CS121 Data Structures
Efficient Sorting

Monika Stepanyan
mstepanyan@aua.am

Spring 2024

Sorting in N-Log-N And Linear Time

Sorting problem: start with an unordered array of elements and rearrange them into nondecreasing order

- Merge Sort
- Quick Sort
- Bucket Sort
- Radix Sort

Heap Sort will be introduced later when we discuss heaps.

Divide and Conquer

Divide and conquer is an algorithmic design pattern that consists of the following three steps:

1. ***Divide:*** divide the input data S into two or more disjoint subsets $S_1, S_2, \ldots$
2. ***Conquer:*** solve the subproblems recursively
3. ***Combine:*** combine the solutions for $S_1, S_2, \ldots$, into a solution for S

The base case for the recursion are subproblems of constant size

Examples we have seen?

Merge Sort

To sort a sequence S with n elements, merge sort does:

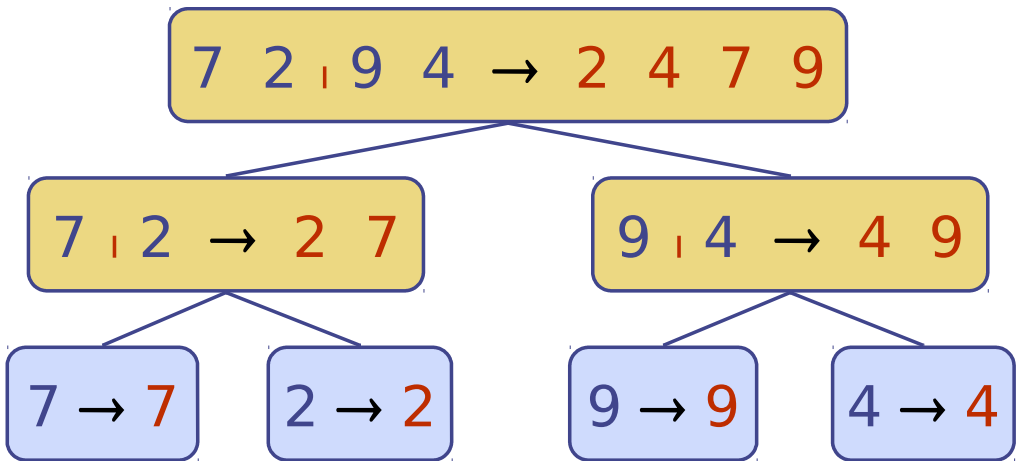
1. ***Divide:*** If $|S| \leq 1$, return S. Otherwise, partition S into S_1 with $\lfloor \frac{n}{2} \rfloor$ elements, and S_2 with $\lceil \frac{n}{2} \rceil$ elements.
2. ***Conquer:*** Recursively sort sequences S_1 and S_2.
3. ***Combine:*** **Merge** the sorted sequences S_1 and S_2 into a unique sorted sequence.

Reminder: $\lfloor 3.14 \rfloor = 3$ and $\lceil 3.14 \rceil = 4$.

Merge Sort Tree

An execution of merge sort is depicted by a binary tree

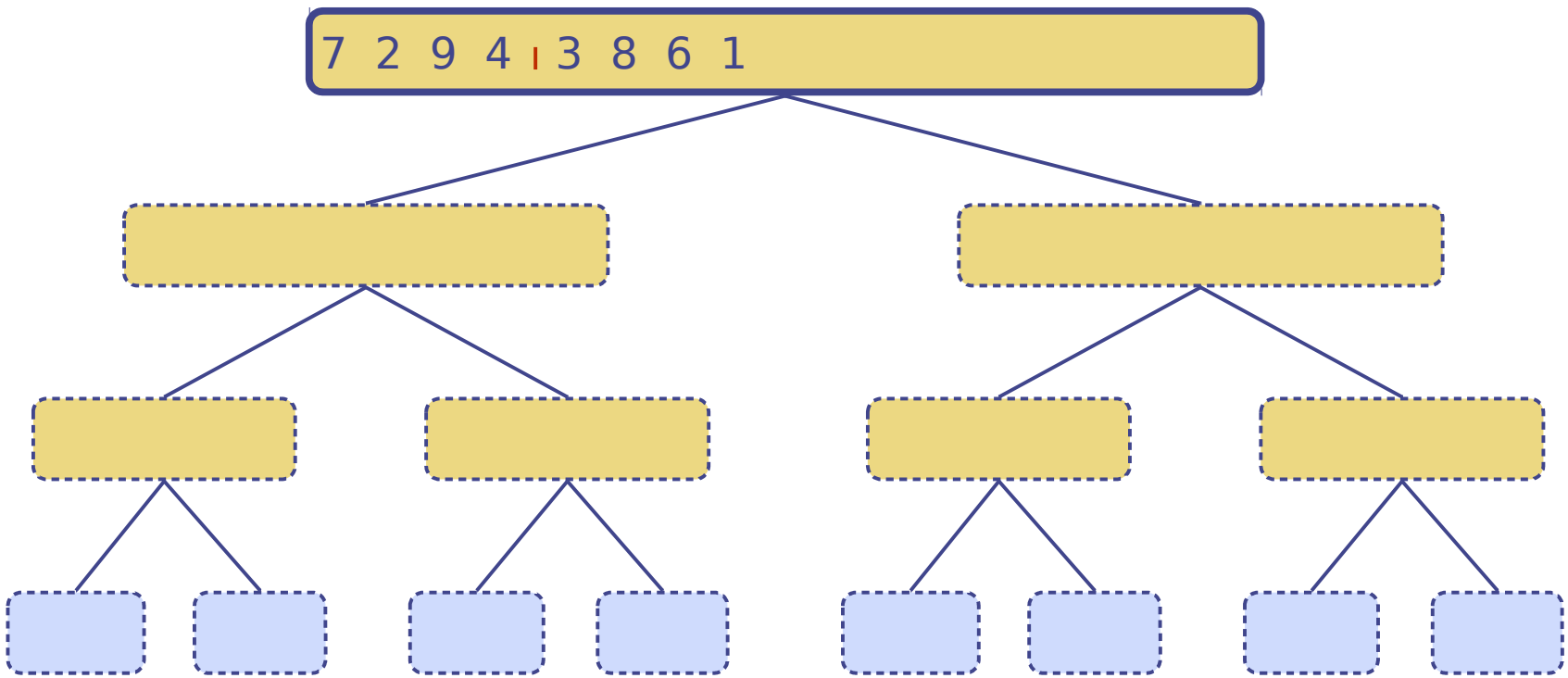
- each node represents a recursive call of merge sort and stores
  - unsorted sequence before the execution and its partition
  - sorted sequence at the end of the execution
- the root is the initial call
- the leaves are calls on subsequences of size 0 or 1



The merge sort tree associated with an execution of merge sort on a sequence of size n has height $\lceil \log n \rceil$

