Sorting

Selection Sort

- For each pass, select the **minimum** value in the unsorted sequence and put it in its sorted place.
  - Find the smallest item, swap it with the item in the first position
    - Find the next smallest item, swap it with the item in the second position
      - Find the next smallest item, swap it with the item in third position
        - …
- N-1 passes

Selection Sort

pass	12	4	2	26	7	30	19	11	21

Selection Sort

Sorted
Minimum

pass	12	4	2	26	7	30	19	11	21
P=1	**2**	**4**	12	26	7	30	19	11	21
P=2	2	**4**	12	26	**7**	30	19	11	21
P=3	2	4	**7**	26	12	30	19	**11**	21
P=4	2	4	7	**11**	**12**	30	19	26	21
P=5	2	4	7	11	**12**	30	**19**	26	21
P=6	2	4	7	11	12	**19**	30	26	**21**
P=7	2	4	7	11	12	19	**21**	**26**	30
P=8	2	4	7	11	12	19	21	**26**	**30**

Running time = $O(n^2)$

Bubble Sort

- Compare adjacent items and swap them if the left item is less than the right item
- N-1 passes
- Naïve implementation
  - Continue for N-1 passes
- Optimizations
  - Check if no swapping on previous pass
  - On each pass, traverse 1 less item
  - Bubble-up and bubble-down

Bubble Sort

pass	10	14	2	16	7	21	8
P=1	10	2	14	7	16	8	21
P=2	2	10	7	14	8	16	21
P=3	2	7	10	8	14	16	21
P=4	2	7	8	10	14	16	21
P=5	2	7	8	10	14	16	21

Running time = $O(n^2)$ on all implementations. Can be *linear* if data already sorted – check for no swaps.

Insertion Sort

- Insert **next** item into correct position in sorted subset of items to be sorted so far.
- Make N-1 passes

Insertion Sort

pass	34	26	8	61	17	51	32	9	22
P=1	26	34	8	61	17	51	32	9	22
P=2	8	26	34	61	17	51	32	9	22
P=3	8	26	34	61	17	51	32	9	22
P=4	8	17	26	34	61	51	32	9	22
P=5	8	17	26	34	51	61	32	9	22
P=6	8	17	26	32	34	51	61	9	22
P=7	8	9	17	26	32	34	51	61	22
P=8	8	9	17	22	26	32	34	51	61

Running time = $O(n^2)$. Can be *linear* if data already sorted – check if item to be inserted is larger than largest item in sorted subsequence sorted so far.

Divide-and-Conquer

- Divide-and-conquer is a general algorithm design paradigm:
  - Divide: the problem into a number of smaller instances of same problem (sub-problems)
  - Conquer: the sub-problems by solving them recursively. If problem is small enough, solve directly.
  - Combine: the solutions to the sub-problems into solution for original problem

Examples:

- Merge sort
- Quick sort

Merge Sort

Divide-and-Conquer

- Divide: Divide the n-element sequence to be sorted into two subsequences of $n/2$ each, i.e., S into two disjoint subsets S_1 and S_2.
- Conquer: Sort the two subsequences S_1 and S_2 recursively using merge sort.
- Combine: Merge the two sorted sequences to produce sorted answer.

- The base case for the recursion are sub problems of size 0 or 1 (sorted).
- Merge Sort is a sorting algorithm based on the divide-and-conquer paradigm

Merging Two Sorted Sequences

- The combine step of merge sort consists of merging two sorted sequences A and B into a sorted sequence C
- Merging two sorted sequences, and implemented by means of a doubly linked list each with $n/2$ elements takes $O(n)$ time.

Algorithm ***merge***(*A*, *B*)
  Input sequences A and B with $n/2$ elements each
  Output sorted sequence of $A \cup B$

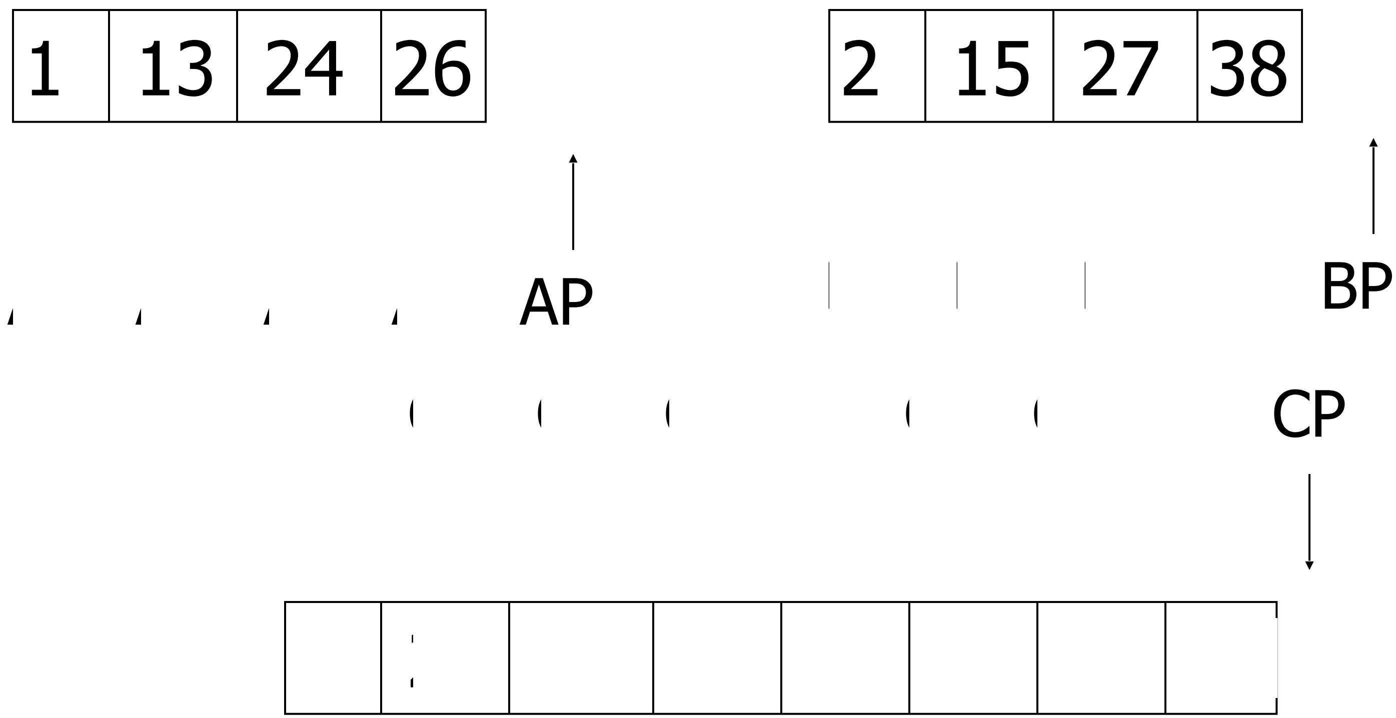
  $C \leftarrow$ empty sequence
  while $\neg$*A.isEmpty*() $\wedge$ $\neg$*B.isEmpty*()
    if *A.first*().*element*() < *B.first*().*element*()
      C.insertLast(*A.remove*(*A.first*()))
    else
      C.insertLast(*B.remove*(*B.first*()))
  while $\neg$*A.isEmpty*()
  C.insertLast(*A.remove*(*A.first*()))
  while $\neg$*B.isEmpty*()
  C.insertLast(*B.remove*(*B.first*()))
  return C

Merge Sort

- Merge Sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Conquer: recursively sort S_1 and S_2
  - Combine: merge S_1 and S_2 into a unique sorted sequence

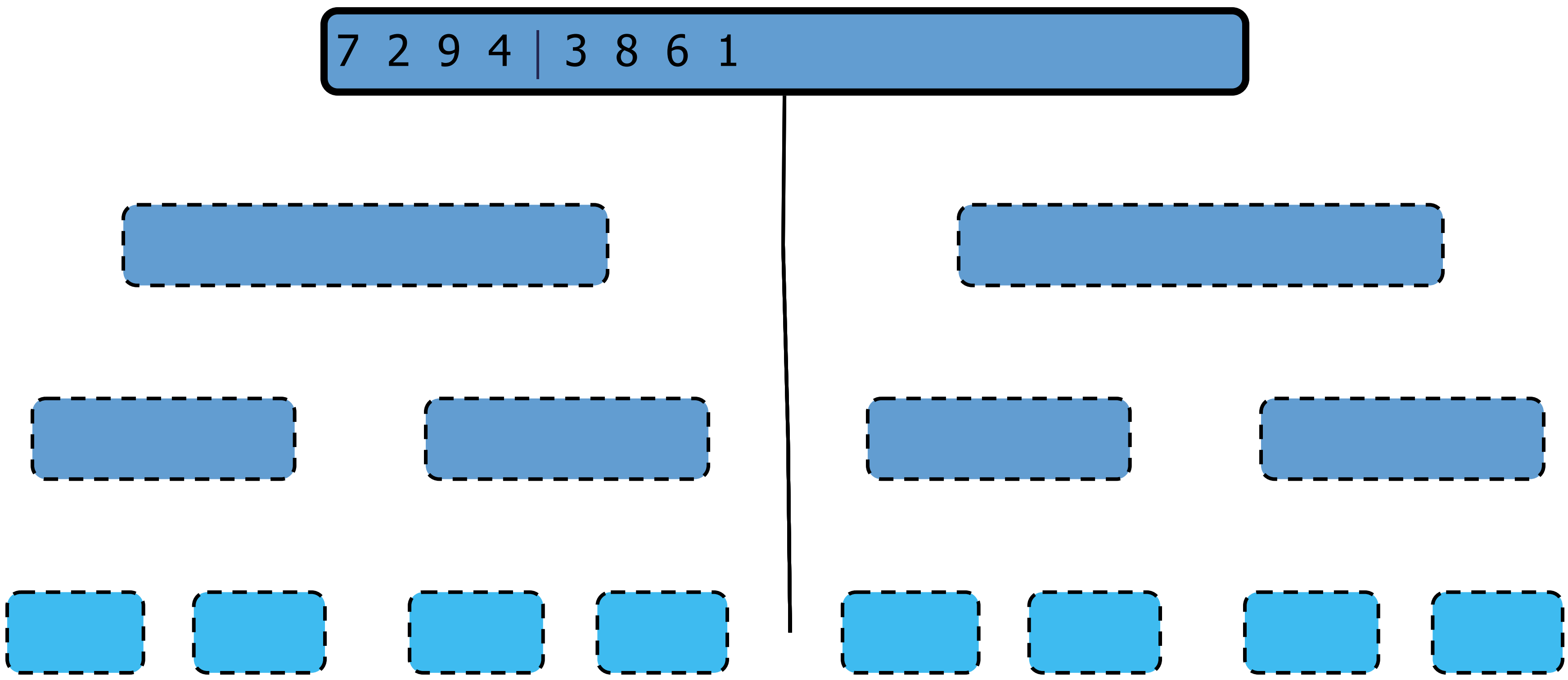
Algorithm *mergeSort*(S, C)
    Input sequence S with n elements, comparator C
    Output sequence S sorted according to C
    if $S.size() > 1$
        $(S_1, S_2) \leftarrow partition(S, n/2)$
        $mergeSort(S_1, C)$
        $mergeSort(S_2, C)$
        $S \leftarrow merge(S_1, S_2)$

