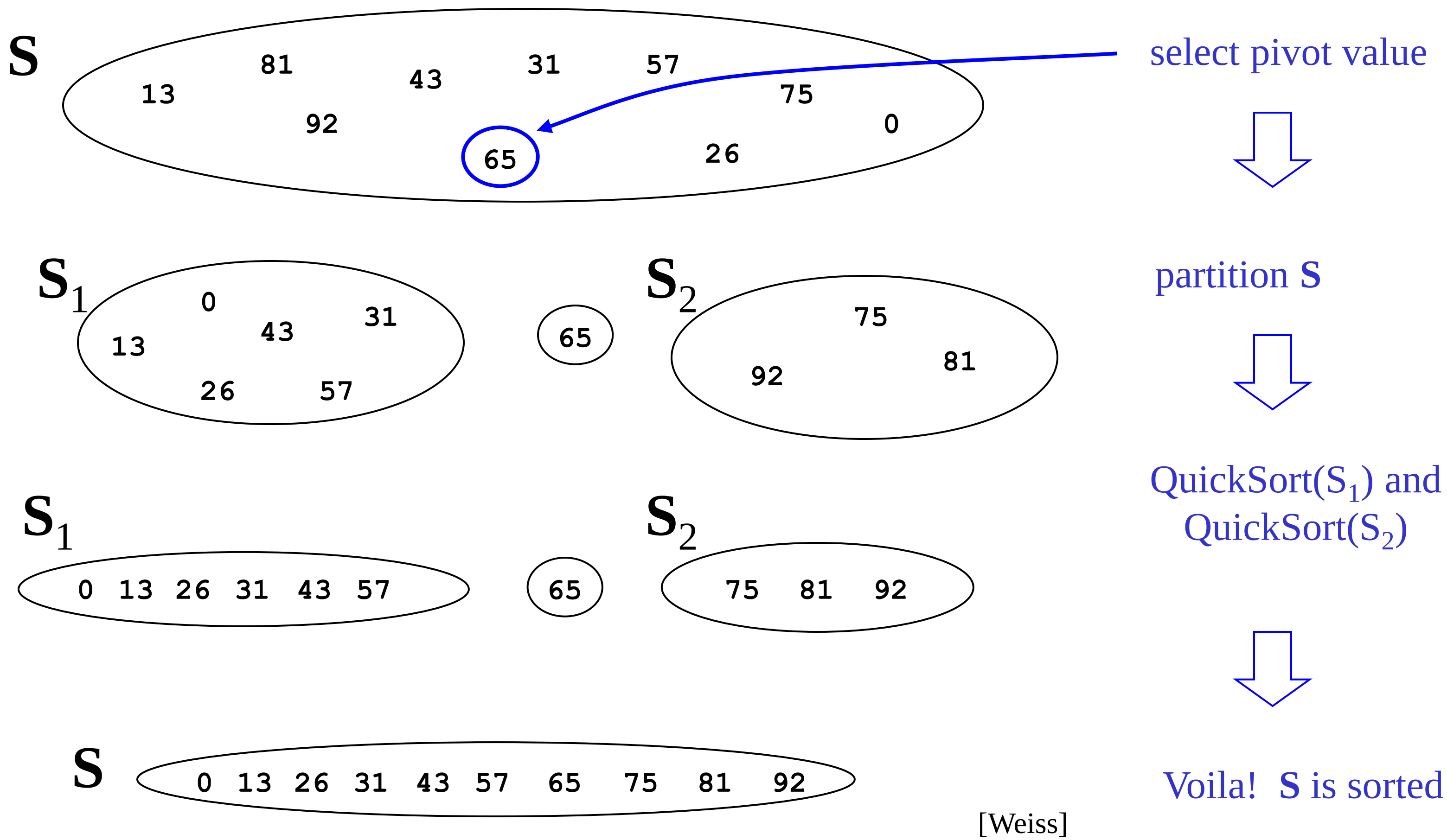
Quicksort

- 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, $\mathbf{S}_1 = \{\text{all values } x \le v\}$, and $\mathbf{S}_2 = \{\text{all values } x \ge v\}$.
  4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

