Priority Queues and a first intro to sorting

CSE 373

Data Structures

Readings

- Reading
  - Chapter 8 Sections 8.1 – 8.2
  - Chapter 11 Section 11.1

Revisiting FindMin

- Application: Find the smallest (or highest priority) item quickly
  - Operating system needs to schedule jobs according to priority instead of FIFO
  - Event simulation (bank customers arriving and departing, ordered according to when the event happened)
  - Find student with highest grade, employee with highest salary etc.

Priority Queue ADT

- Priority Queue can efficiently do:
  - › FindMin() (called Min() in GT (your text book))
    - Returns minimum value but does not delete it
  - › DeleteMin( ) (called removeMin() in GT)
    - Returns minimum value and deletes it
  - › Insert (k)
    - In GT Insert (k,x) where k is the key and x the value. In all algorithms the important part is the key, a “comparable” item. We’ll skip the value.
  - › size() and isEmpty()

List implementation of a Priority Queue

- What if we use unsorted lists:
  - FindMin and DeleteMin are O(n)
    - In fact you have to go through the whole list
  - Insert(k) is O(1)
- What if we used sorted lists
  - FindMin and DeleteMin are O(1)
    - Be careful if we want both Min and Max (circular array or doubly linked list)
  - Insert(k) is O(n)
    - Recall Assignment 1!

Selection Sort

- Selection Sort
  - Sorts an unsorted list S into a sorted list T

```
While !S.isEmpty(){
  k := S.DeleteMin();
  T.addlast(k); // An easy simplification of Insert(k)
}
```

- Time complexity?
- Easy modification to do it in place

Insertion Sort

- Start with unsorted S and want sorted T

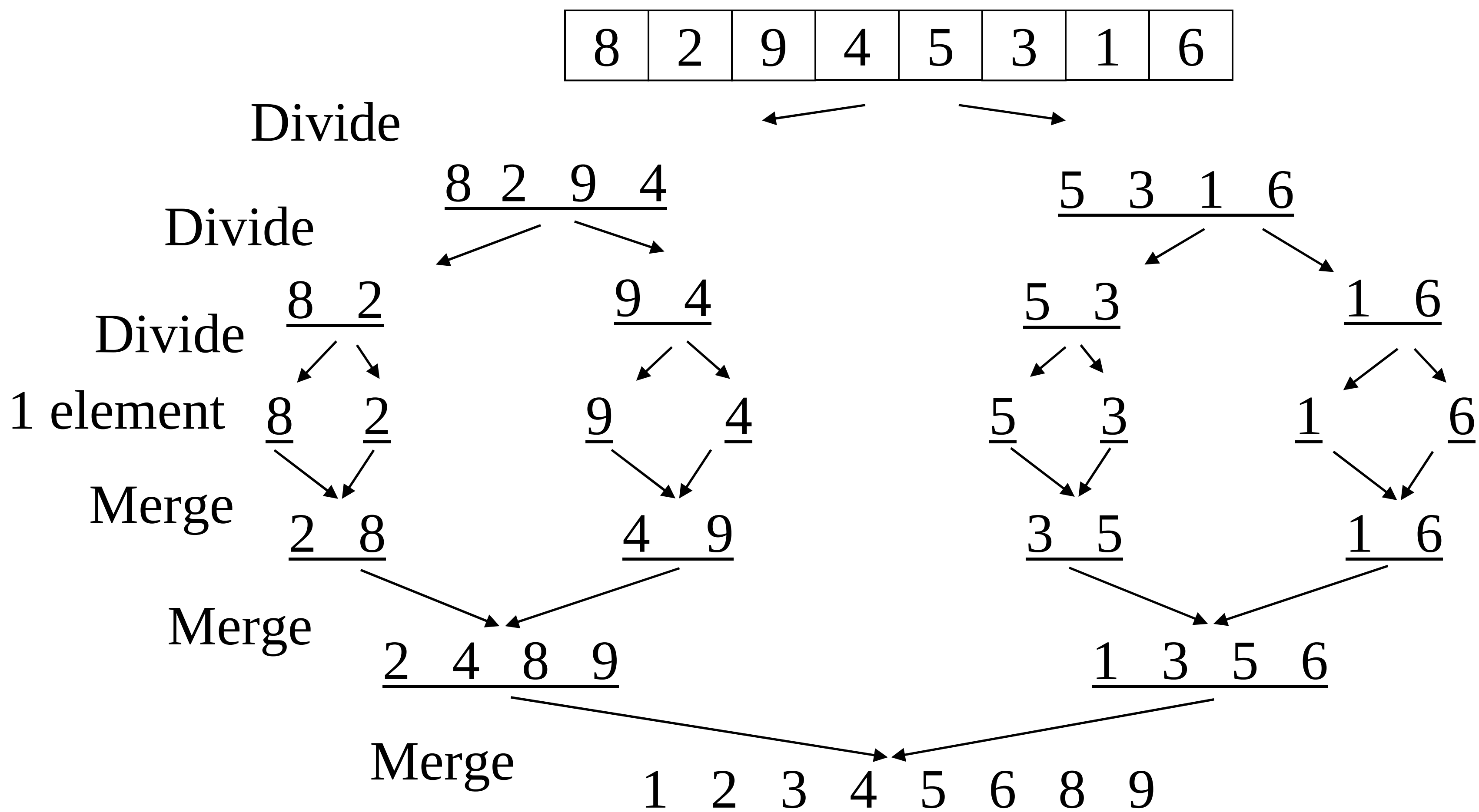
```
While !S.isEmpty() {
  k:= S.deletelast(); // or deletefirst whichever is easier
  T.Insert(k); // Insert so that T is sorted
}
```