Merge Sort - Merge



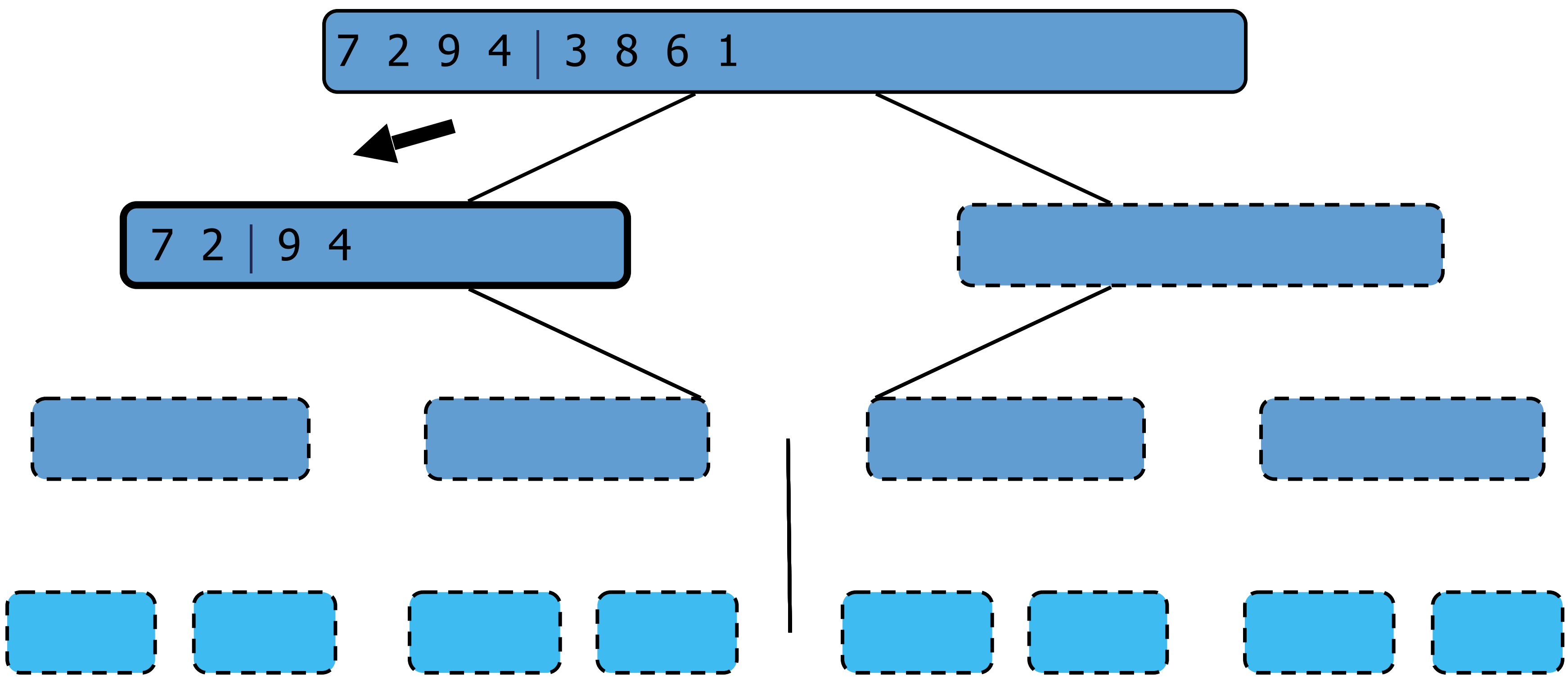
Merge Sort
Execution Example

- Partition



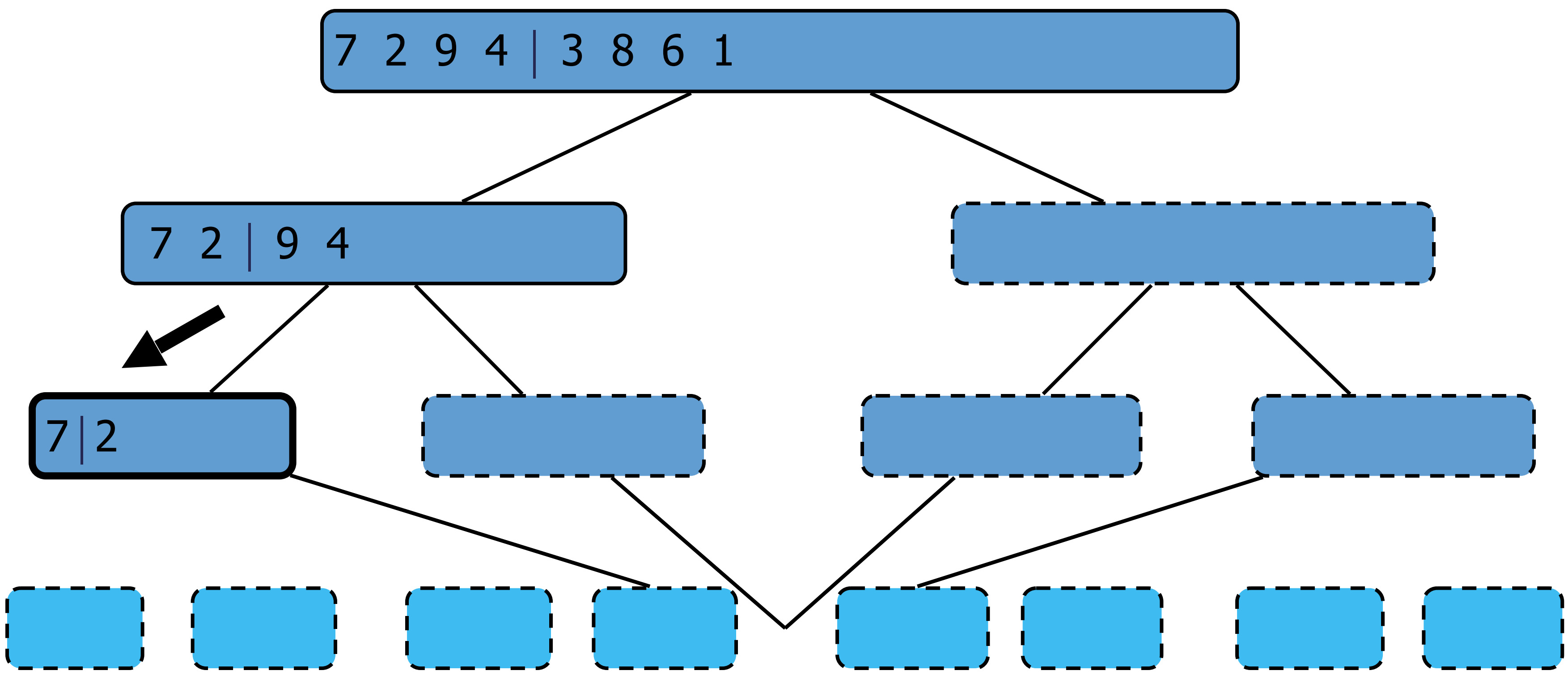
Merge Sort Execution Example

- Recursive call, partition



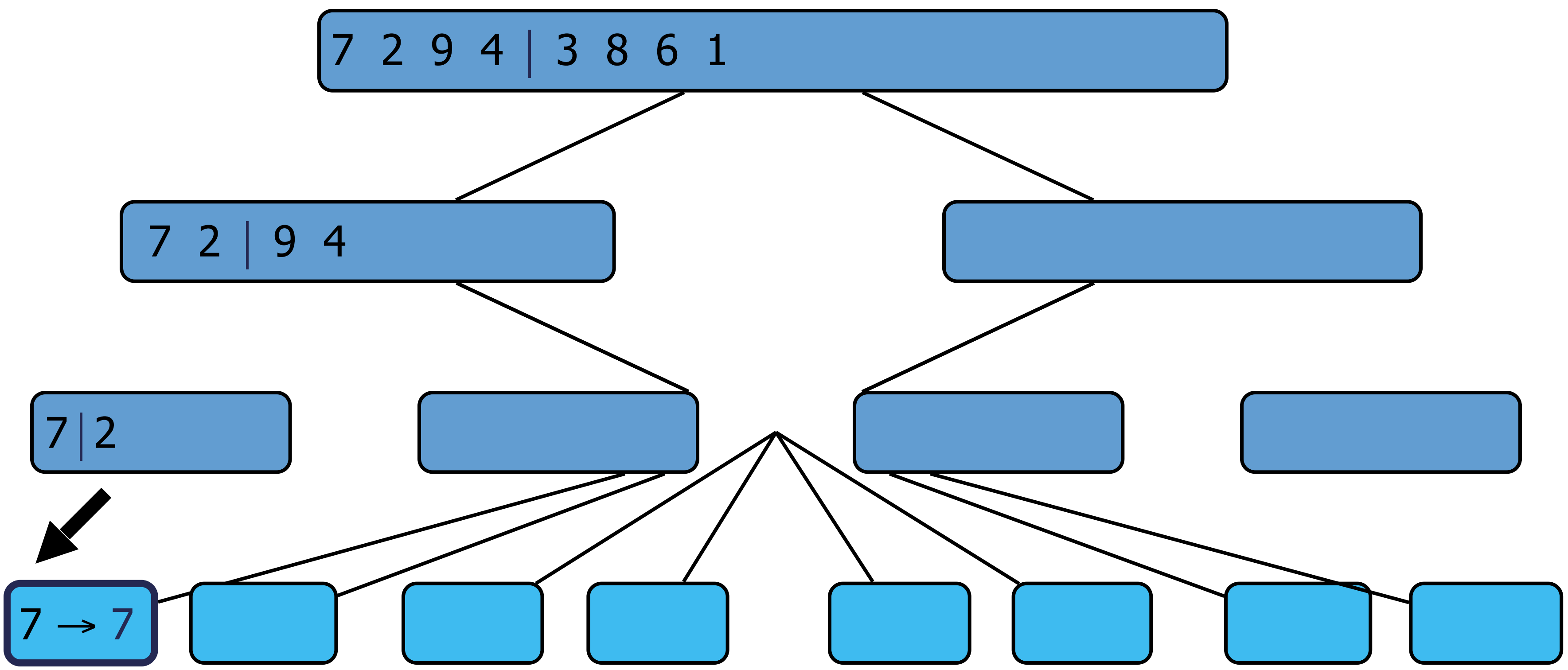
Merge Sort
Execution Example

- Recursive call, partition



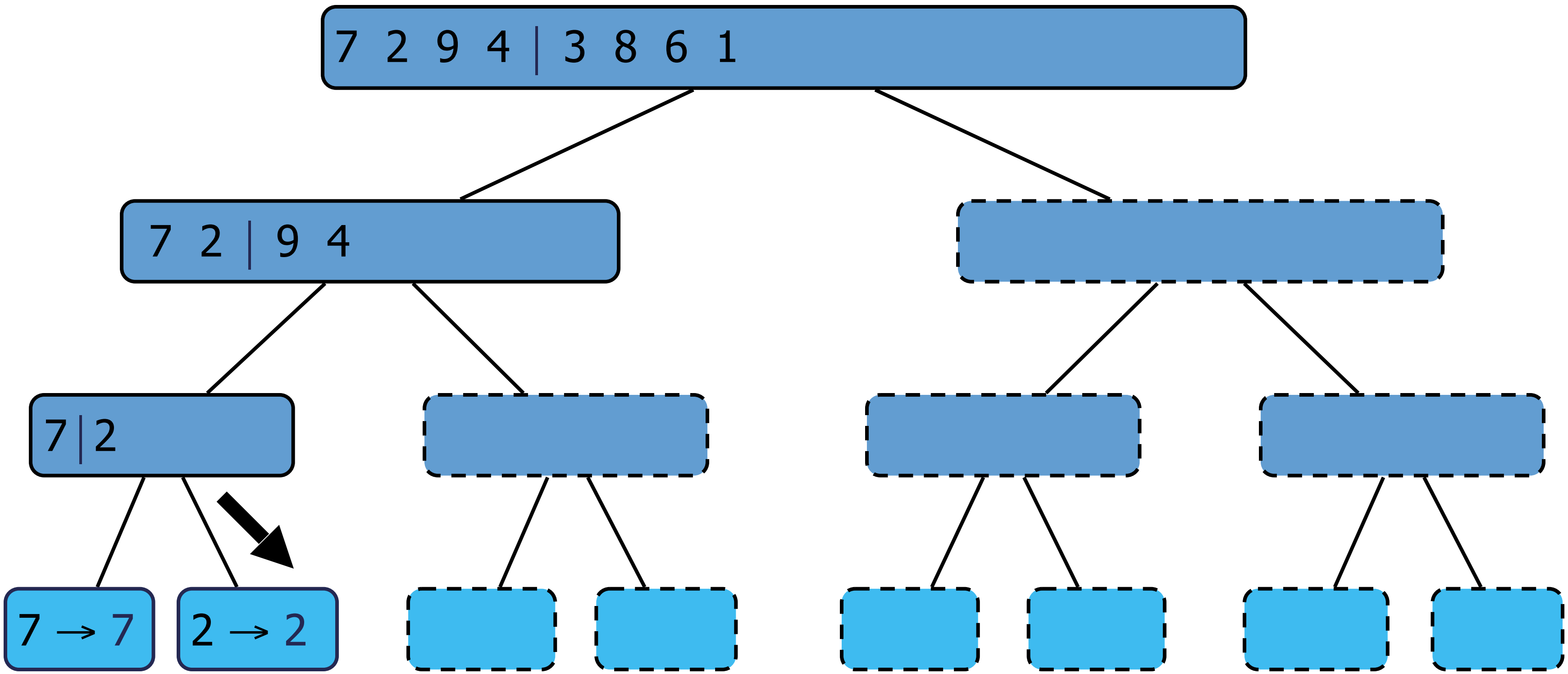
Merge Sort Execution Example

- Recursive call, base case



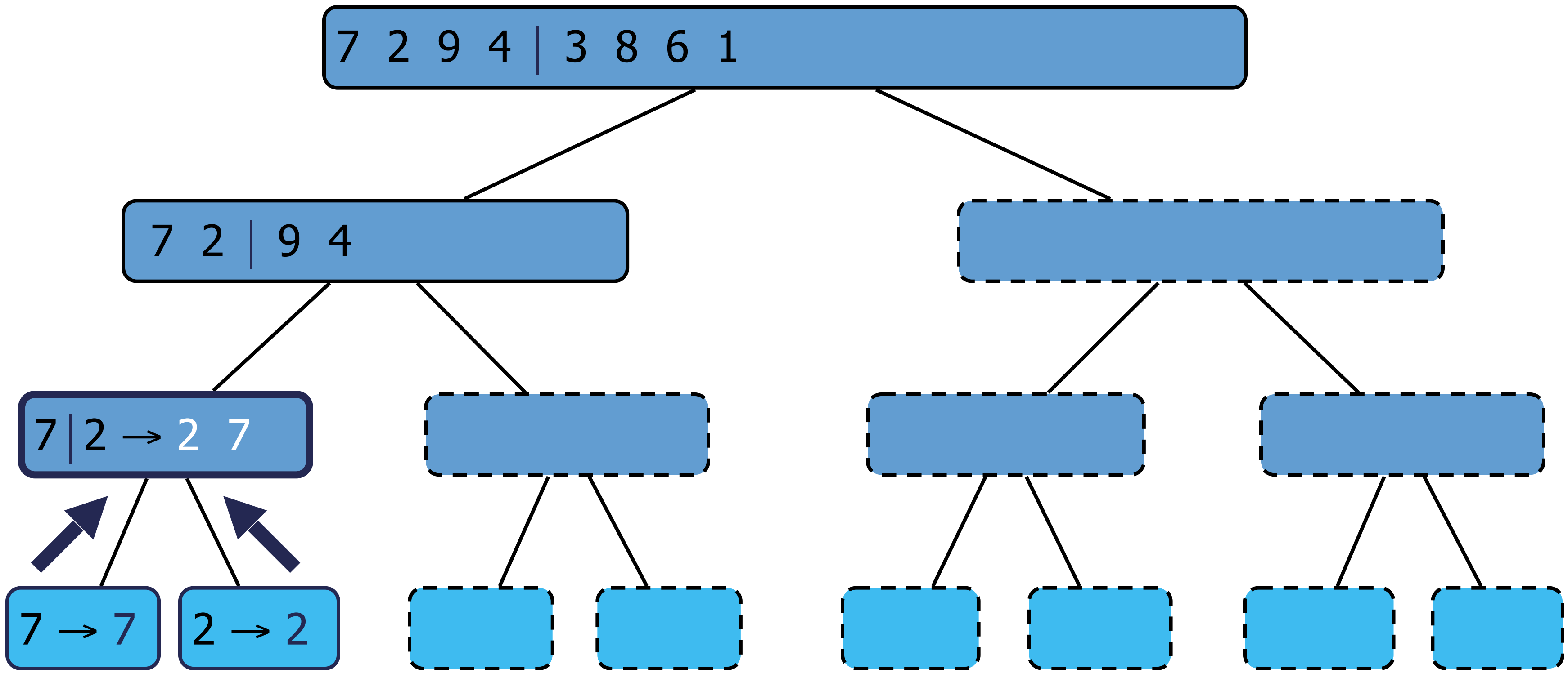
Merge Sort Execution Example