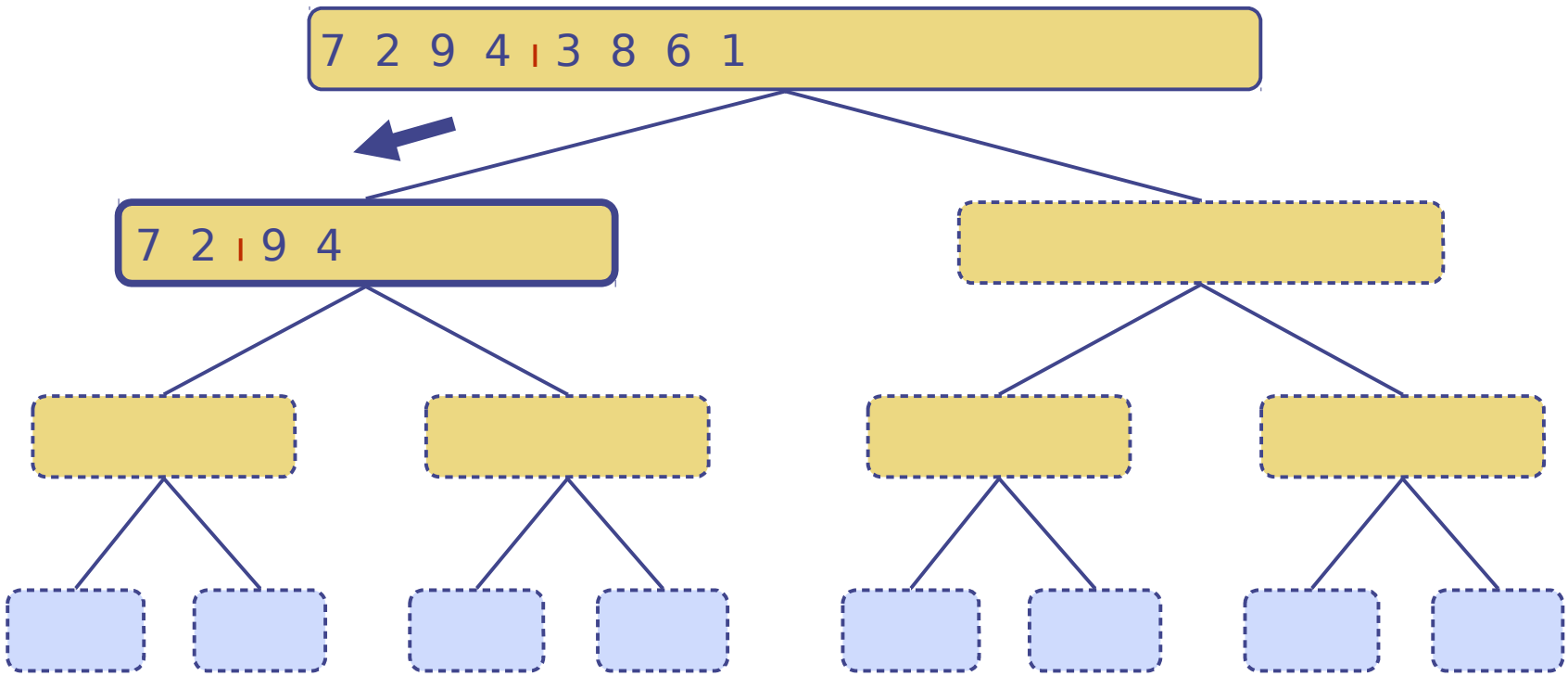
Merge Sort: Illustration I

Partition



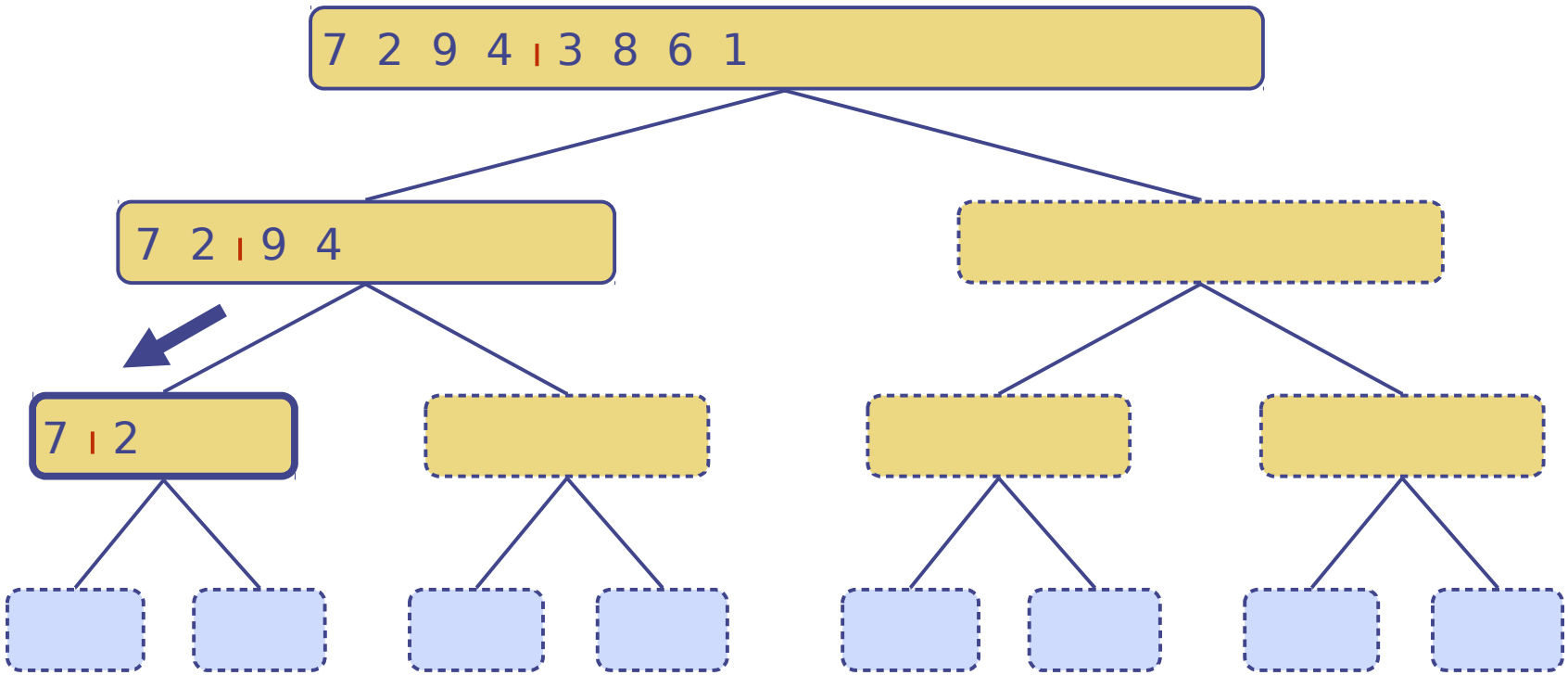
Merge Sort: Illustration II

Recursive call, partition



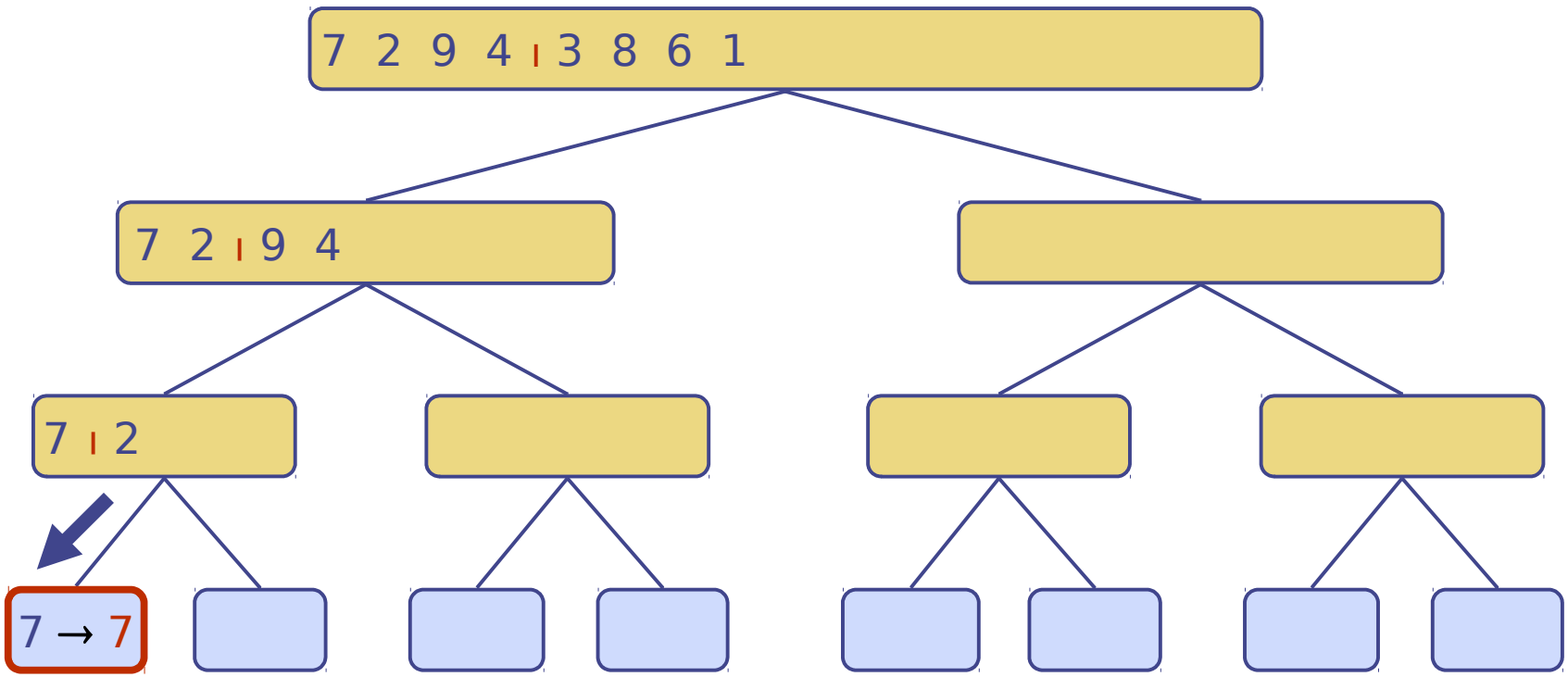
Merge Sort: Illustration III

Recursive call, partition



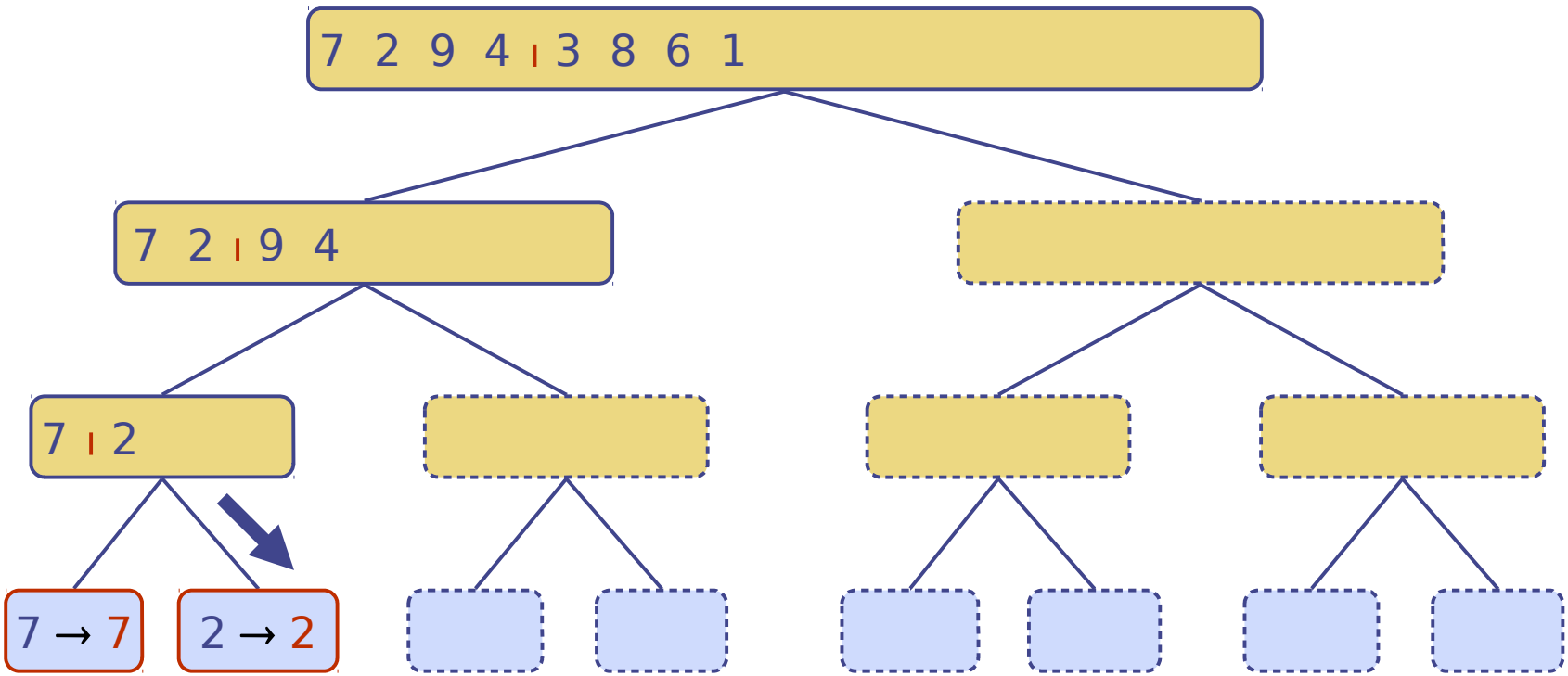
Merge Sort: Illustration IV

Recursive call, base case



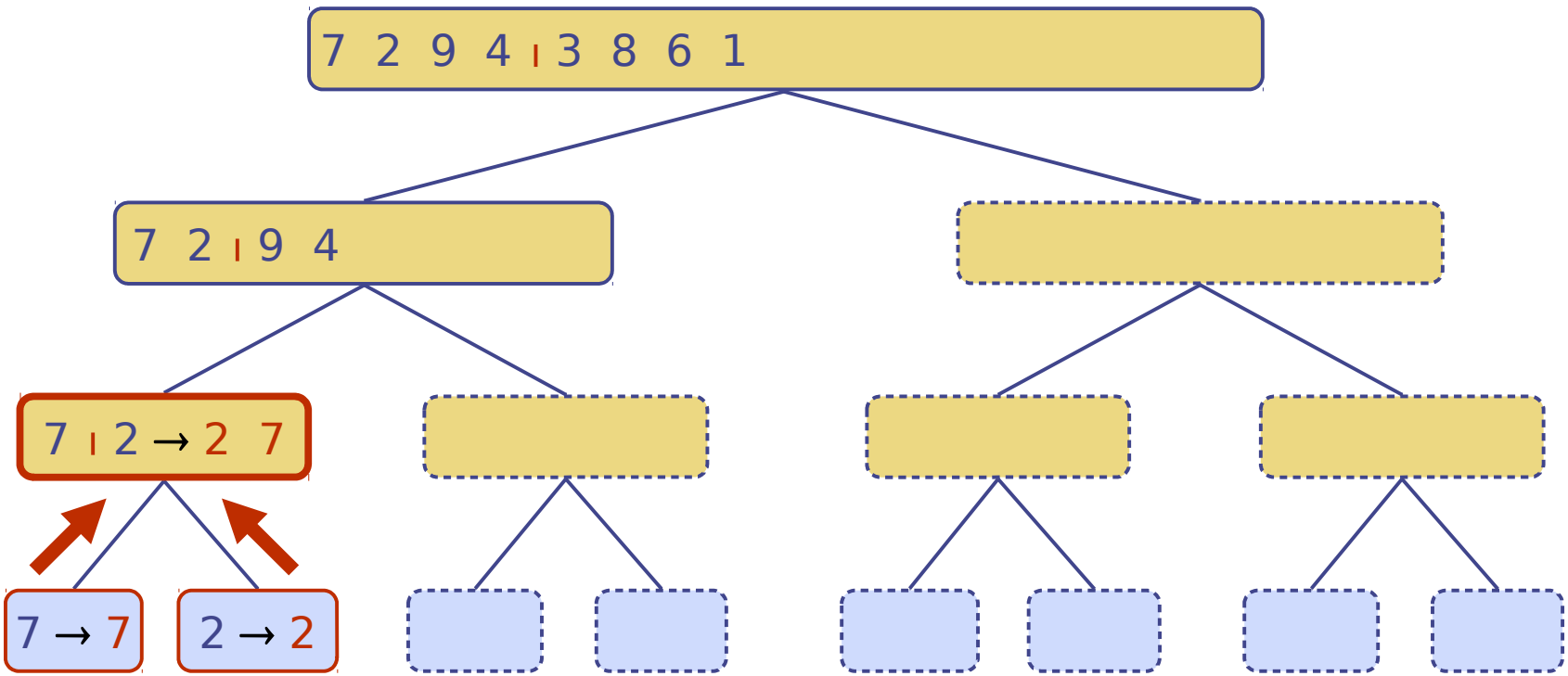
Merge Sort: Illustration V

Recursive call, base case



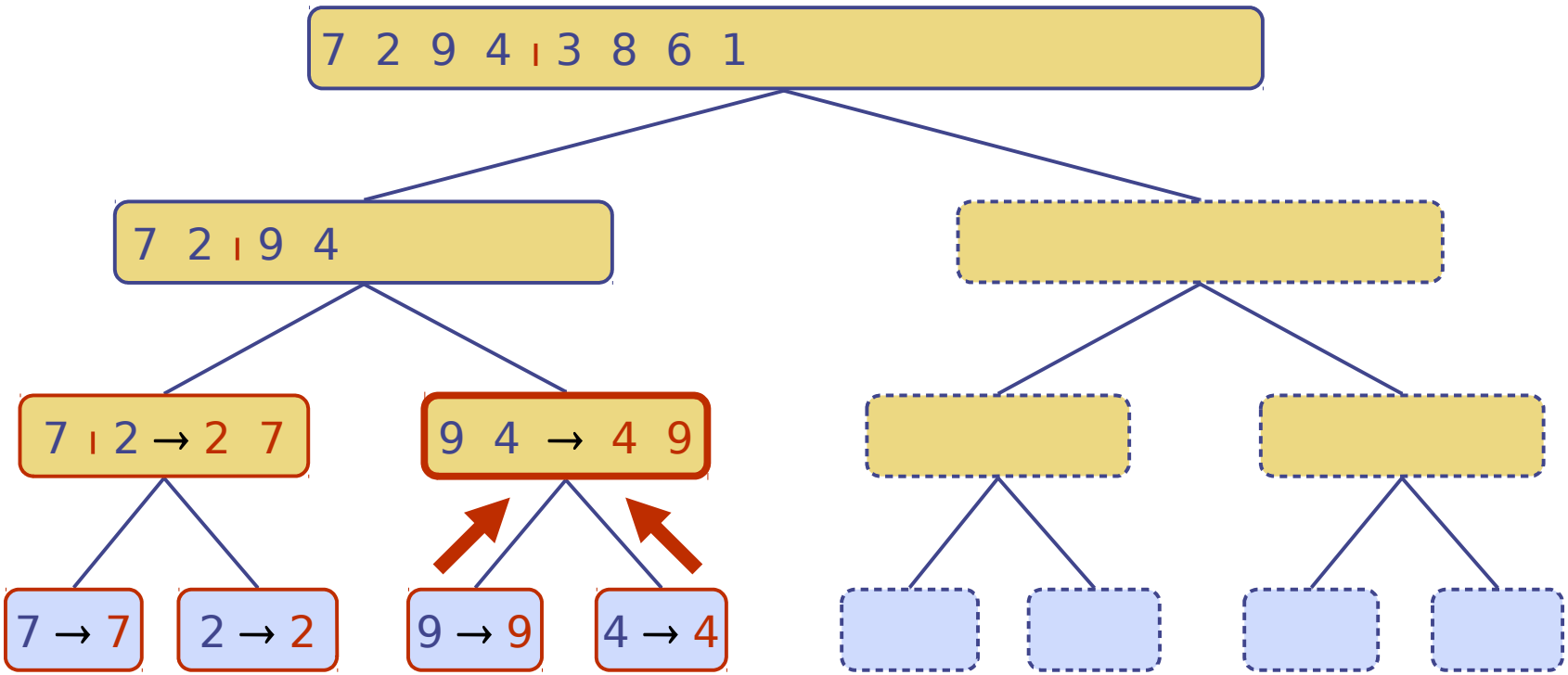
Merge Sort: Illustration VI

Merge



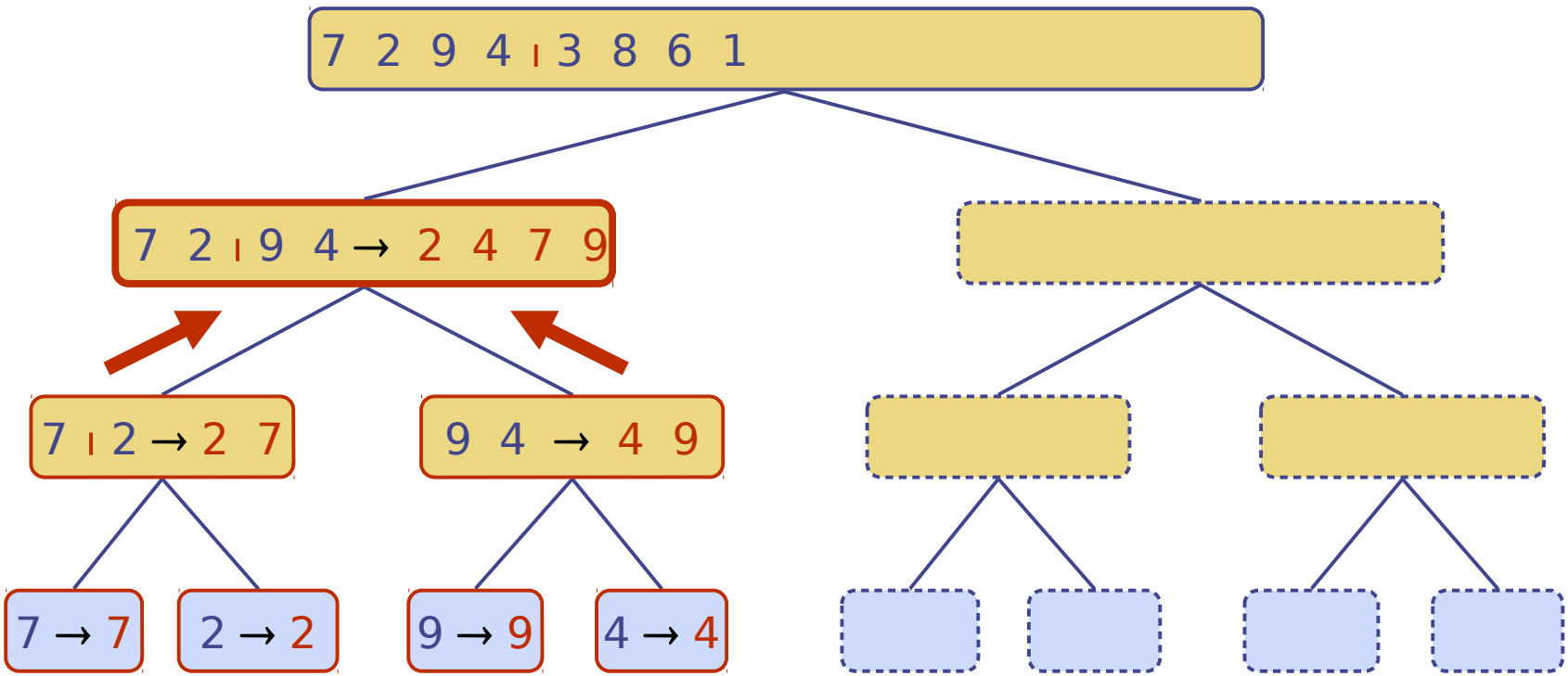
Merge Sort: Illustration VII

Recursive call, ..., base case, merge