The steps of QuickSort



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

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
  - the elements in left sub-array are ≤ pivot
  - elements in right sub-array are ≥ pivot
- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions

Partitioning:Choosing the pivot

- One implementation (there are others)
  - › median3 finds pivot and sorts left, center, right
    - Median3 takes the median of leftmost, middle, and rightmost elements
    - An alternative is to choose the pivot randomly (need a random number generator; “expensive”)
    - Another alternative is to choose the first element (but can be very bad. Why?)
  - › Swap pivot with next to last element

Partitioning in-place

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] > pivot
- Decrement j until you hit elmt A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross
- Swap pivot (at A[N-2]) with A[i]

Example

Choose the pivot as the median of three

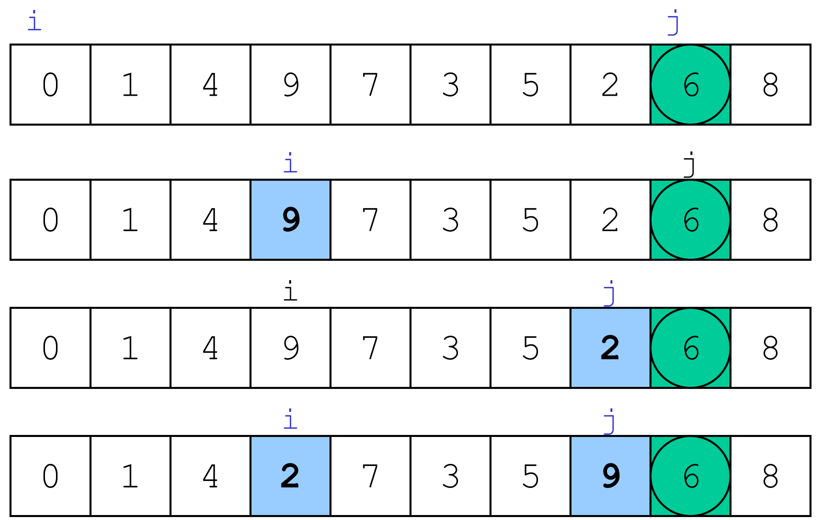
0	1	2	3	4	5	6	7	8	9
8	1	4	9	0	3	5	2	7	6

Median of 0, 6, 8 is 6. Pivot is 6

0	1	4	9	7	3	5	2	6	8
i								j	

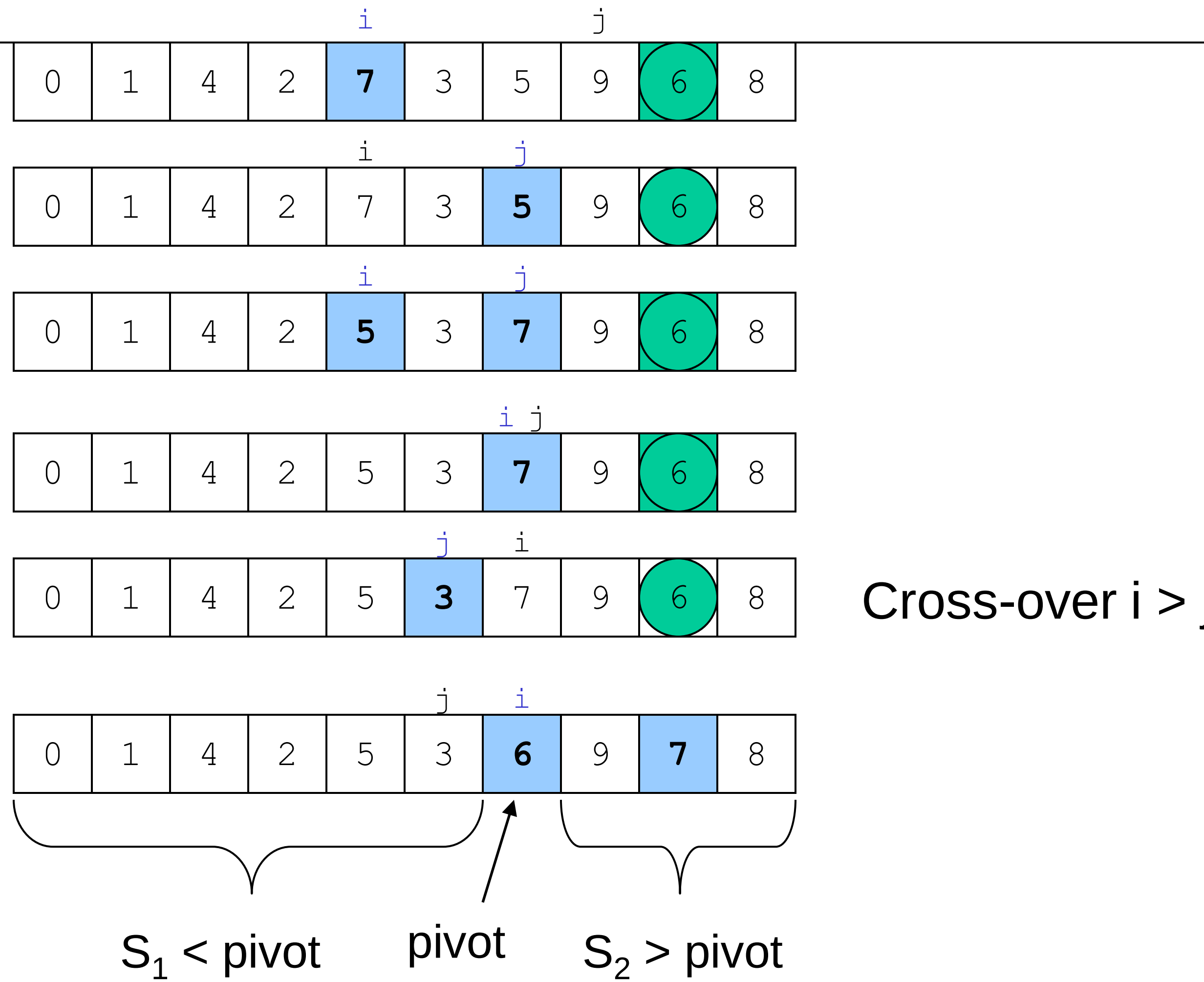
Place the largest at the right
and the smallest at the left.
Swap pivot with next to last element.