- Complexity?
- Again easy to do it place.

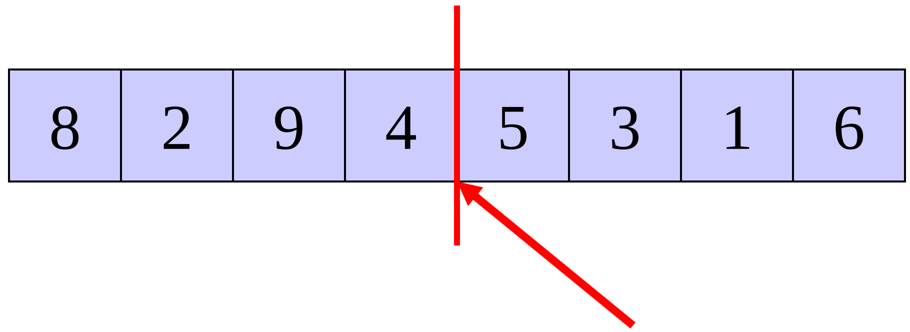
Mergesort: A More efficient sorting algorithm

- Uses a “Divide and Conquer” strategy
  - › Divide problem into smaller parts
  - › Independently solve the parts
  - › Combine these solutions to get overall solution
- **Main idea** Divide list into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort