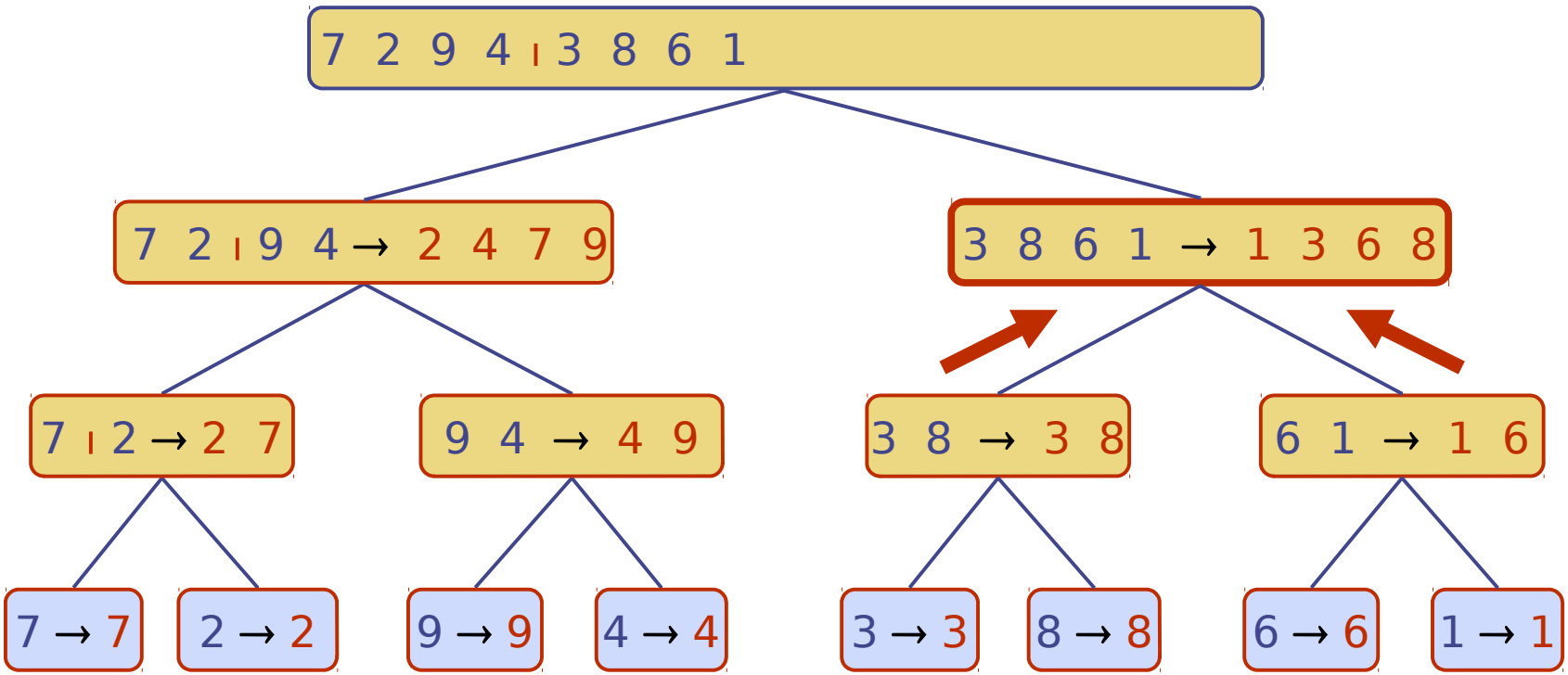
Merge Sort: Illustration VIII

Merge



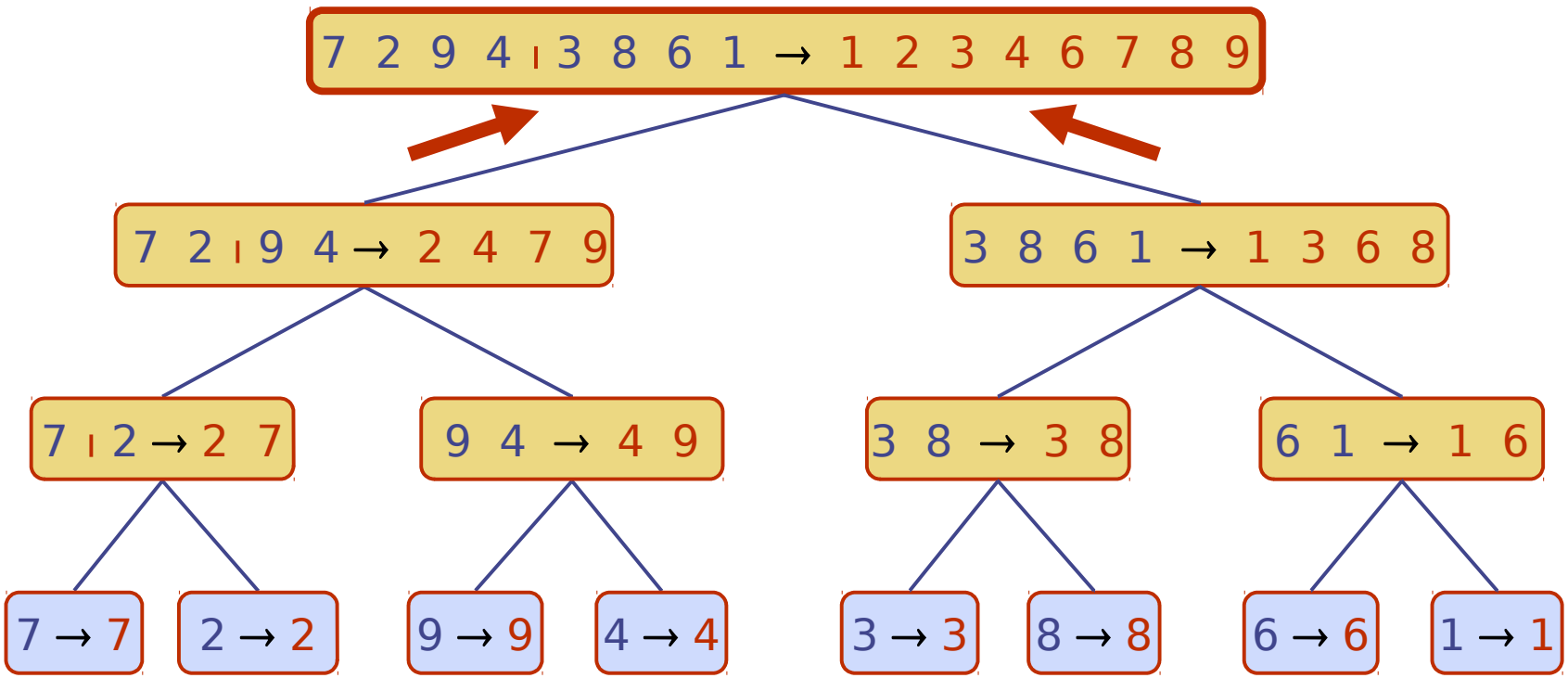
Merge Sort: Illustration IX

Recursive call, ..., merge, merge



Merge Sort: Illustration X

Merge



Merge Sort: An Implementation

```
 1  /** Merge-sort contents of array S. */
 2  public static void mergeSort(int[ ] S) {
 3    int n = S.length;
 4    if (n < 2) return;              // array is trivially sorted
 5    // divide
 6    int mid = n/2;
 7    int[ ] S1 = Arrays.copyOfRange(S, 0, mid); // copy of first half
 8    int[ ] S2 = Arrays.copyOfRange(S, mid, n); // copy of second half
 9    // conquer (with recursion)