- Recursive call, base case



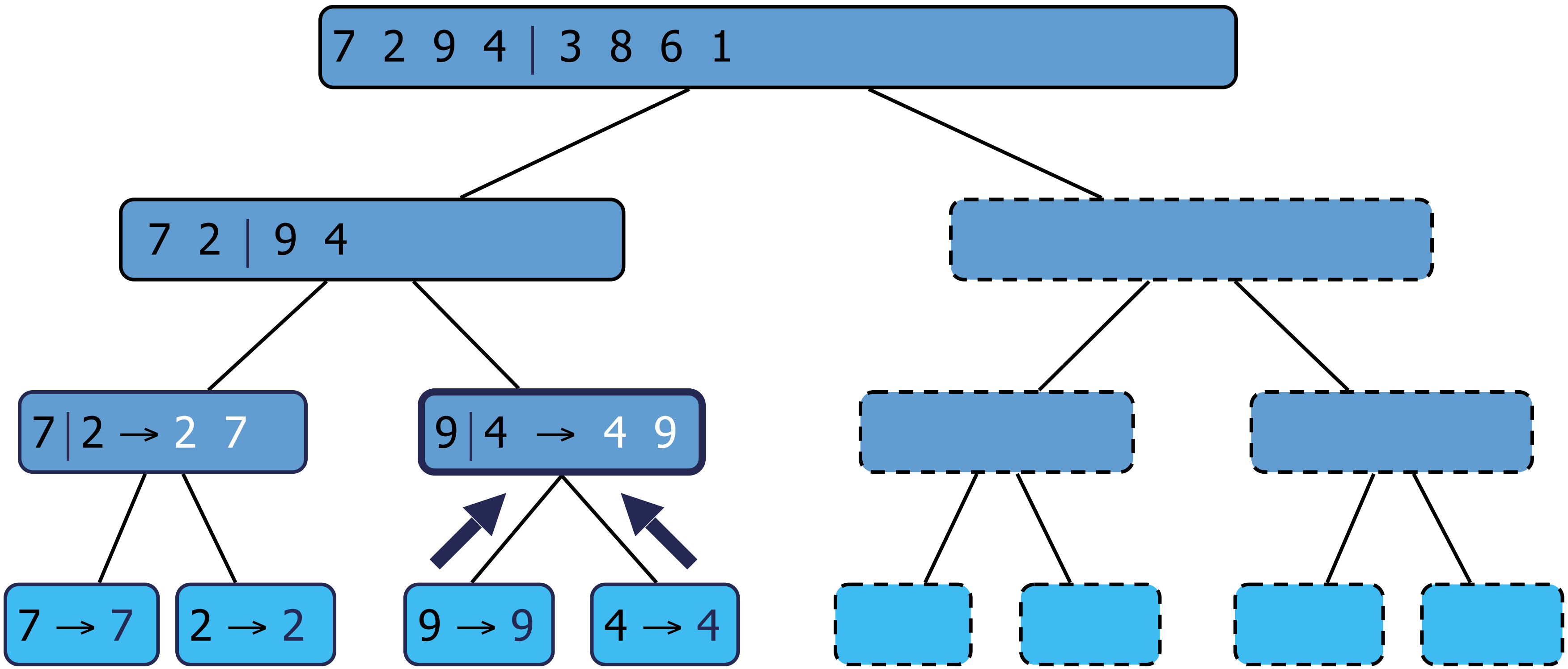
Merge Sort Execution Example

- Merge



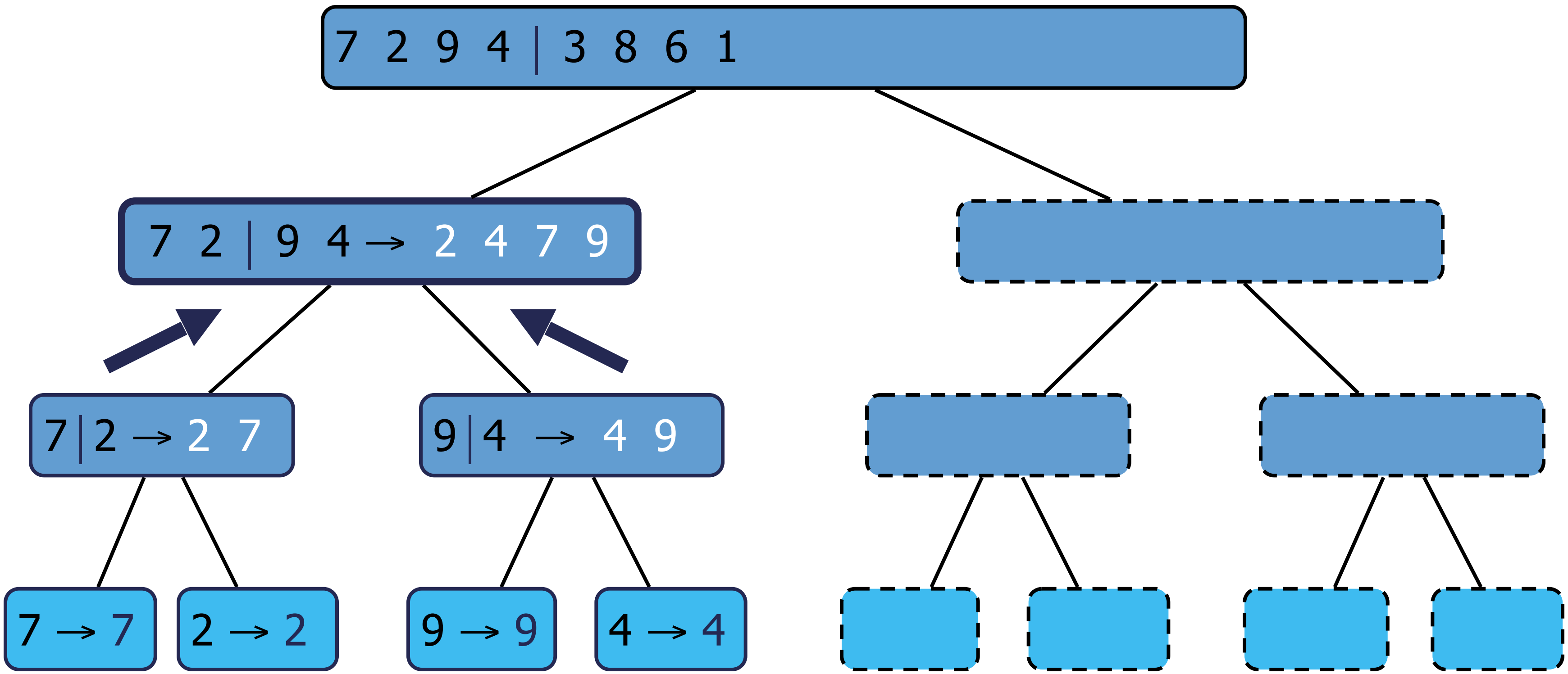
Merge Sort Execution Example

- Recursive call, …, base case, merge



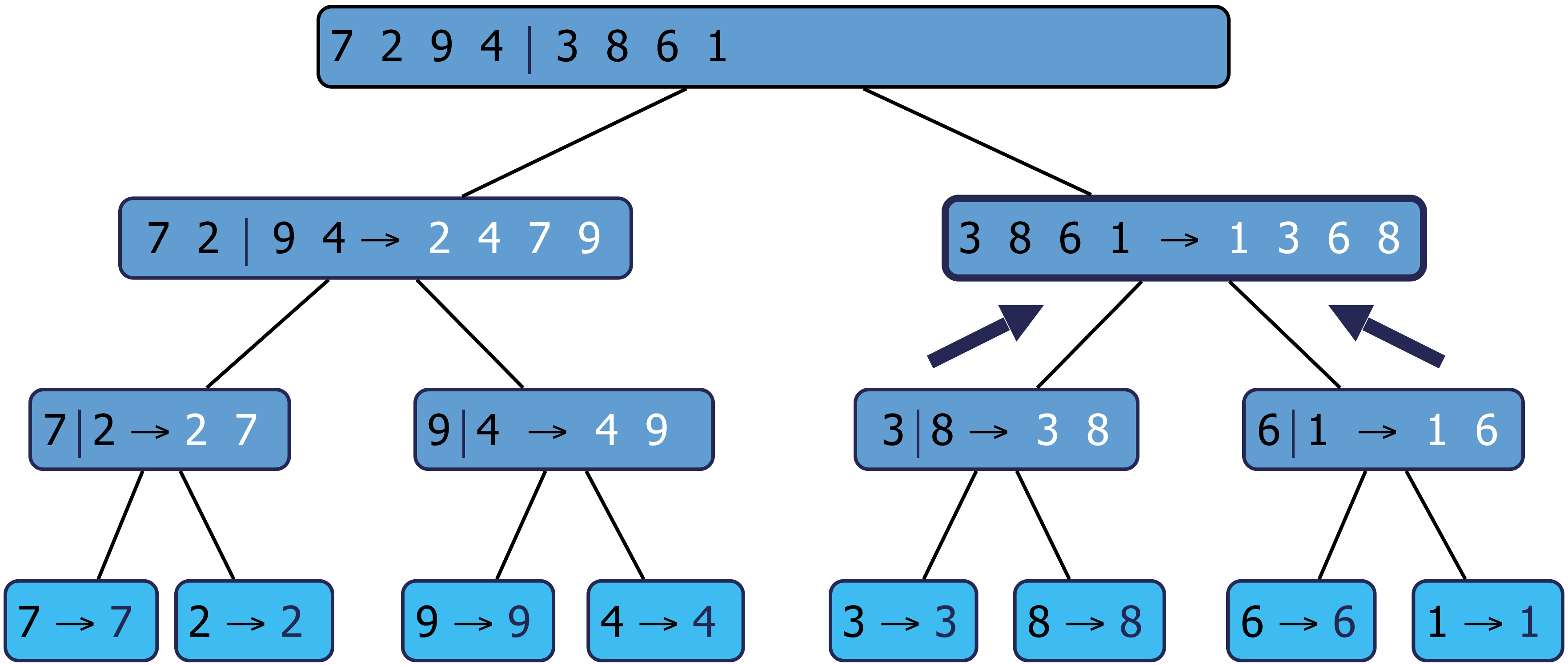
Merge Sort Execution Example

- Merge



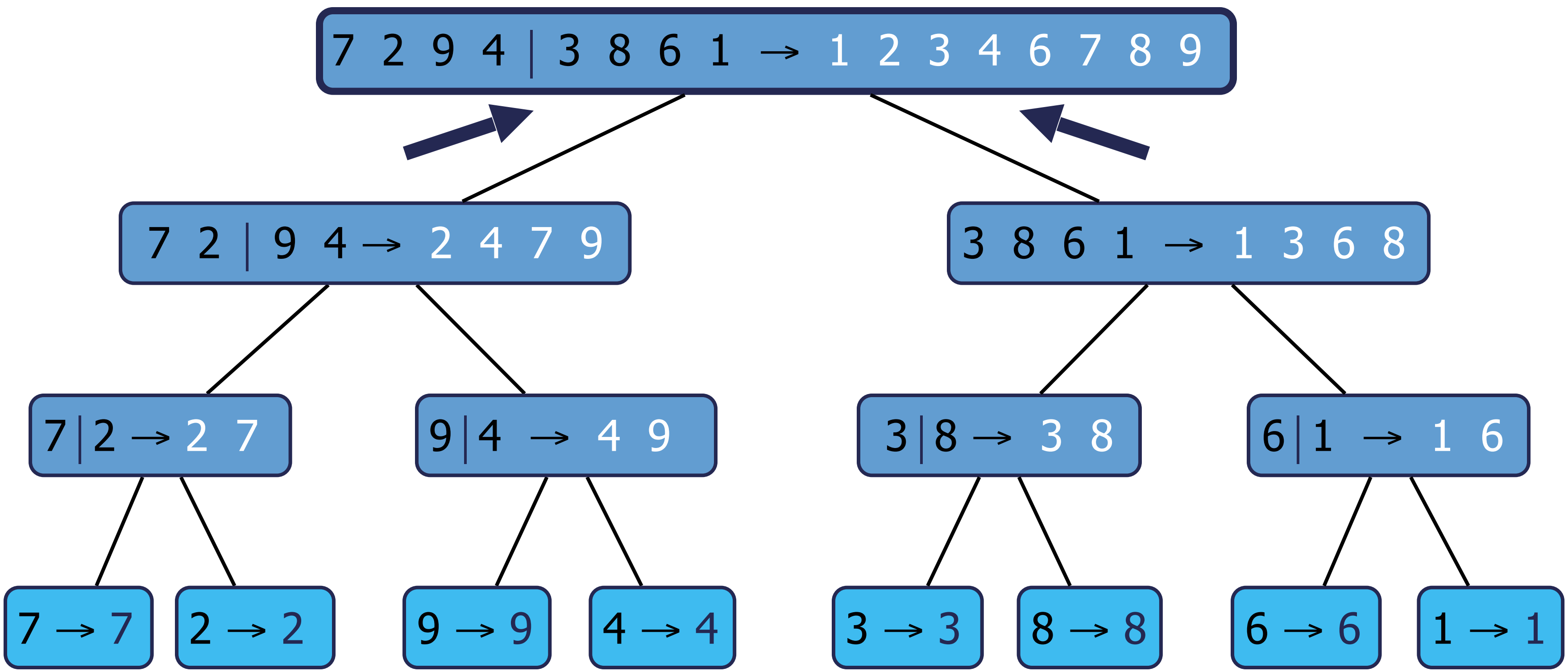
Merge Sort Execution Example

- Recursive call, …, merge, merge

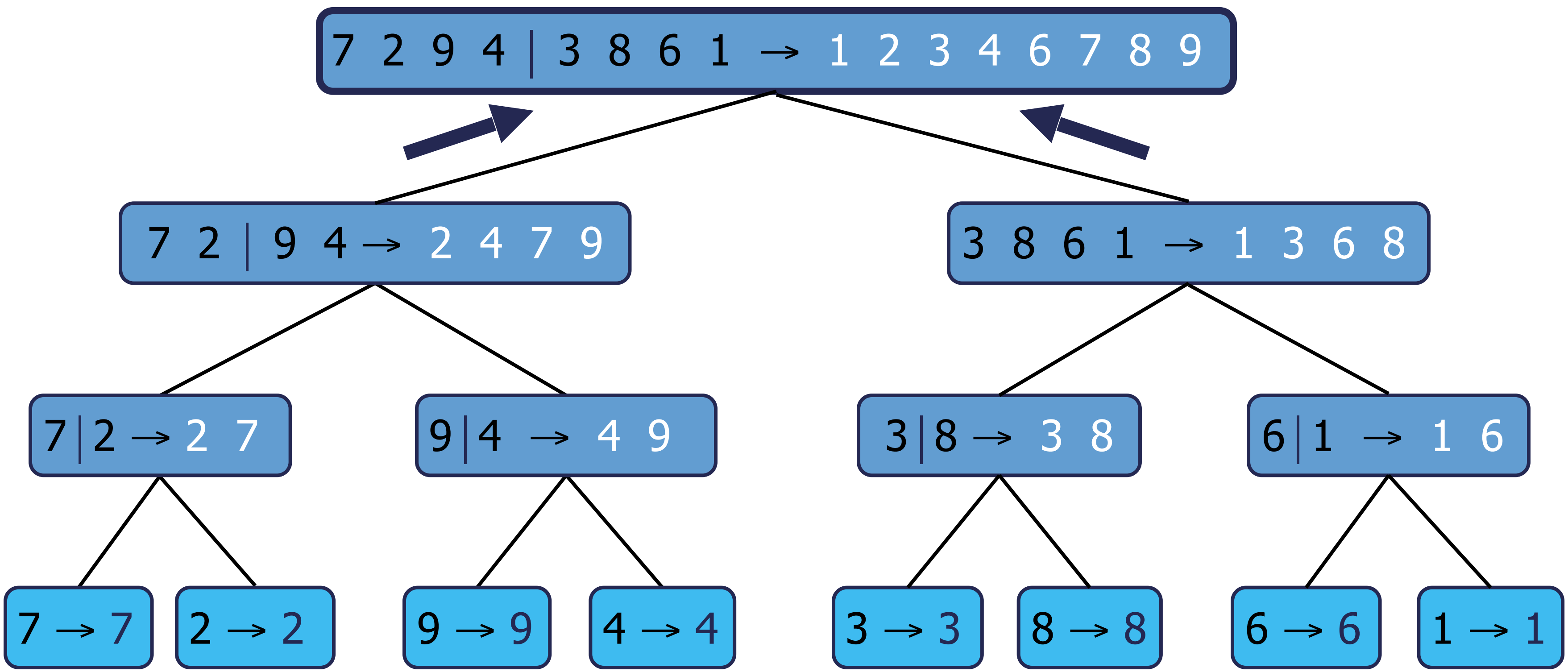