Mergesort Example



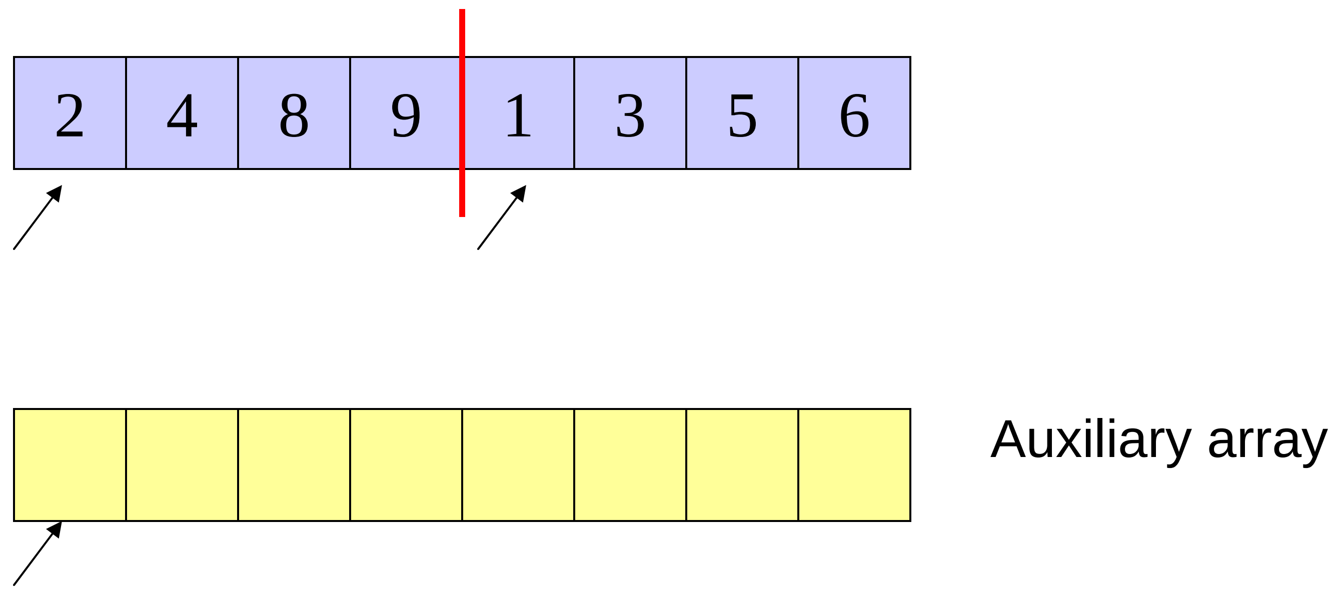
Mergesort (array implementation)



