ECE36800 Data structures

Mergesort

Chapter 8 (pp. 335-361)

Lu Su

School of Electrical and Computer Engineering
Purdue University

Overview

- Recursive algorithm
- Iterative algorithm
- Theoretical lower bound of exchange-sort

- Reference pages pp. 335-361

Recursive mergesort

- Mergesort algorithm

1) Split array into two subarrays of about the same size
2) Sort each subarray recursively (if size > 1)
3) Merge the sorted subarrays

- Complexity
  - Recursion depth is at most log n + 1 = O(log n)
  - If merging two sorted subarrays takes O(n), mergesort is O(n log n)

Mergesort: algorithm

- Sort n integers r[0] to r[n-1] in ascending order
- Call mergesort(r, 0, n – 1)

```
mergesort (r[], lidx, ridx):
  if lidx ≥ ridx
    return;
  // divide
  mid = (lidx+ridx)/2; // assume no overflow
  // conquer left subarray
  mergesort(r, lidx, mid);
  // conquer right subarray
  mergesort(r, mid+1, ridx);
  // merge subarrays
  merge(r, lidx, mid, ridx)
```

Mergesort

- Another divide-and-conquer algorithm (natural to use recursion)
- Quicksort: “difficult” division, easy combination, preorder

```
pivot_idx = partition(r, lidx, ridx);
qsort(r, lidx, pivot_idx-1);
qsort(r, pivot_idx+1, ridx)
```

- Mergesort: easy division, “difficult” combination, postorder

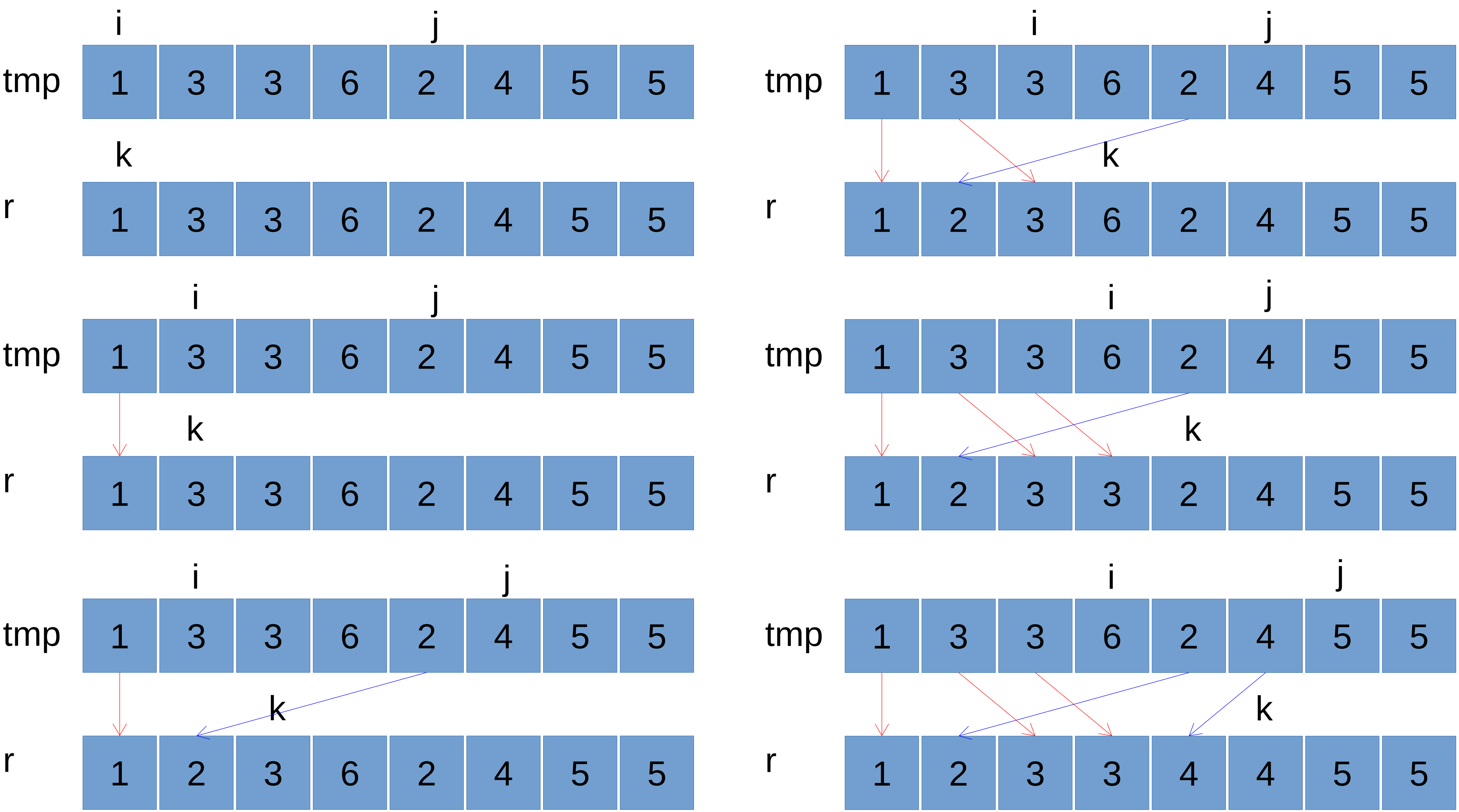
```
mergesort(r, lidx, mid);
mergesort(r, mid+1, ridx);
merge(r, lidx, mid, ridx)
```