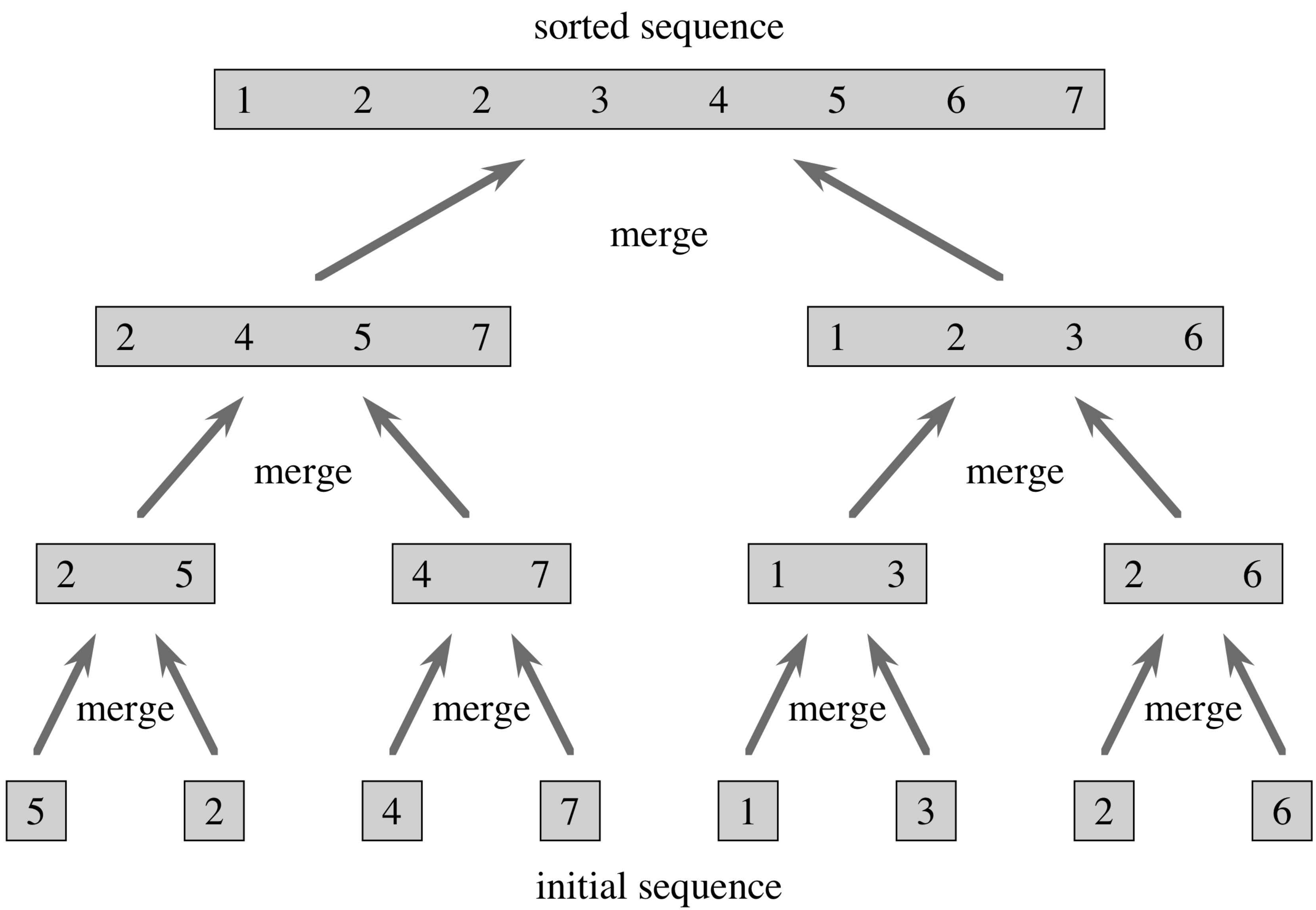
Merge Sort Execution Example

- Merge



Merge Sort

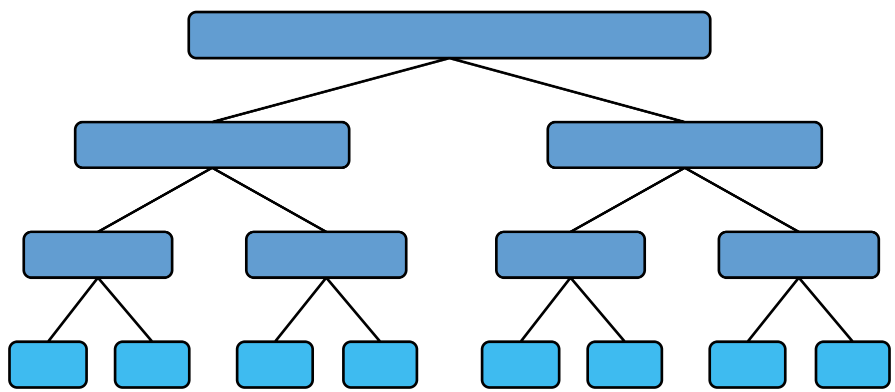

7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9
7 2 | 9 4 → 2 4 7 9
3 8 6 1 → 1 3 6 8
7 | 2 → 2 7
9 | 4 → 4 9
3 | 8 → 3 8
6 | 1 → 1 6
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Analysis of Merge Sort

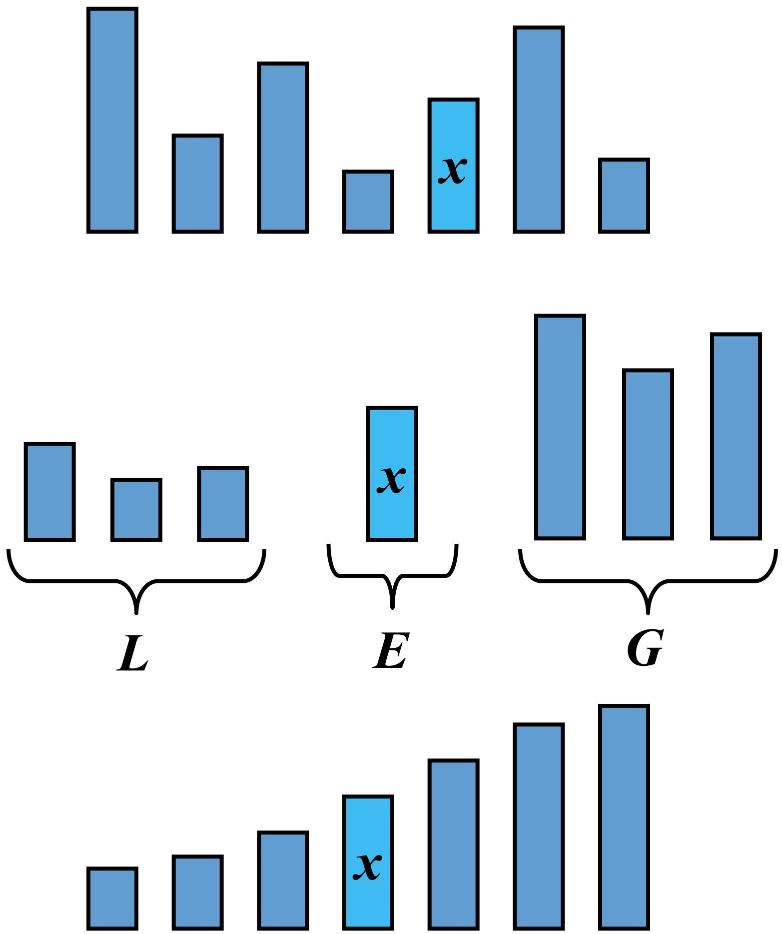
- The height h of the merge sort tree *is* $O(\log n)$