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

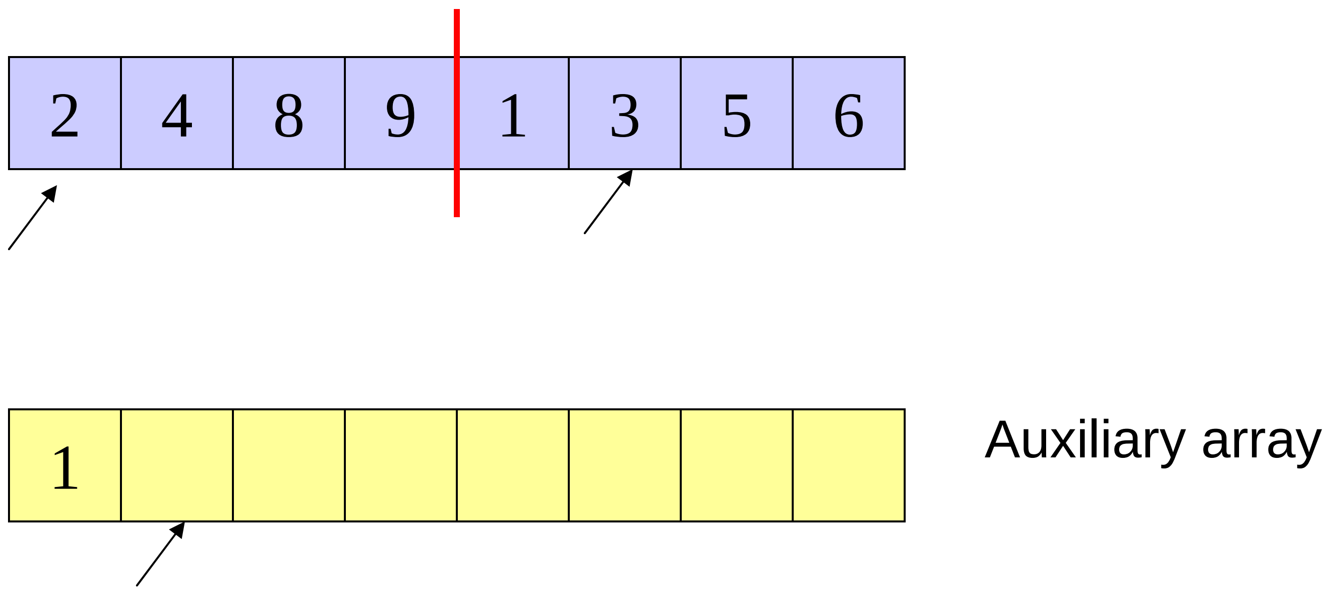
Auxiliary Array

- The merging requires an auxiliary array.



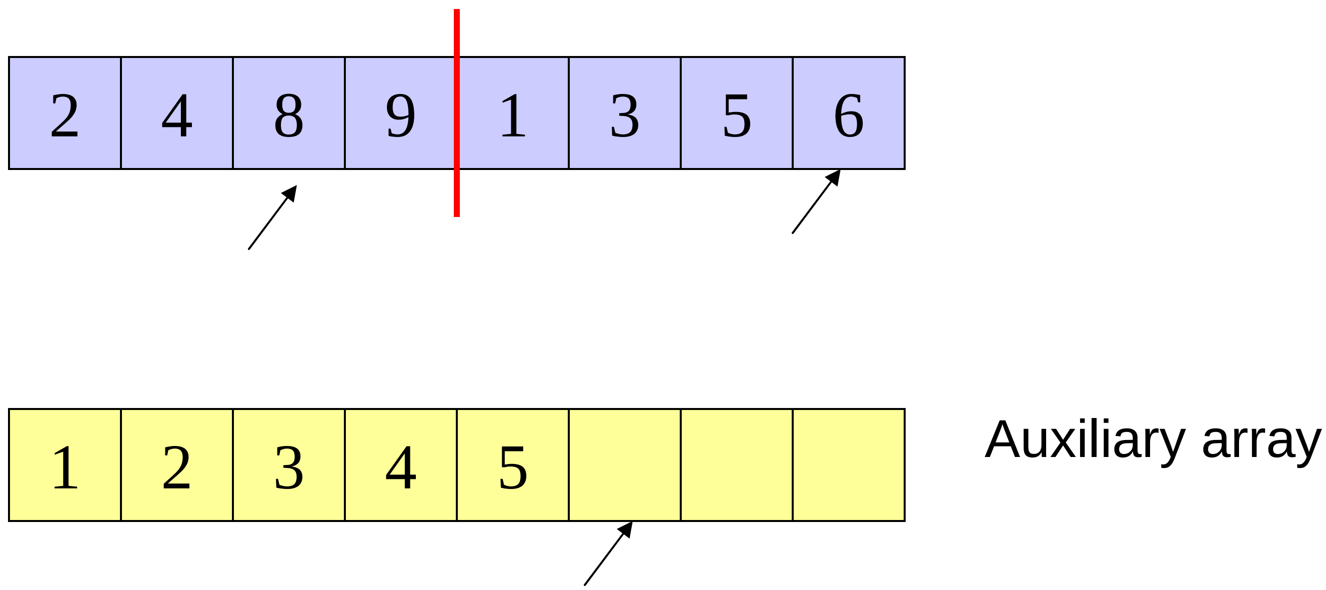
Auxiliary Array

- The merging requires an auxiliary array.