10    mergeSort(S1);                  // sort copy of first half
11    mergeSort(S2);                  // sort copy of second half
12    // merge results
13    merge(S1, S2, S);               // merge sorted halves back into original
14  }
15  /** Merge contents of arrays S1 and S2 into properly sized array S. */
16  public static void merge(int[ ] S1, int[ ] S2, int[ ] S) {
17    int i = 0, j = 0;
18    while (i + j < S.length) {
19      if (j == S2.length || (i < S1.length && S1[i] < S2[j]))
20        S[i+j] = S1[i++];           // copy ith element of S1 and increment i
21      else
22        S[i+j] = S2[j++];           // copy jth element of S2 and increment j
23    }
24  }
```

Running Time of Merge Sort

The method `merge` runs in $O(n_1 + n_2)$, where n_1 is the length of S_1 and n_2 is the length of S_2.

- $O(1)$ for each pass of the while loop
- number of iterations is $n_1 + n_2$

The height h of the merge sort tree is $O(\log n)$.

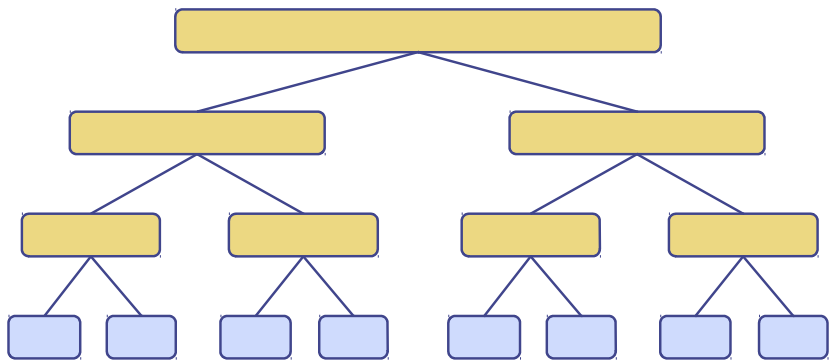
- at each recursive call we divide in half the sequence

The overall amount of work done at the nodes of depth i is $O(n)$.

- we partition and merge 2^i sequences of size $n/2^i$

Thus, the total running time of merge-sort is $O(n \log n)$

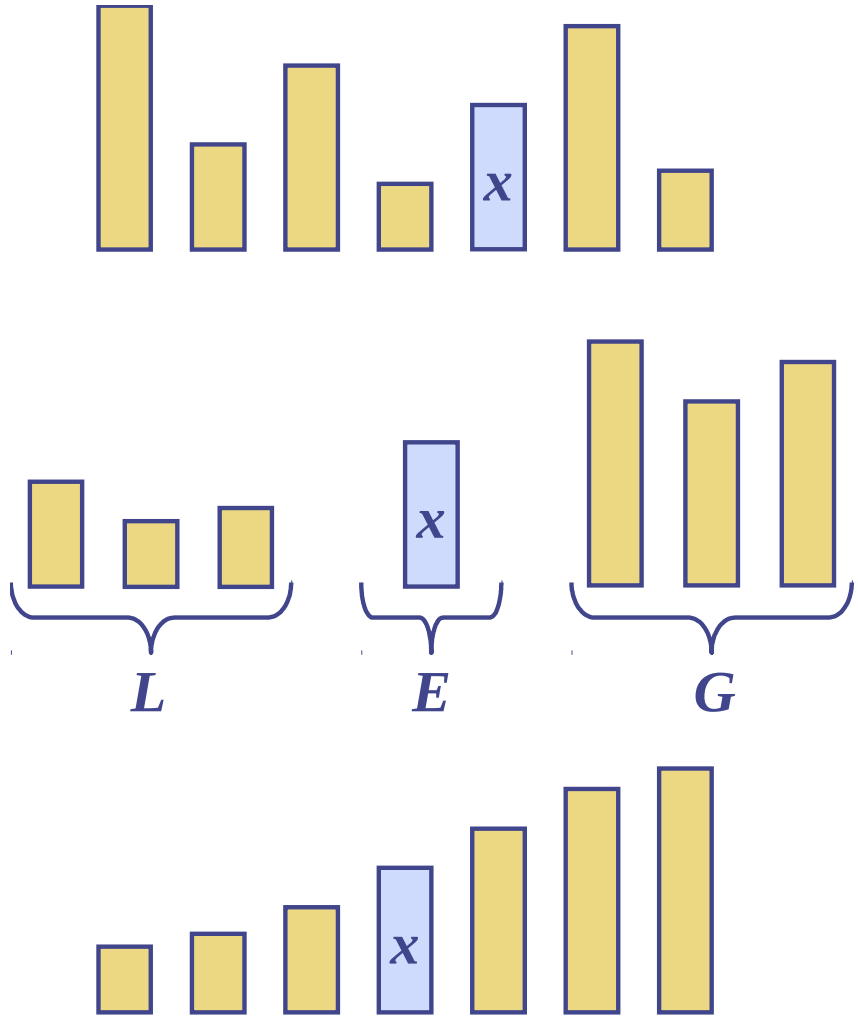
depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
...	...	...