  - Each recursive merge sort call divides sequence by 2
- The overall amount of work done at the nodes of depth i is $O(n)$
  - Partition and merge 2^i sequences of size $n/2^i$
  - 2^{i+1} recursive calls
- Thus, the total running time of merge sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
…	…	…
i	2^i	$n/2^i$



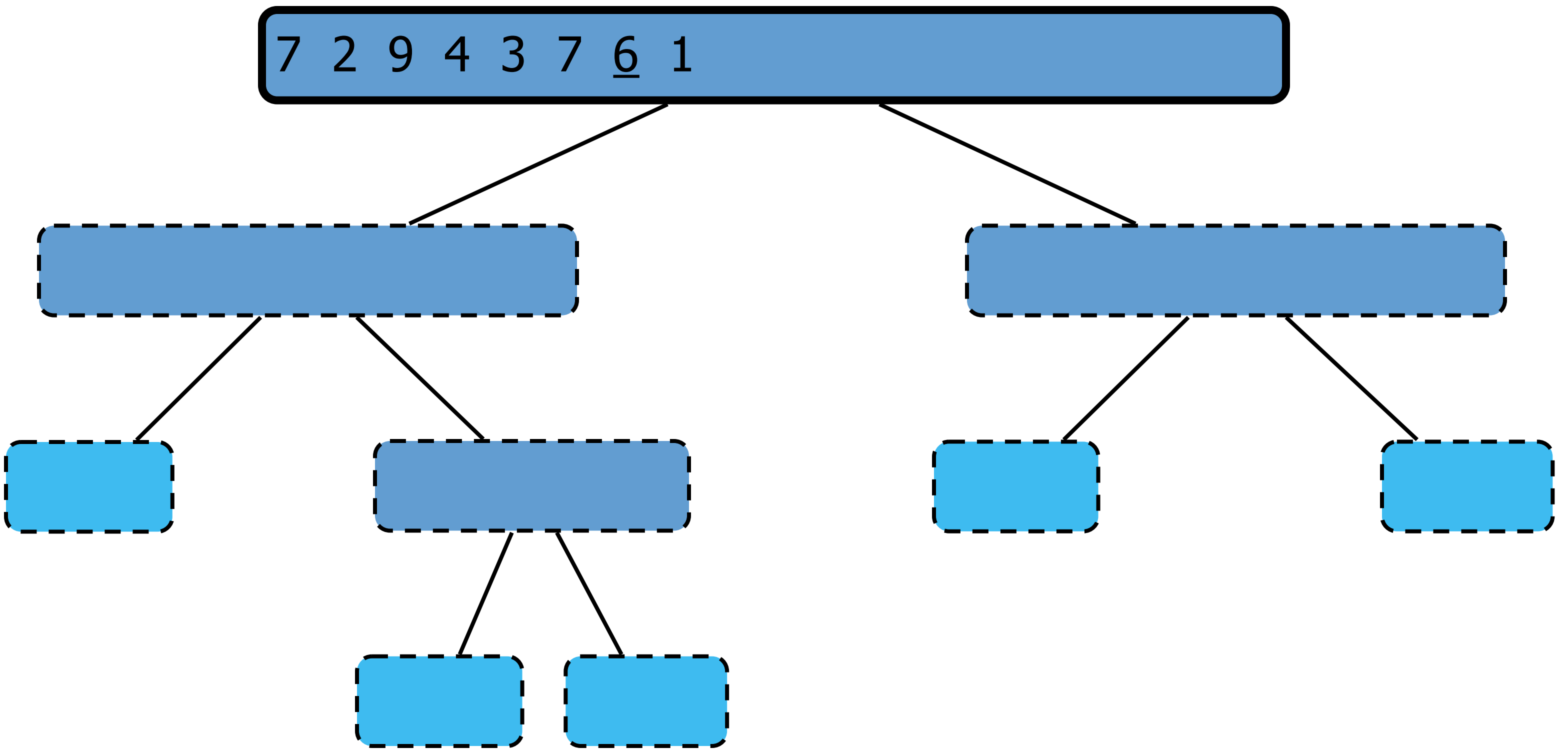
Quicksort

- Quicksort - algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Conquer: sort L and G
  - Combine: join L, E and G