Mergesort: merge

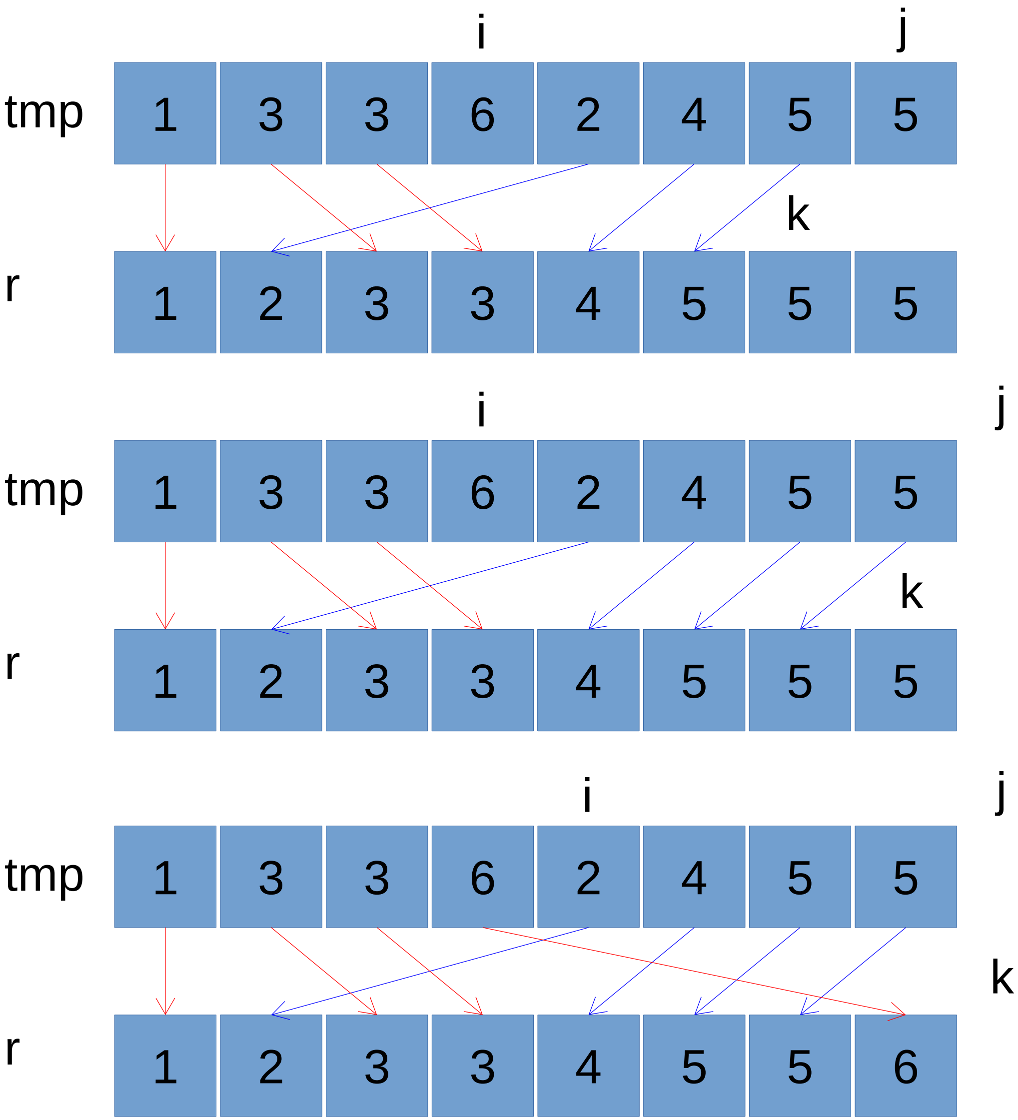
- Merge sorted subarrays r[lidx..mid] and r[mid+1..ridx] in ascending order
- Assume auxiliary space tmp[0..n-1], where n is the number of elements in the array
- The function memcpy copies the contents at second parameter to the location at first parameter; the number of elements copied is specified by third parameter
  - Not exactly the function in string.h

```
merge (r[], lidx, mid, ridx):
  memcpy(&tmp[lidx], &r[lidx], mid-lidx+1);
  memcpy(&tmp[mid+1], &r[mid+1], ridx-mid);
  i ← lidx; j ← mid+1;
  for k ← lidx to ridx
    if (i > mid) r[k] ← tmp[j++];
    else if (j > ridx) r[k] ← tmp[i++];
    else if (tmp[j] < tmp[i])
           r[k] ← tmp[j++];
    else r[k] ← tmp[i++];
```