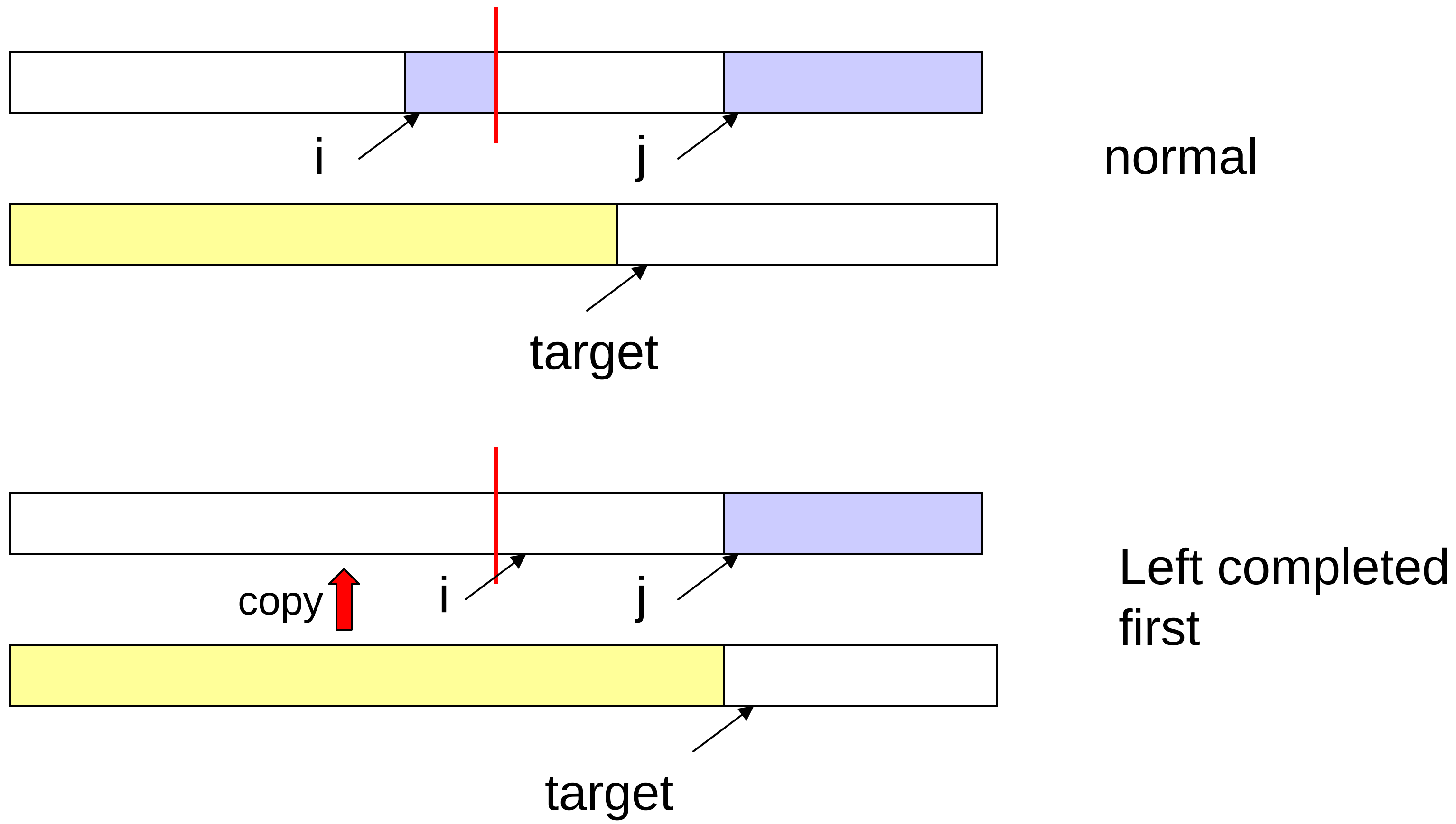


Auxiliary Array

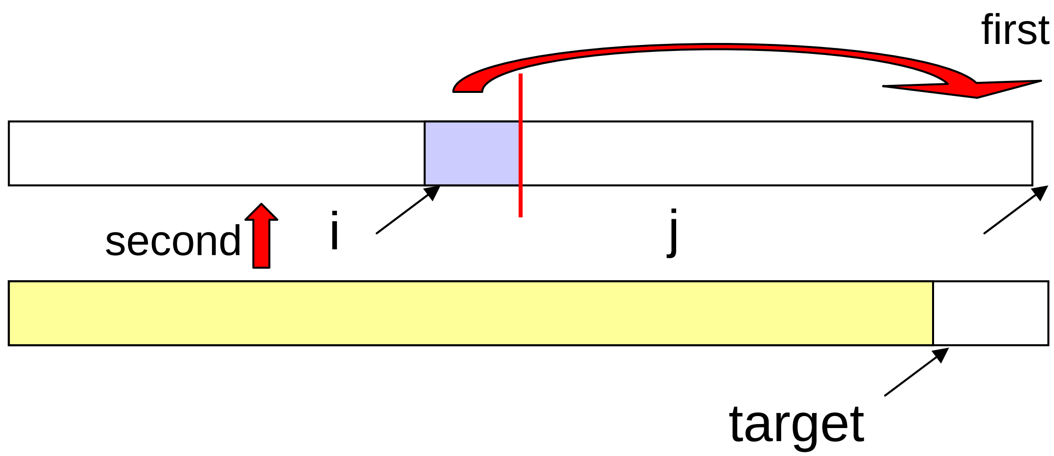
- The merging requires an auxiliary array.



Merging



Merging



Right completed first

Merging

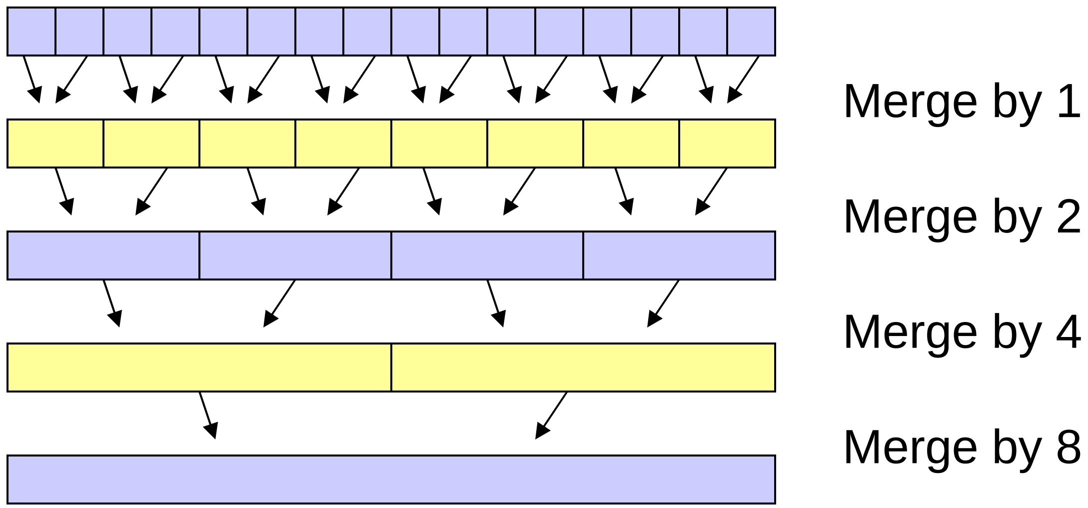
```
Merge(A[], T[] : integer array, left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;
  i := left; j := mid + 1; target := left;
  while i ≤ mid and j ≤ right do
    if A[i] ≤ A[j] then T[target] := A[i] ; i:= i + 1;
      else T[target] := A[j]; j := j + 1;
    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];
}
```