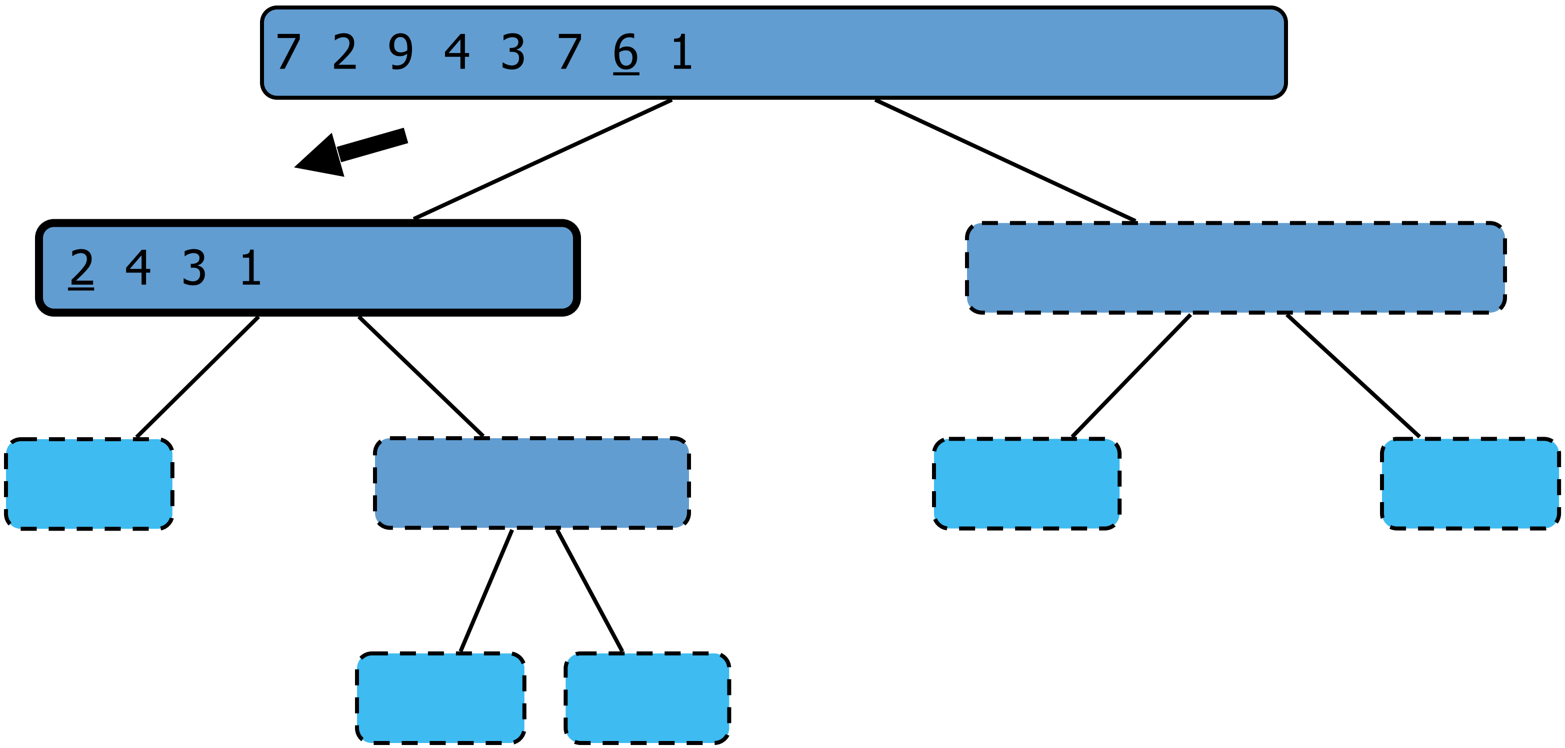
Quicksort Execution Example

- Pivot selection



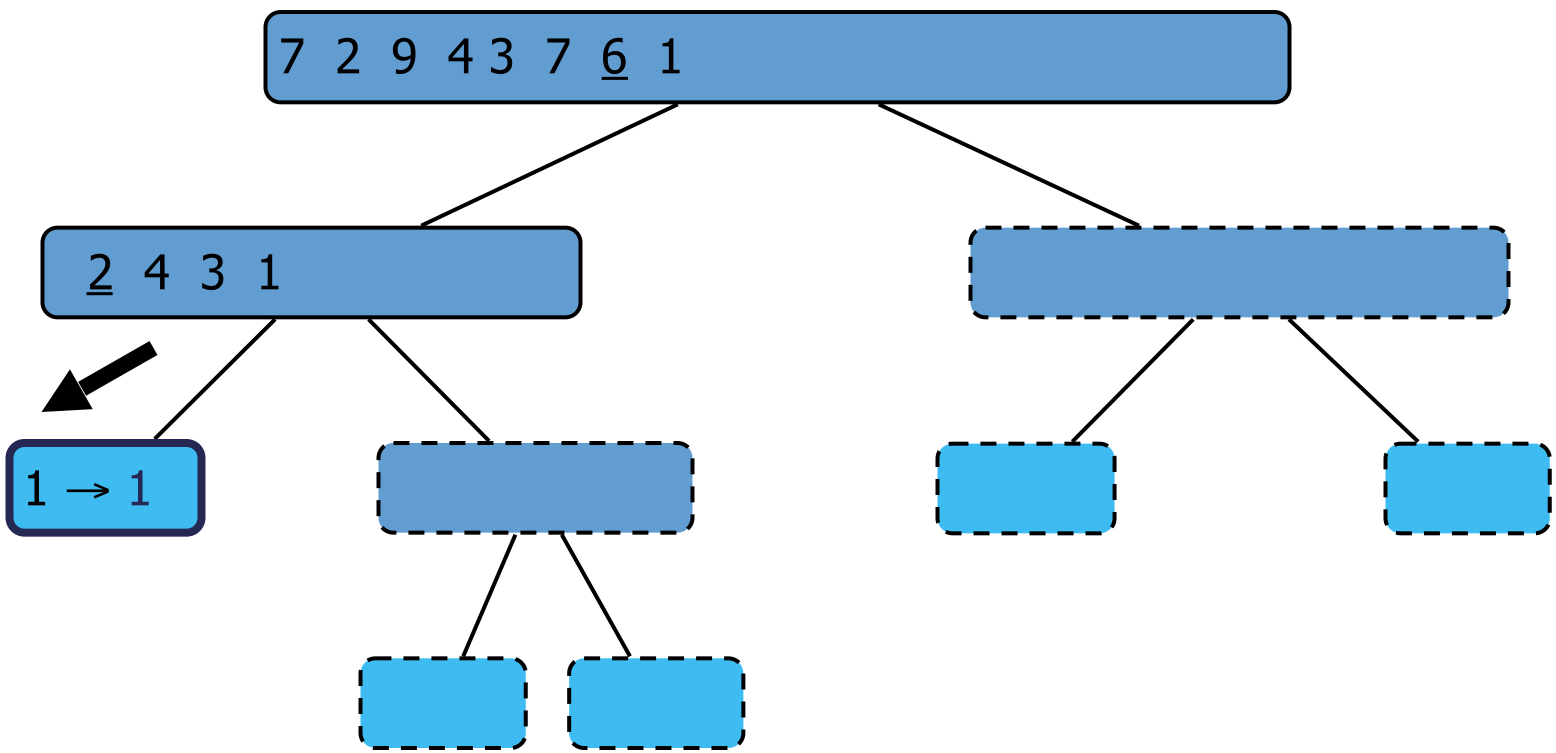
Quicksort Execution Example

- Partition, recursive call, pivot selection



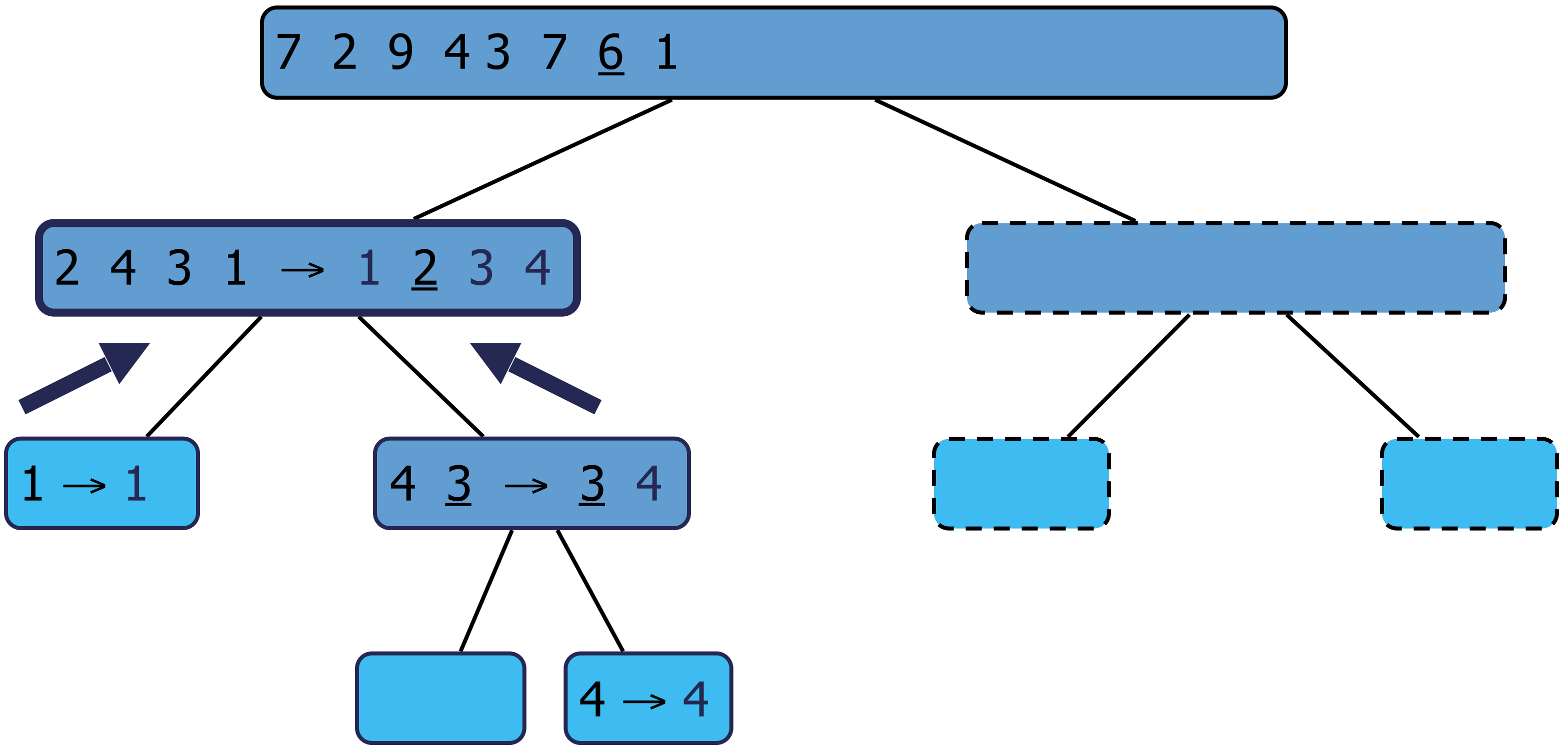
Quicksort Execution Example

- Partition, recursive call, base case



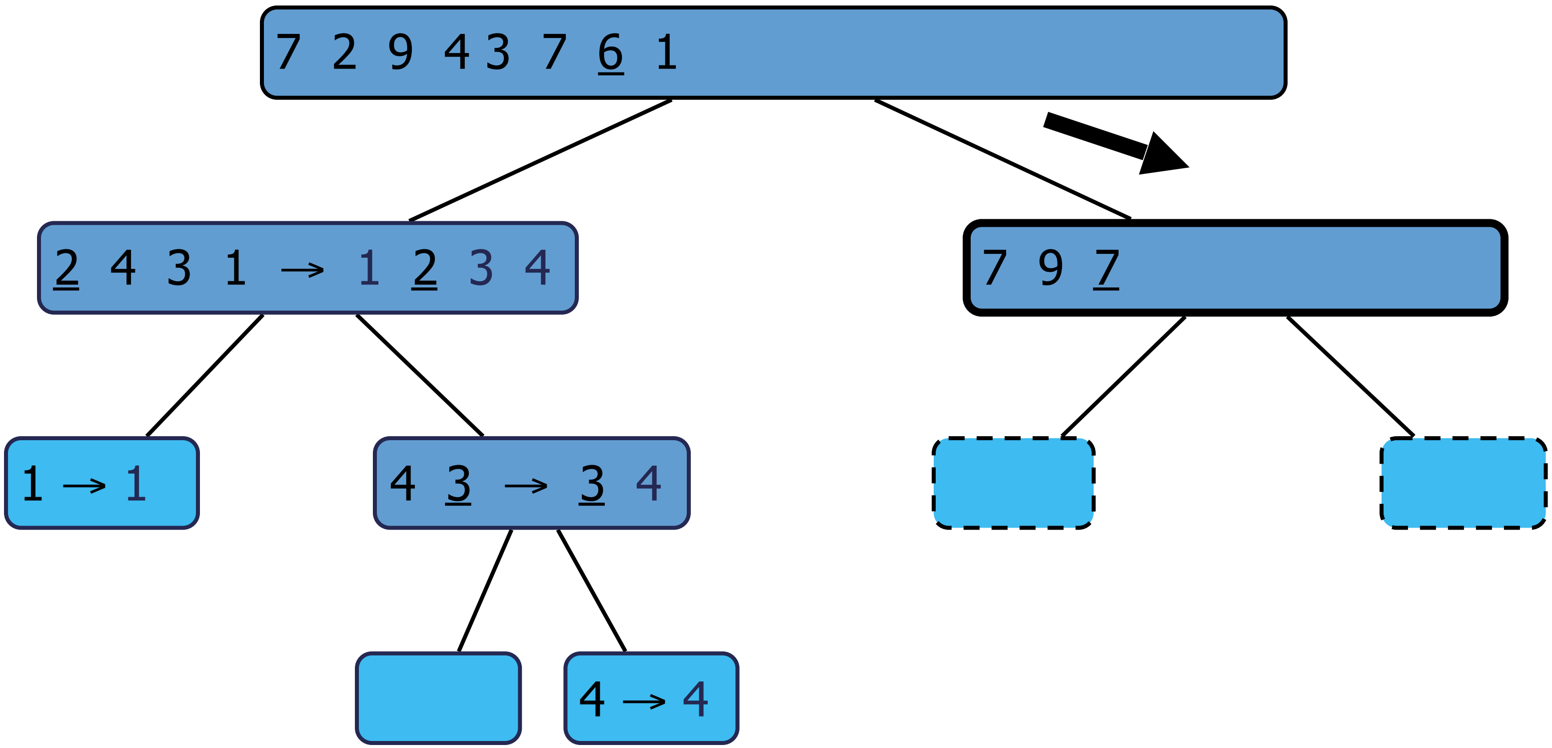
Quicksort Execution Example

- Recursive call, …, base case, join



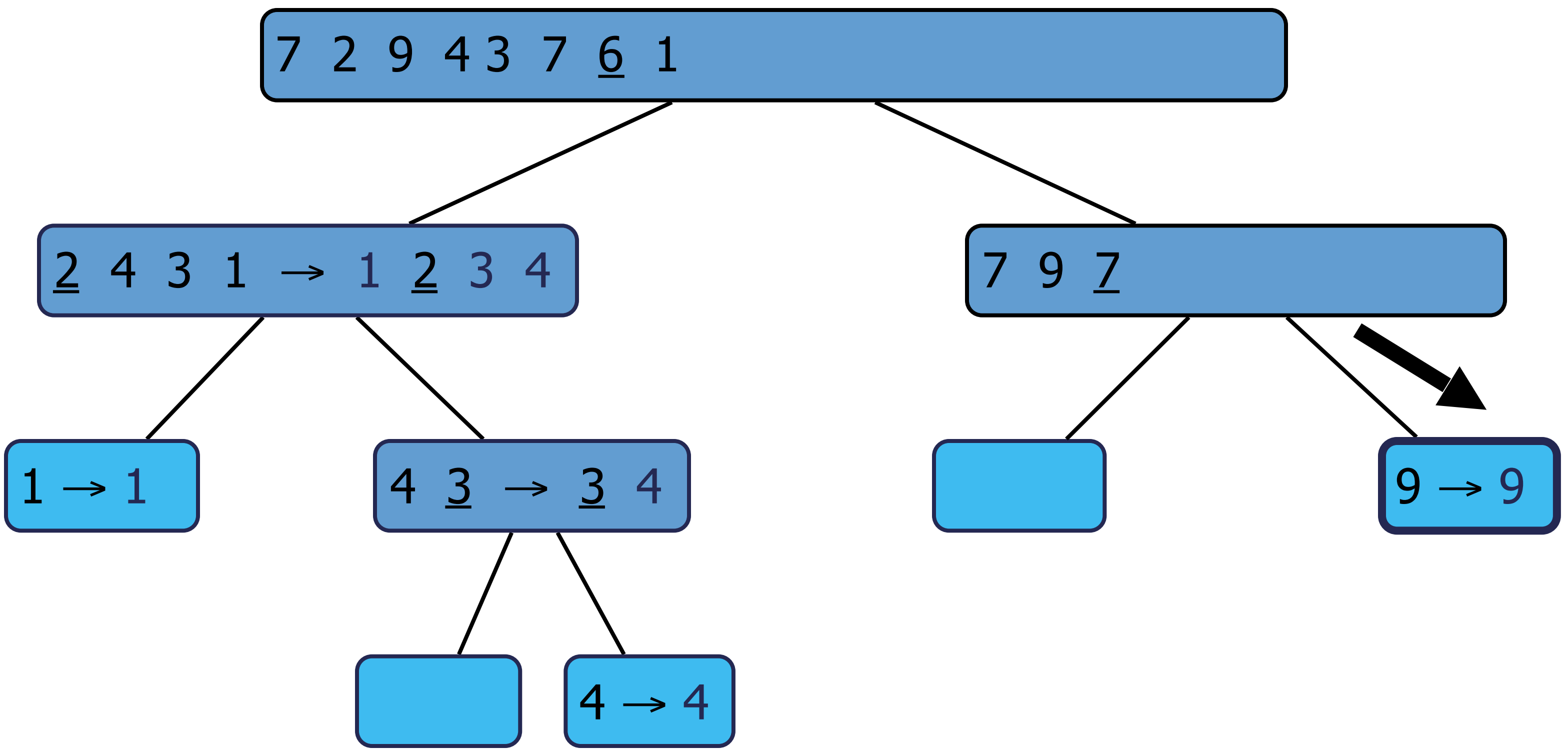
Quicksort Execution Example

- Recursive call, pivot selection



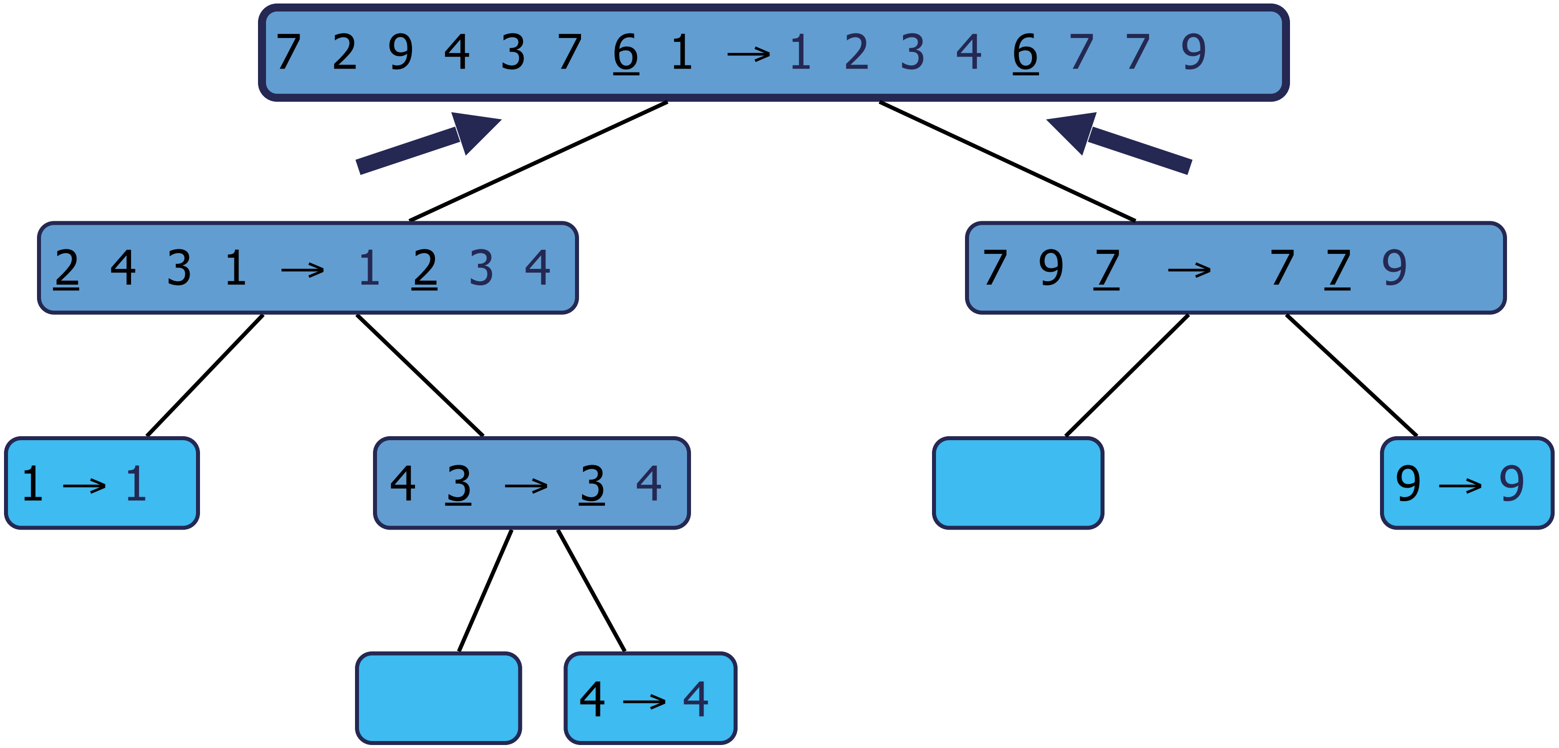
Quicksort Execution Example

- Partition, …, recursive call, base case



Quicksort Execution Example

- Join, join



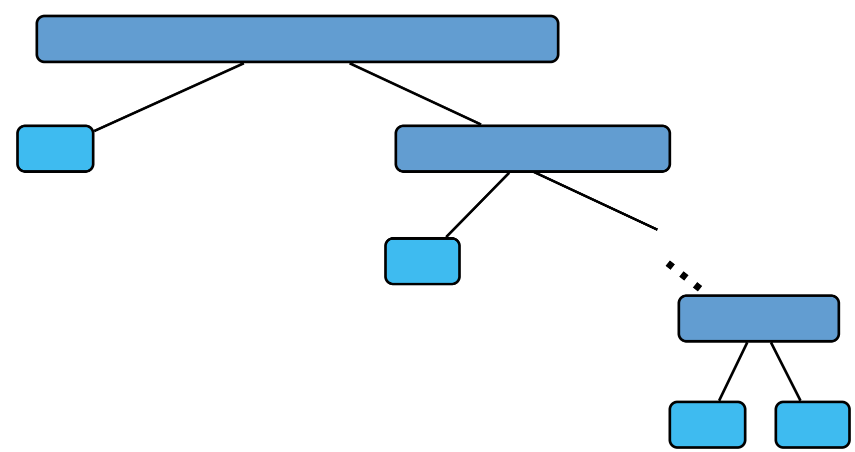
Worst-case Running Time

- The worst case for quicksort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

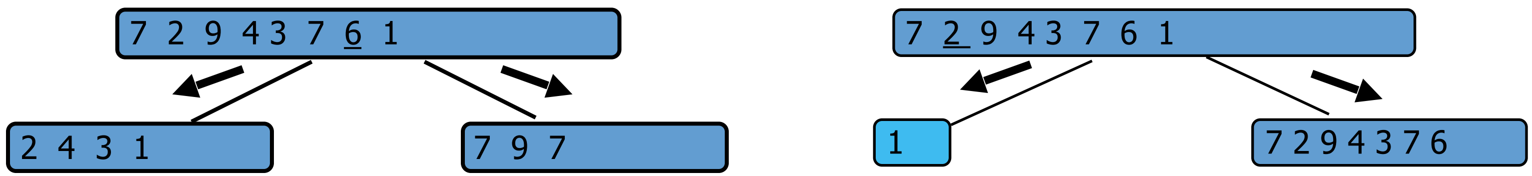
- Thus, the worst-case running time of quicksort is $O(n^2)$

depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1

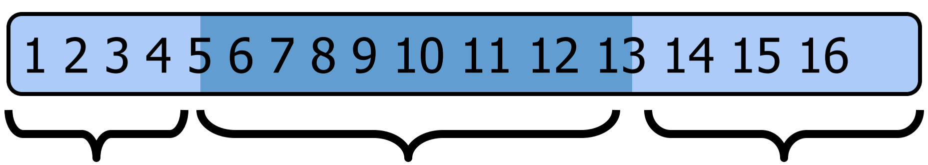


Expected Running Time

- Consider a recursive call of quicksort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Good call **Bad call**



Bad pivots **Good pivots** **Bad pivots**