Quick Sort

To sort a sequence S with n elements, quick sort does:

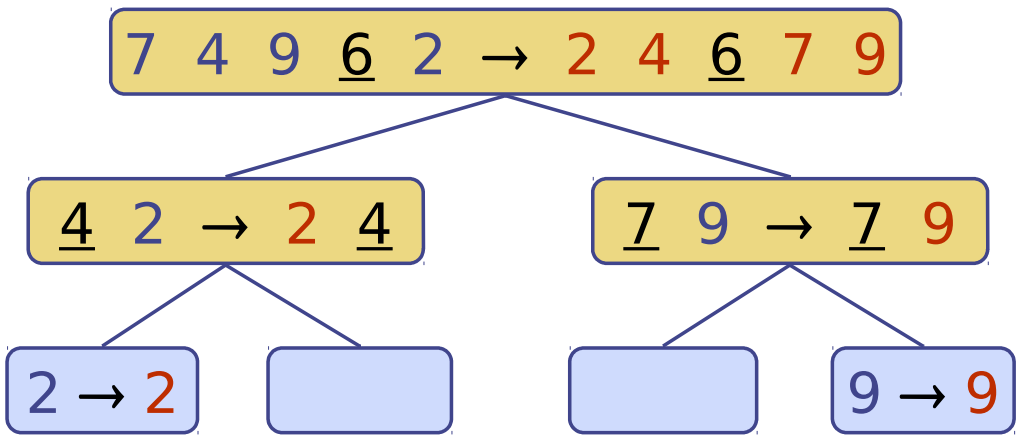
1. ***Divide:*** If $|S| \leq 1$, return S. Otherwise, select element x (called **pivot**, commonly the last element in S) and partition S into
   - L elements less than x
   - E elements equal to x
   - G elements greater than x
2. ***Conquer:*** Recursively sort sequences L and G
3. ***Combine:*** Join L, E and G



Quick Sort Tree

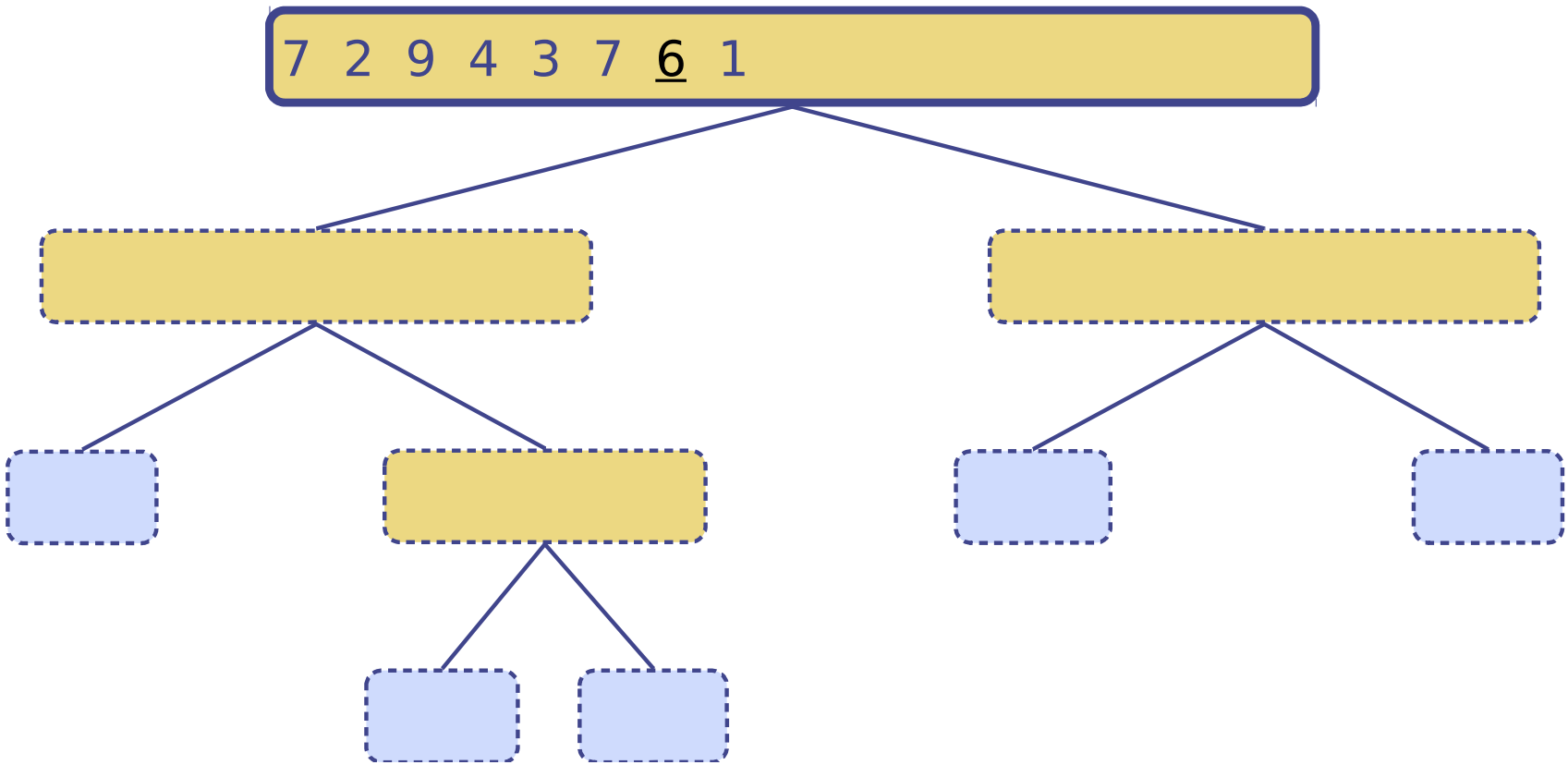
An execution of quick sort is depicted by a binary tree

- each node represents a recursive call of quick sort and stores
  - unsorted sequence before the execution and its pivot
  - sorted sequence at the end of the execution
- the root is the initial call
- the leaves are calls on subsequences of size 0 or 1