Recursive Mergesort

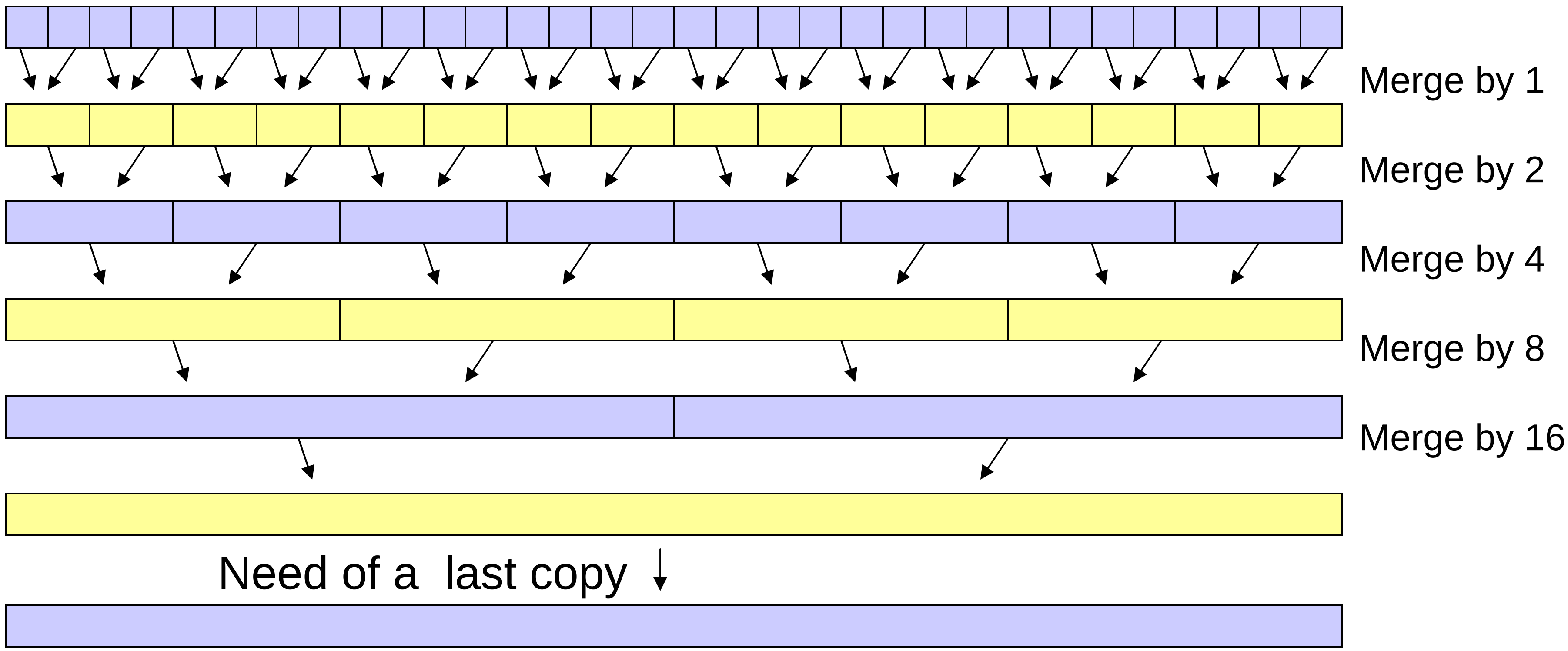
```
Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A,T,left,mid);
    Mergesort(A,T,mid+1,right);
    Merge(A,T,left,right);
}

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

Iterative Mergesort



Iterative Mergesort



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?

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

Mergesort Analysis
Upper Bound

$$
\begin{aligned}
T(n) &\le 2T(n/2) + dn \qquad \text{Assuming } n \text{ is a power of } 2 \\
&\le 2(2T(n/4) + dn/2) + dn \\
&= 4T(n/4) + 2dn \\
&\le 4(2T(n/8) + dn/4) + 2dn \\
&= 8T(n/8) + 3dn \\
&\vdots \\
&\le 2^k T(n/2^k) + kdn \\
&= nT(1) + kdn \qquad \text{if } n = 2^k \\
&\le cn + dn \log_2 n \\
&= O(n \log n)
\end{aligned}
$$

$n = 2^k$, $k = \log n$

Properties of Mergesort

- Not in-place
  - Requires an auxiliary array (O(n) extra space)
- Stable (sorting does not modify the relative positions of equal values)
  - Make sure that left is sent to target on equal values.