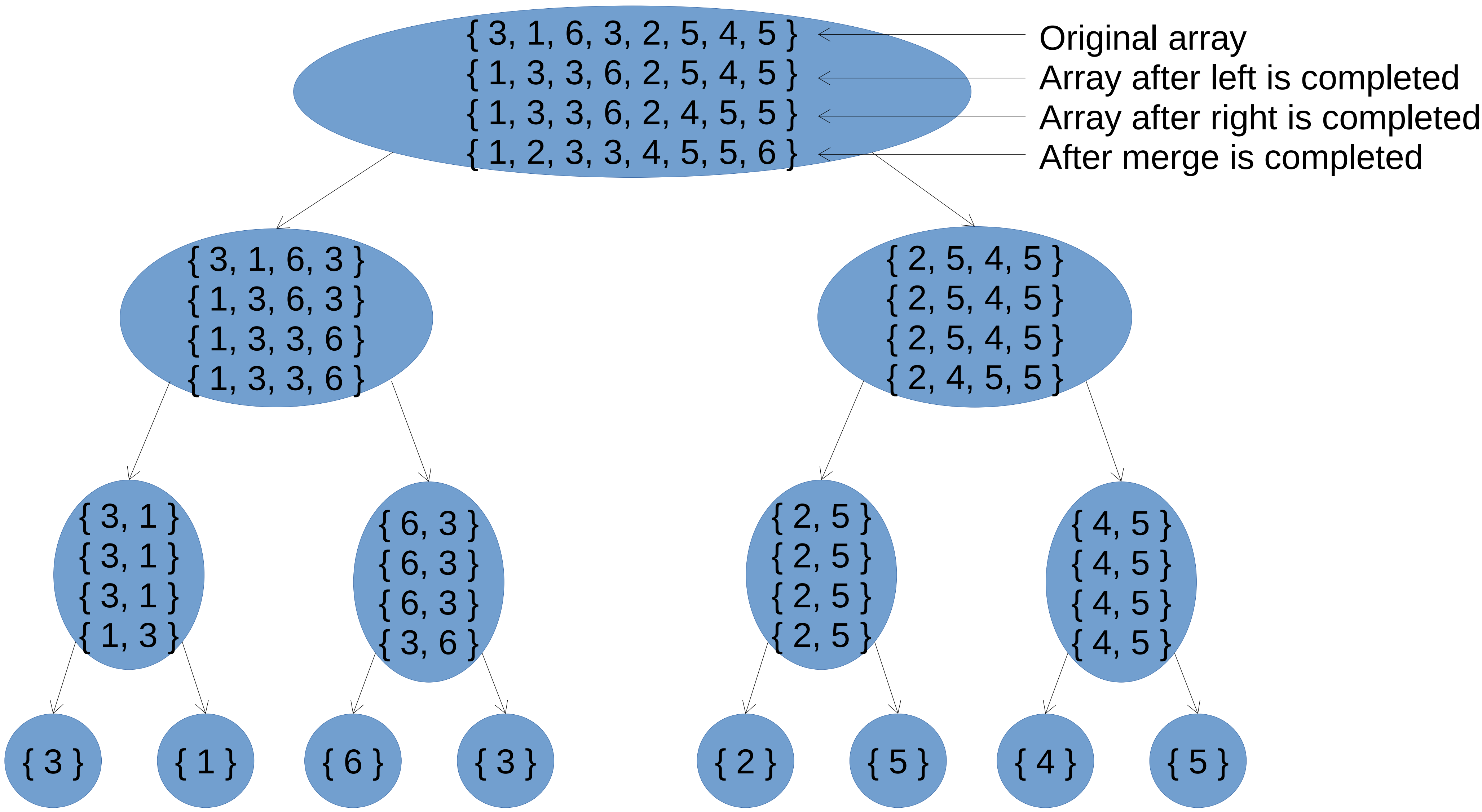
Merge example



Merge example



Mergesort example



Mergesort: Iterative algorithm

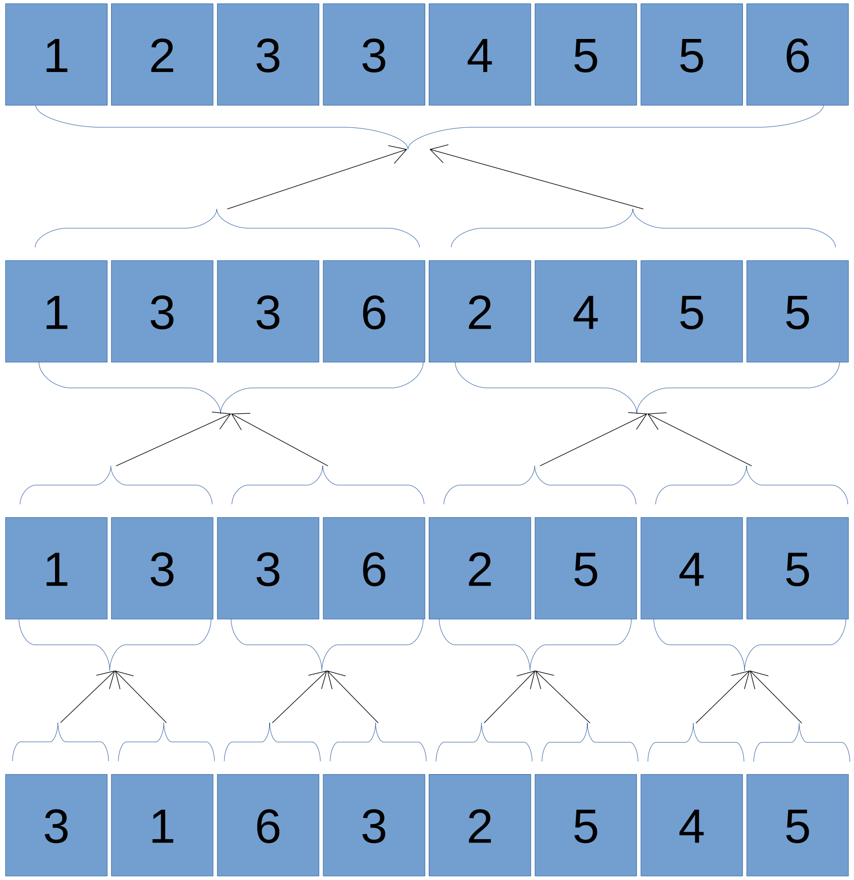
- Bottom-up process
- Leaf nodes: Consider original array as n subarrays, each of size 1
- Scan through array performing n/2 times, merging of two 1-element arrays to produce n/2 sorted subarrays, each of size 2
- Scan through array performing n/4 times, merging of two 2-element arrays to produce n/4 sorted subarrays, each of size 4
- …
- Perform merging of two n/2-element arrays to produce the final sorted array of n elements