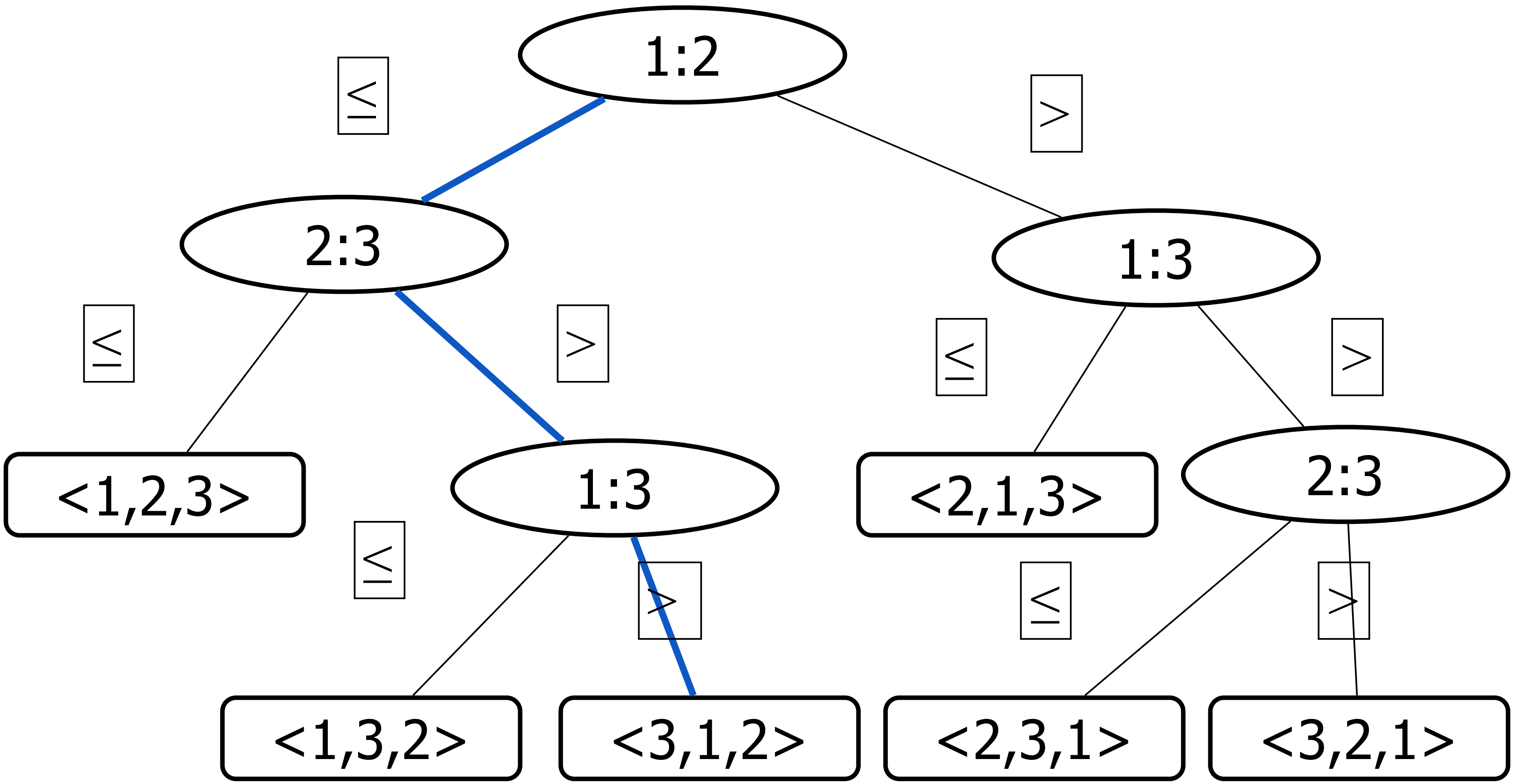
Summary of Sorting Algorithms

Algorithm	Time	Notes
Selection sort	$\boldsymbol{O}(\boldsymbol{n}^2)$	in-place slow (good for small inputs)
Insertion sort	$\boldsymbol{O}(\boldsymbol{n}^2)$	in-place slow (good for small inputs)
Quicksort	$\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$ expected	in-place, randomized fastest (good for large inputs)
Heapsort	$\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$	in-place fast (good for large inputs)
Mergesort	$\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$	sequential data access fast (good for huge inputs)

Lower Limit on Worst Case Time Bound on Sorting

- Comparison Sorts – the sorted order is determined based only on comparisons between the input elements.
- It turns out that we can show a lower limit on the worst case time bound on sorting for comparison sorts
- Without loss of generality, we can consider that all of the input elements are distinct. This makes all comparison operations equivalent.

Decision Tree



$a_3 = 5$, $a_1 = 6$, $a_2 = 8$ <3,1,2> = <5,6,8>

Worst Case Bounds

- For n entries, there are n! permutations
- Consider a decision tree with height h and L reachable leaves