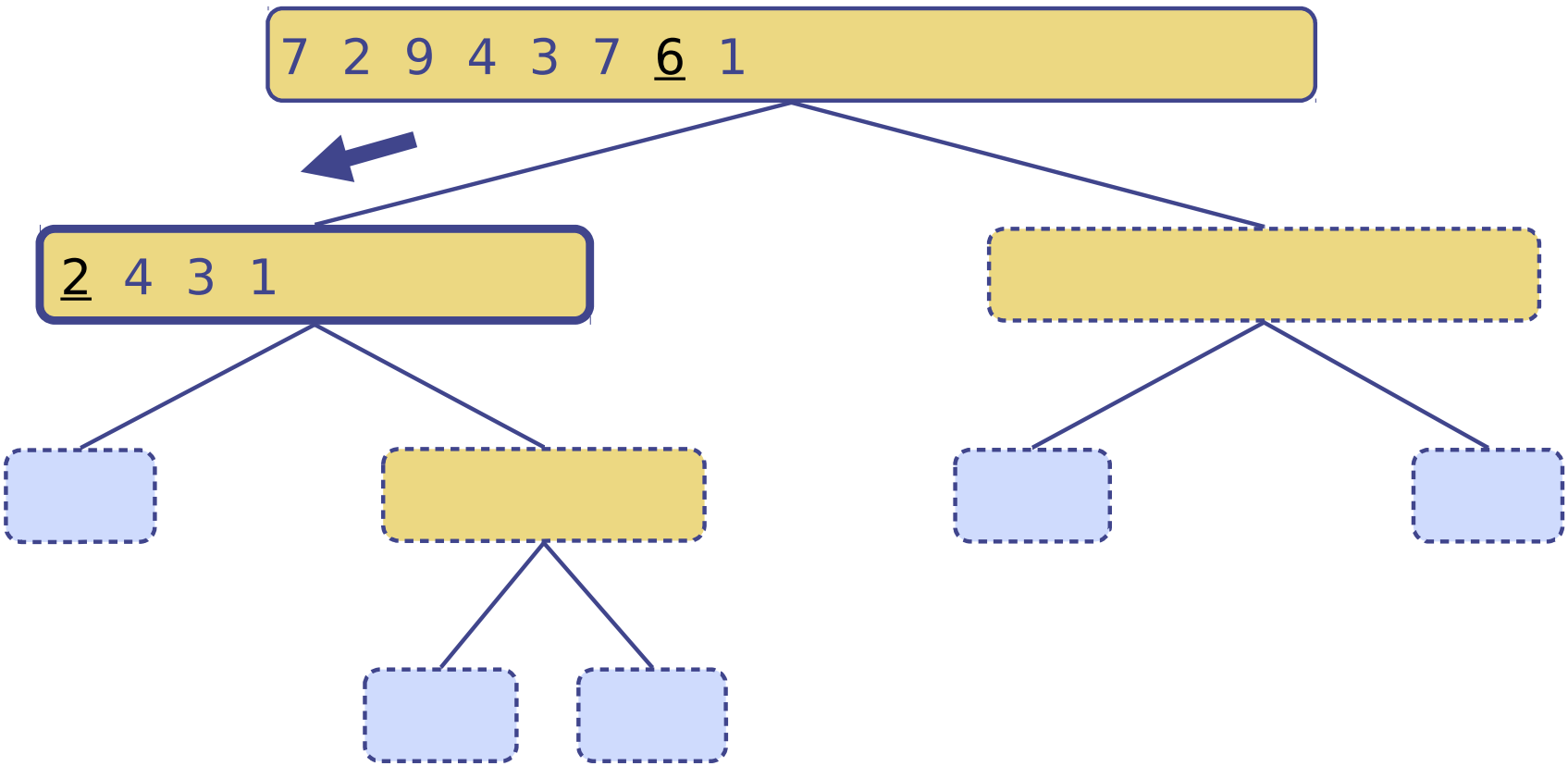
Quick Sort: Illustration I

Pivot selection



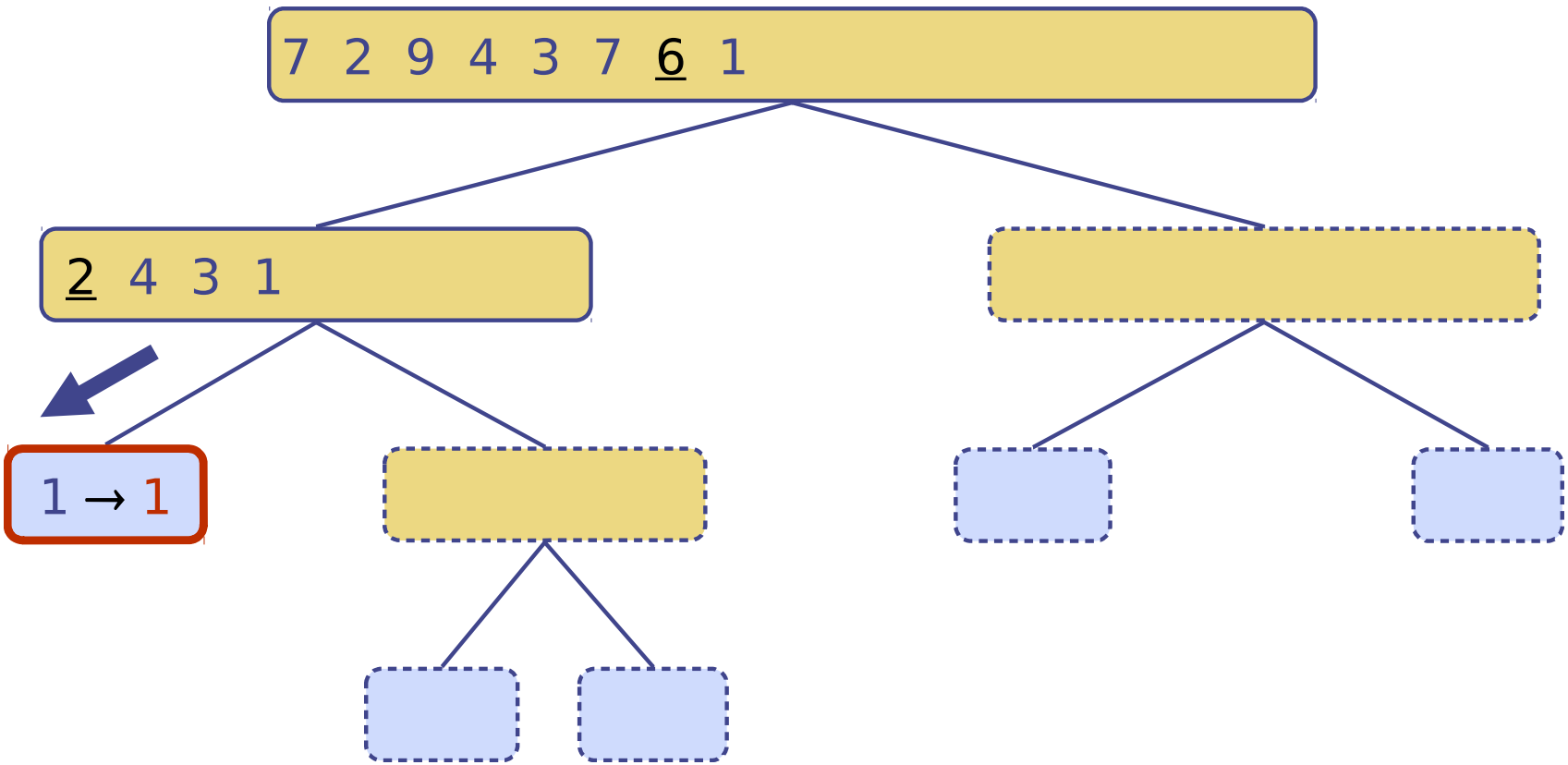
Quick Sort: Illustration II

Partition, recursive call, pivot selection



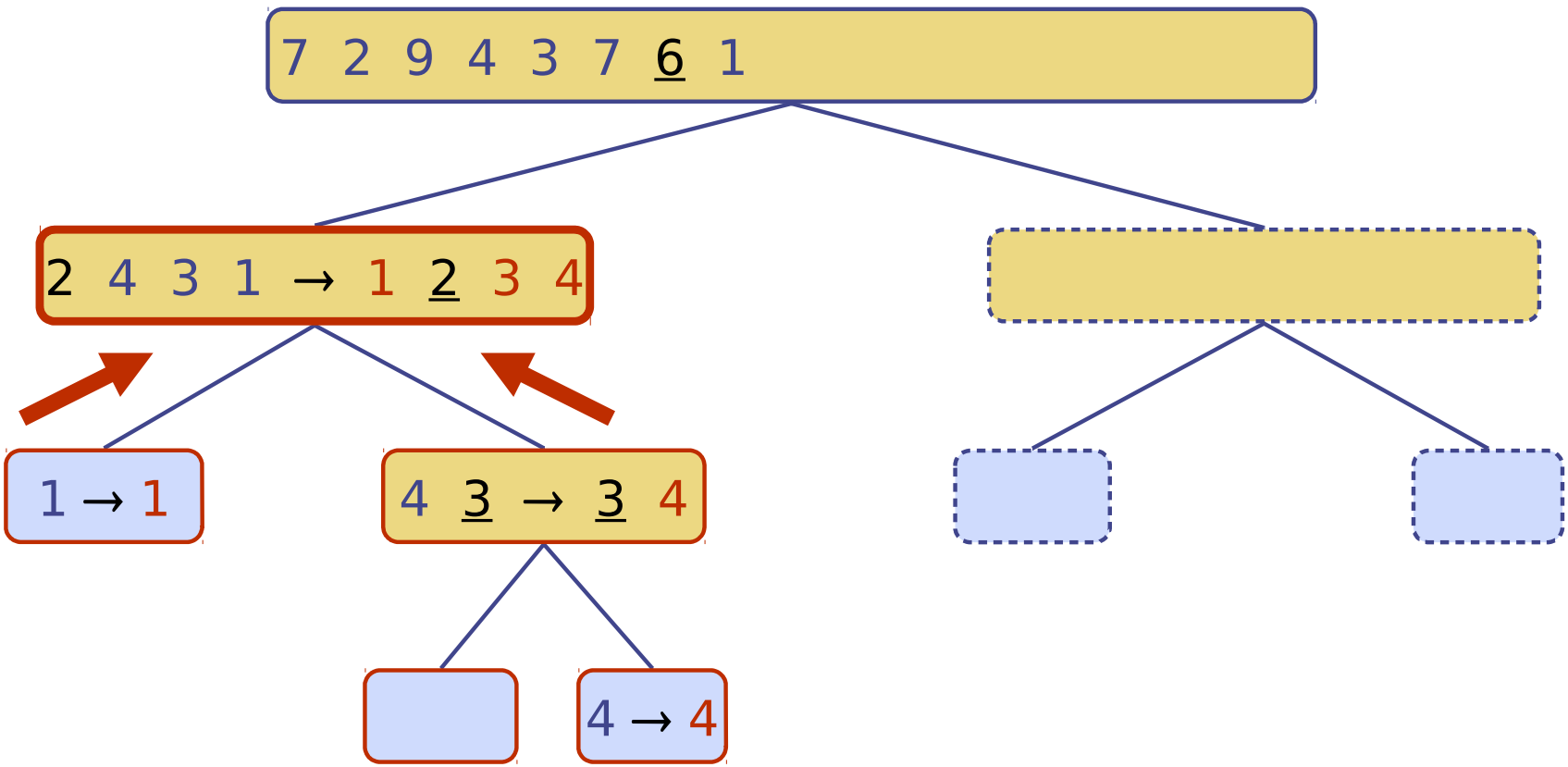
Quick Sort: Illustration III