Iterative mergesort example

Merge 4-element subarrays

Merge 2-element subarrays

Merge 1-element subarrays

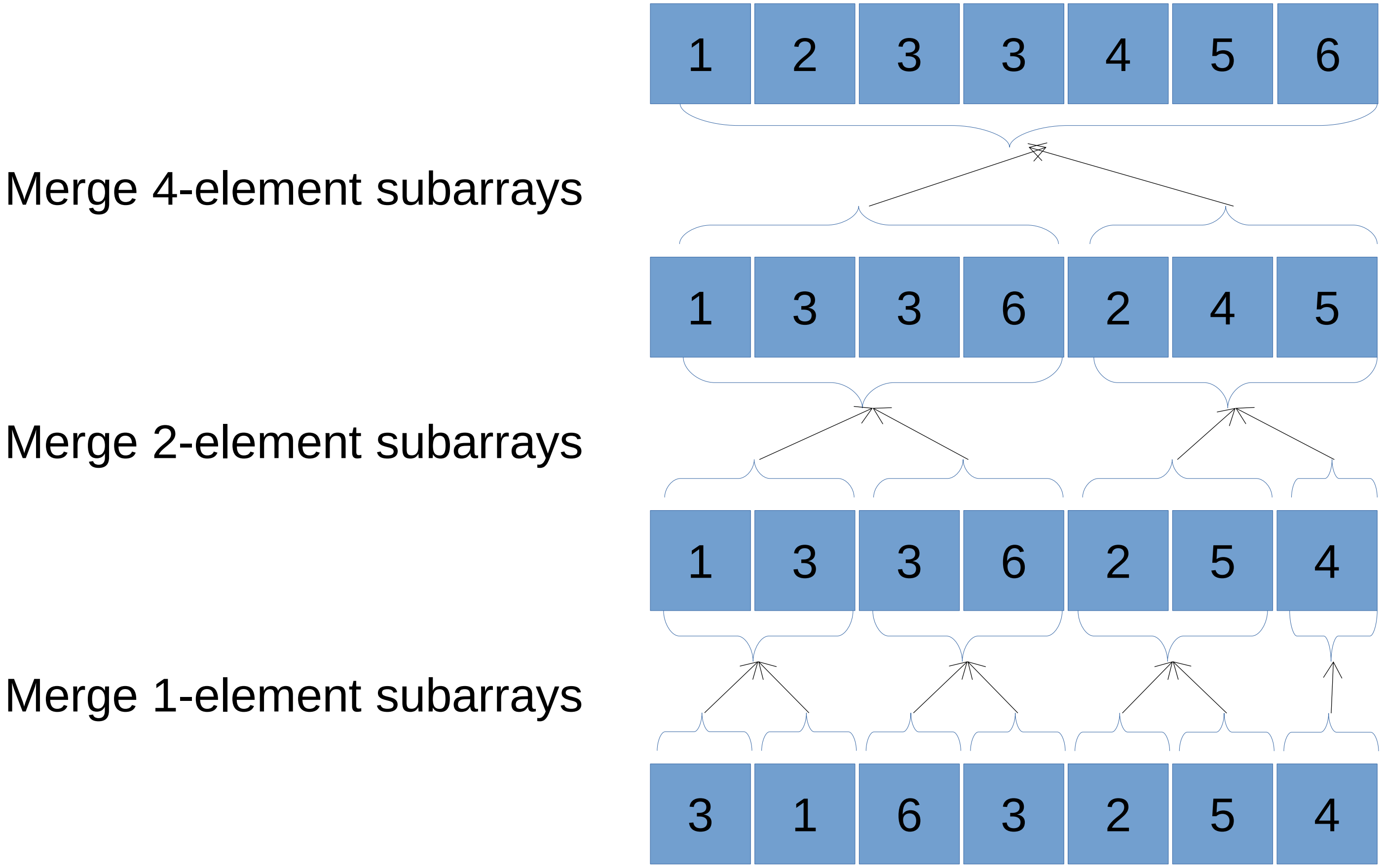


Mergesort: Iterative algorithm

- Iteratively mergesort n integers r[0..n-1] in ascending order

```
iter_mergesort (r[], n):
  size ← 1
  while size < n
    i ← 0
    while i < n – size
      merge(r, i, i+size-1,
            min(i+2*size-1, n-1));
      i ← i + 2*size
    size ← 2*size;
```

Iterative mergesort example



Properties of mergesort

- Memory complexity: $O(n)$, not in-place
  - $O(n)$ copy operations in each pass
  - Main disadvantage to quicksort or heapsort
  - In-place variant exists, but large overhead
- Time complexity: $O(n \log n)$
  - For a sorted array, worse than insertion sort
- Can be stable if merge() is stable
- Easier to implement than quicksort and heapsort
- In practice slower that quicksort, but faster than heapsort
- Advantage: sequential access of data
  - Excellent for external (out-of-core) sorting (e.g. disk)
  - Good cache usage

Bitonic merge

- Inefficiency in merge function: Checks whether indices are out-of-bound
- Use sentinel (as in insertion sort) to avoid checking whether indices are within bounds
  - Not stable