- A tree of height h has no more than 2^h nodes
- Therefore: $n! \le L \le 2^h$
- $h \geq lg(n!)$
- $h = \Omega(n\ lg\ n)$ (use Stirling's approximation,

  equation 3.19, p. 58 in CLRS)

Radix Sort

64, 8, 217, 512, 27, 728, 40, 1, 343, 125, 313

Pass	0	1	2	3	4	5	6	7	8	9
P=1										
P=2										
P=3										

Radix Sort

64, 8, 217, 512, 27, 728, 40, 1, 343, 125, 313

Pass	0	1	2	3	4	5	6	7	8	9
P=1	40	1	512	343 313	64	125		217 27	8 728	
P=2	1 8	512 313 217	125 27 728		40 343		64			
P=3	1 8 27 40 64	125	217	313 343		512		728		

Running time - *O(kN)* where k is digit length

Radix Sort

- Works on integers with d-digits
- Idea:
  - Sort (bin) integers into bins based upon least significant digit
  - Resort based upon next least significant
  - Continue until most significant digit
- O(n) sort

Radix Sort

123

231

534

223

979

311

421

Radix Sort

231
311
421
123
223
534
979

Radix Sort

311

421

123

223

231

534

979

Radix Sort

123

223

231

311

421

534

979

Sorting Stability

- Duplicate numbers appear in the same order in the output as they did in the input.
    - Think about the sorting algorithms - which ones are stable?
    - Swaps introduce the chance for re-ordering.