Example

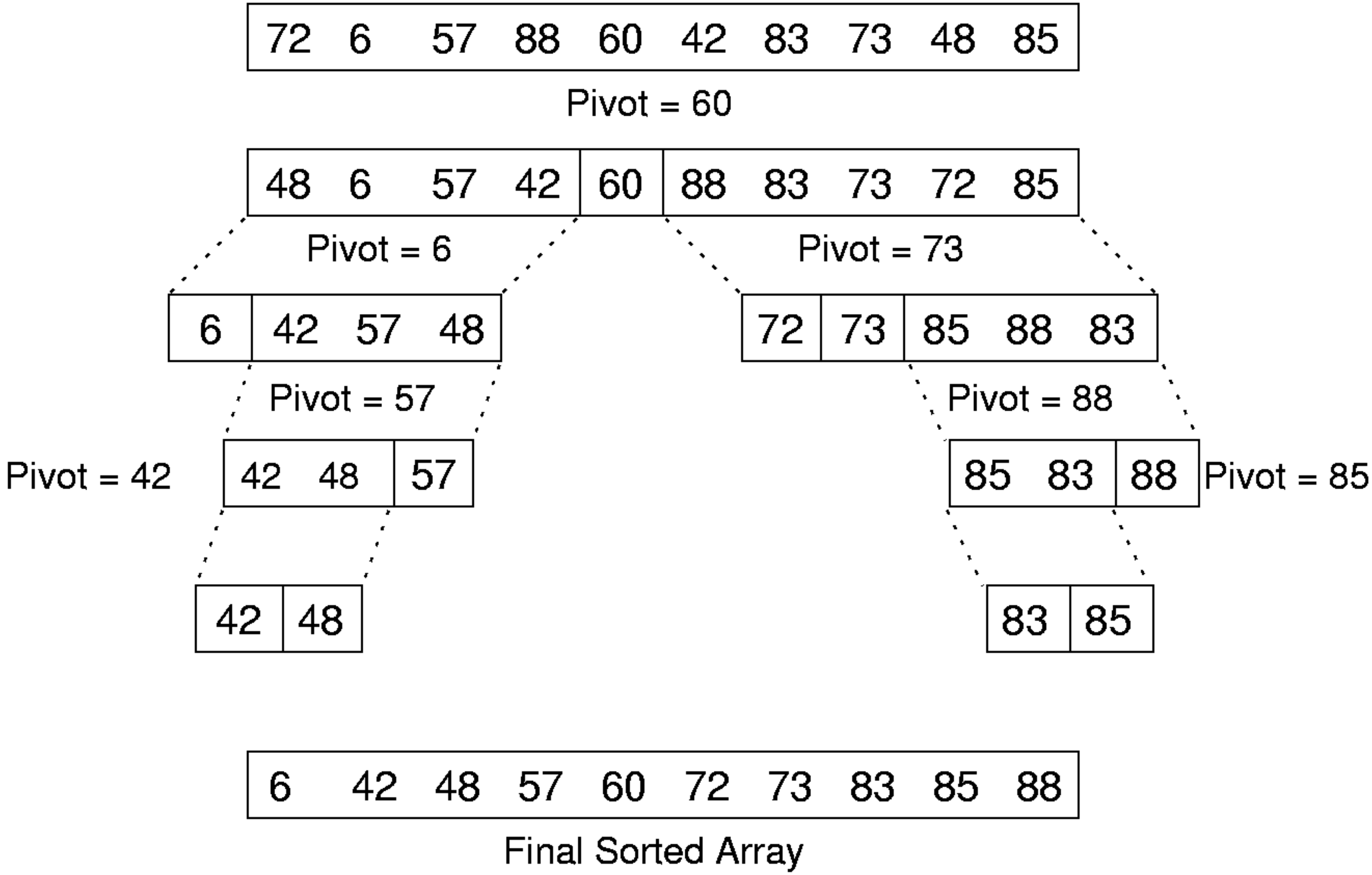


Move i to the right up to A[i] larger than pivot.
Move j to the left up to A[j] smaller than pivot.
Swap

Example


i j
0 1 4 2 7 3 5 9 6 8
i j
0 1 4 2 7 3 5 9 6 8
i j
0 1 4 2 5 3 7 9 6 8
i j
0 1 4 2 5 3 7 9 6 8
j i
0 1 4 2 5 3 7 9 6 8
Cross-over i > j
j i
0 1 4 2 5 3 6 9 7 8
S_1 < pivot
pivot
S_2 > pivot

Quicksort Example



Recursive Quicksort

```
Quicksort(A[]: integer array, left,right : integer): {
pivotindex : integer;
if left + CUTOFF ≤ right then
  pivot := median3(A,left,right);
  pivotindex := Partition(A,left,right-1,pivot);
  Quicksort(A, left, pivotindex – 1);
  Quicksort(A, pivotindex + 1, right);
else
  Insertionsort(A,left,right);
}
```

Don't use quicksort for small arrays.
CUTOFF = 10 is reasonable.

Quicksort Best Case Performance

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - › $T(0) = T(1) = O(1)$
    - constant time if 0 or 1 element
  - › For $N > 1$, 2 recursive calls plus linear time for partitioning
  - › $T(N) = 2T(N/2) + O(N)$
    - Same recurrence relation as Mergesort
  - › $T(N) = O(N \log N)$

Quicksort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - $T(N) = O(N^2)$

- Fortunately, *average case performance* is O(N log N) (see text for proof)

Properties of Quicksort

- Not stable because of long distance swapping.
- Pure quicksort not good for small arrays.
- “In-place”, but uses auxiliary storage because of recursive call (O(logn) space).
- O(n log n) average case performance, but $O(n^2)$ worst case performance.
- Quicksort is the best in-memory sorting algorithm.
- Quicksort algorithm is unstable.