```
bsmerge (r[], lidx, mid, ridx):
  memcpy(&tmp[lidx], &r[lidx], mid-lidx+1);
  j ← ridx;
  for k ← mid+1 to ridx
    tmp[k] ← r[j];  // copy in reverse
    j--;            // order
  i ← lidx; j ← ridx;
  for k ← lidx to ridx
    if (tmp[j] < tmp[i]) r[k] ← tmp[j--];
    else r[k] ← tmp[i++];
```

Time-complexity summary

Algorithms	Best	Average	Worst
Insertion	O(n)	O(n^2)	O(n^2)
Bubble	O(n)	O(n^2)	O(n^2)
Quick	O(n log n)	O(n log n)	O(n^2)
Simple selection	O(n^2)	O(n^2)	O(n^2)
Heap	O(n)	O(n log n)	O(n log n)
Merge	O(n log n)	O(n log n)	O(n log n)

O(n^2) vs. O($n \log n$)

- How can mergesort, quicksort, and heapsort achieve O($n \log n$) when insertion sort and bubble sort run at O(n^2)?
  - Mergesort, quicksort, and heapsort move elements far distances, correcting multiple inversions (incorrect ordering) at a time
  - Insertion and bubble sort correct one inversion at a time
- Why is quicksort worst-case O(n^2) while mergesort has no such problem?
  - The choice of pivot determines size of partitions, whereas mergesort cuts array in half every iteration
  - Simple selection sort uses the worst pivot in every iteration
- Is O(n log n) the best we can do?
  - Yes, if we use binary comparison on the key values

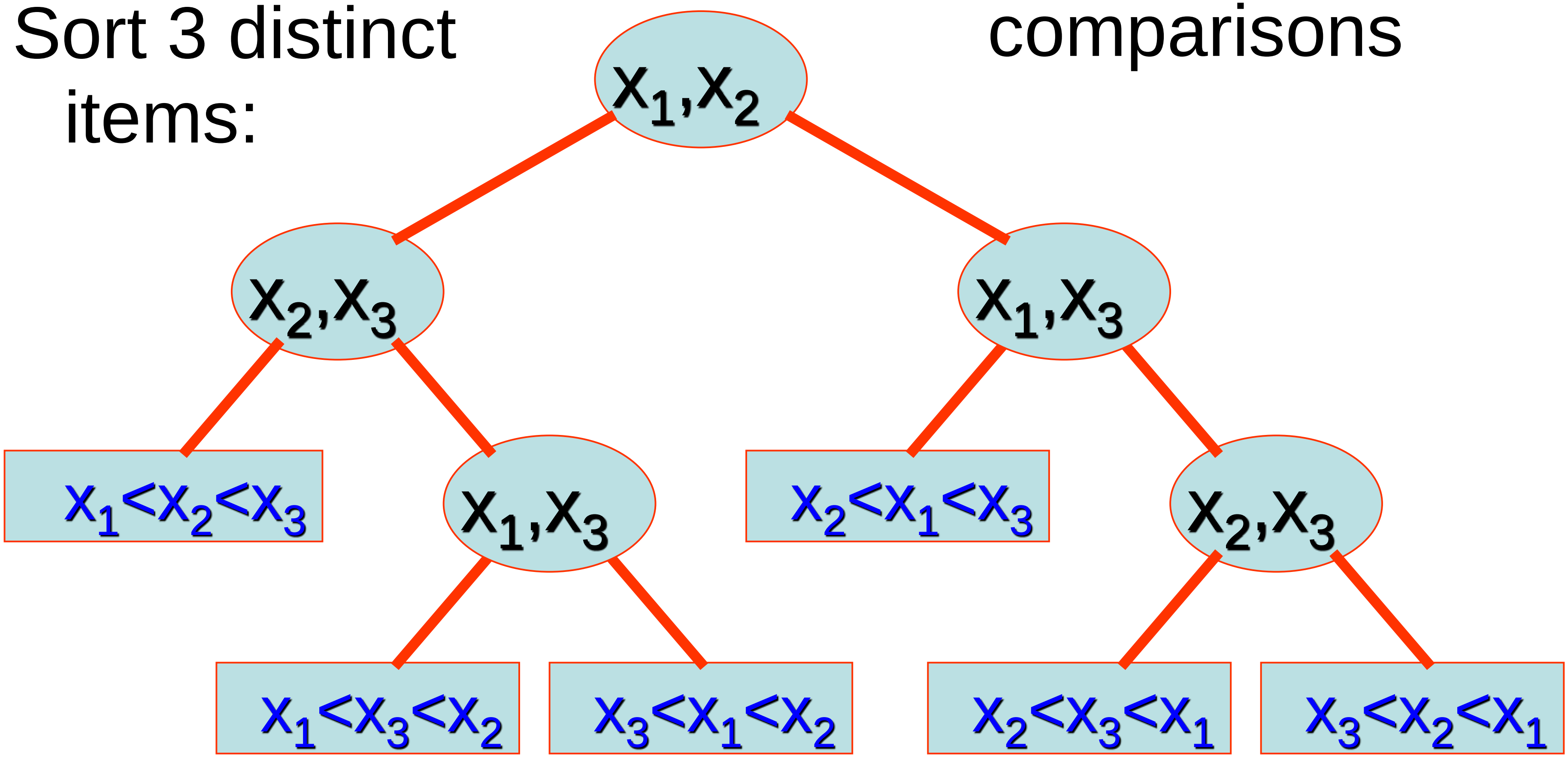
Theoretic lower bound

- *n* elements can have *n*! permutations
- A sort decision tree is a binary tree that represents a sorting method based on comparisons
- Each internal node compares two elements
  - Left branch implies first element smaller than second
  - Right branch otherwise
  - No redundant comparisons
- Each leaf nodes corresponds to one of the *n*! permutations

Binary decision tree for sorting

Sort 3 distinct items:

No redundant comparisons



leaf nodes = *n*!

Theoretic lower bound

- A binary decision tree for sorting has $(2n! - 1)$ nodes
- Height of tree is at least $\log n!$
- $n! \geq (n/2)^{(n/2)} \Rightarrow \log n!$ is $\Omega(n \log n)$
- Sorting algorithms that use comparisons must make $\Omega(n \log n)$ comparisons

Quick, Merge, or Heap?

- Quicksort is the fastest sorting algorithm in practice
  - Can perform as badly as $O(n^2)$
  - Requires $O(\log n)$ stack space
- Mergesort is guaranteed to run in $O(n \log n)$
  - Slower than quicksort on average
  - Requires $O(n)$ additional temporary storage
- Heapsort is guaranteed to run in $O(n \log n)$, in-place, and requires no recursions
  - Many real world tests show that heapsort is slower than quicksort (and mergesort) on average
  - Useful for really large problems