Partition, recursive call, base case



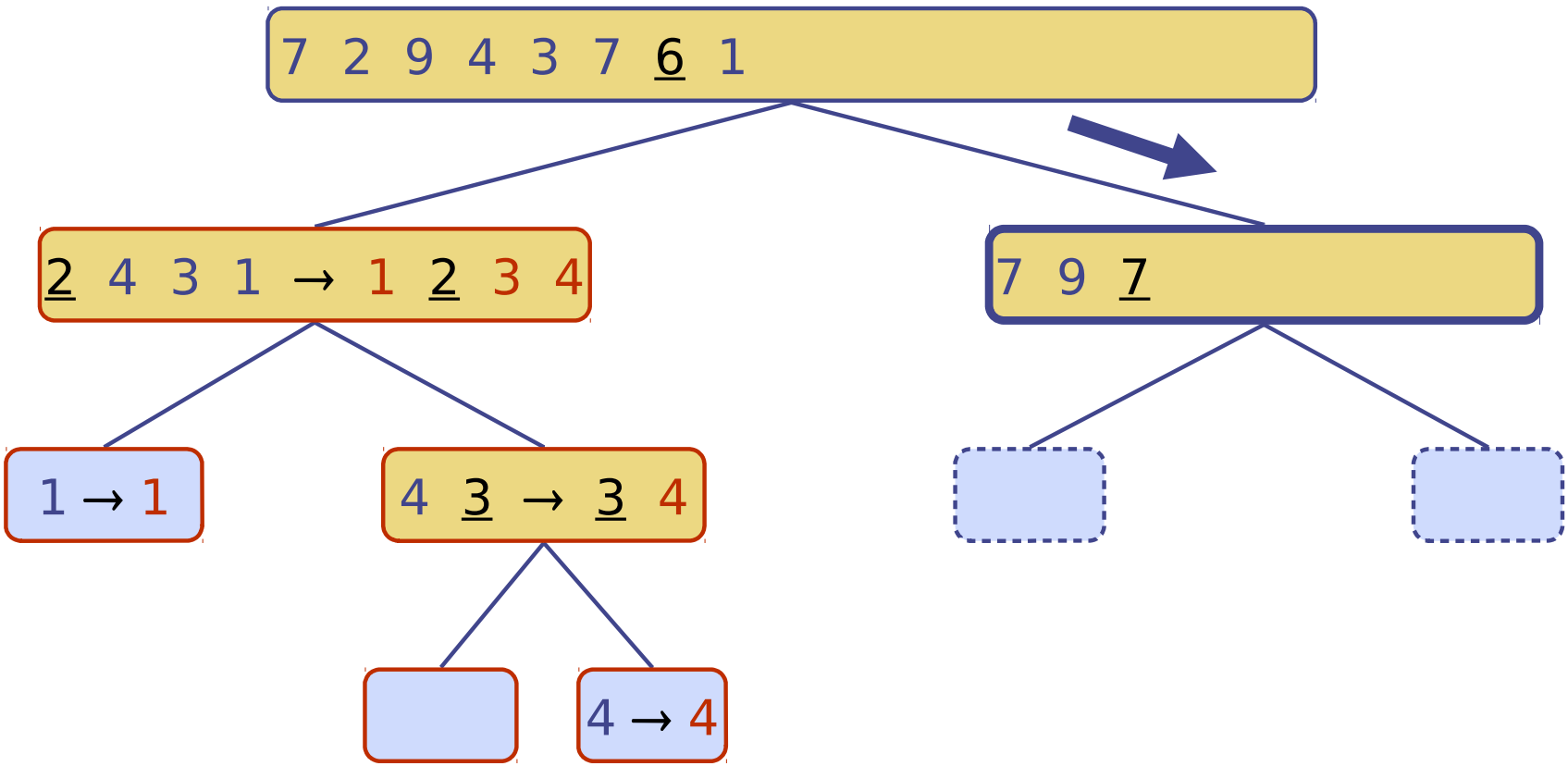
Quick Sort: Illustration IV

Recursive call, ..., base case, join



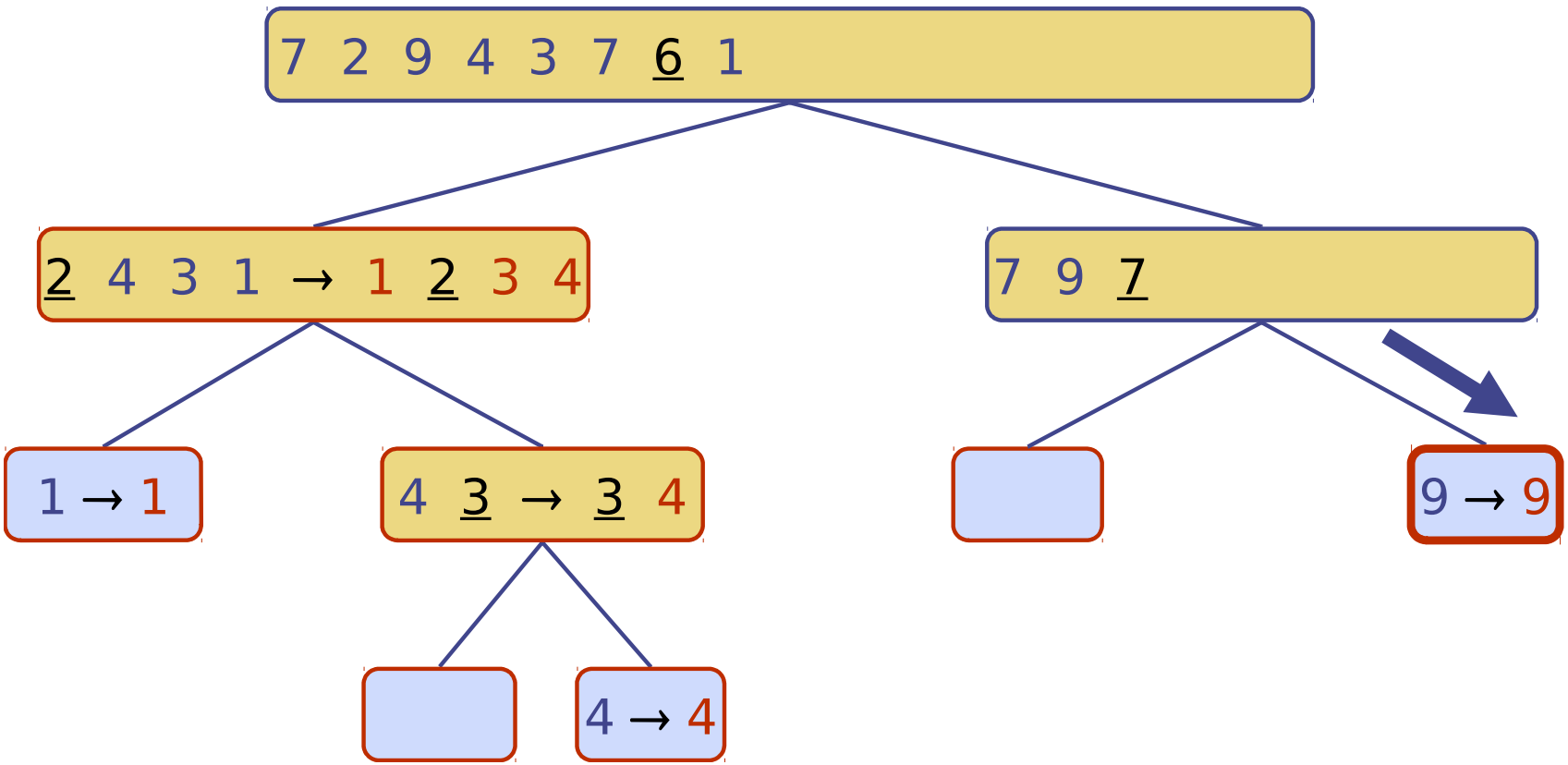
Quick Sort: Illustration V

Recursive call, pivot selection



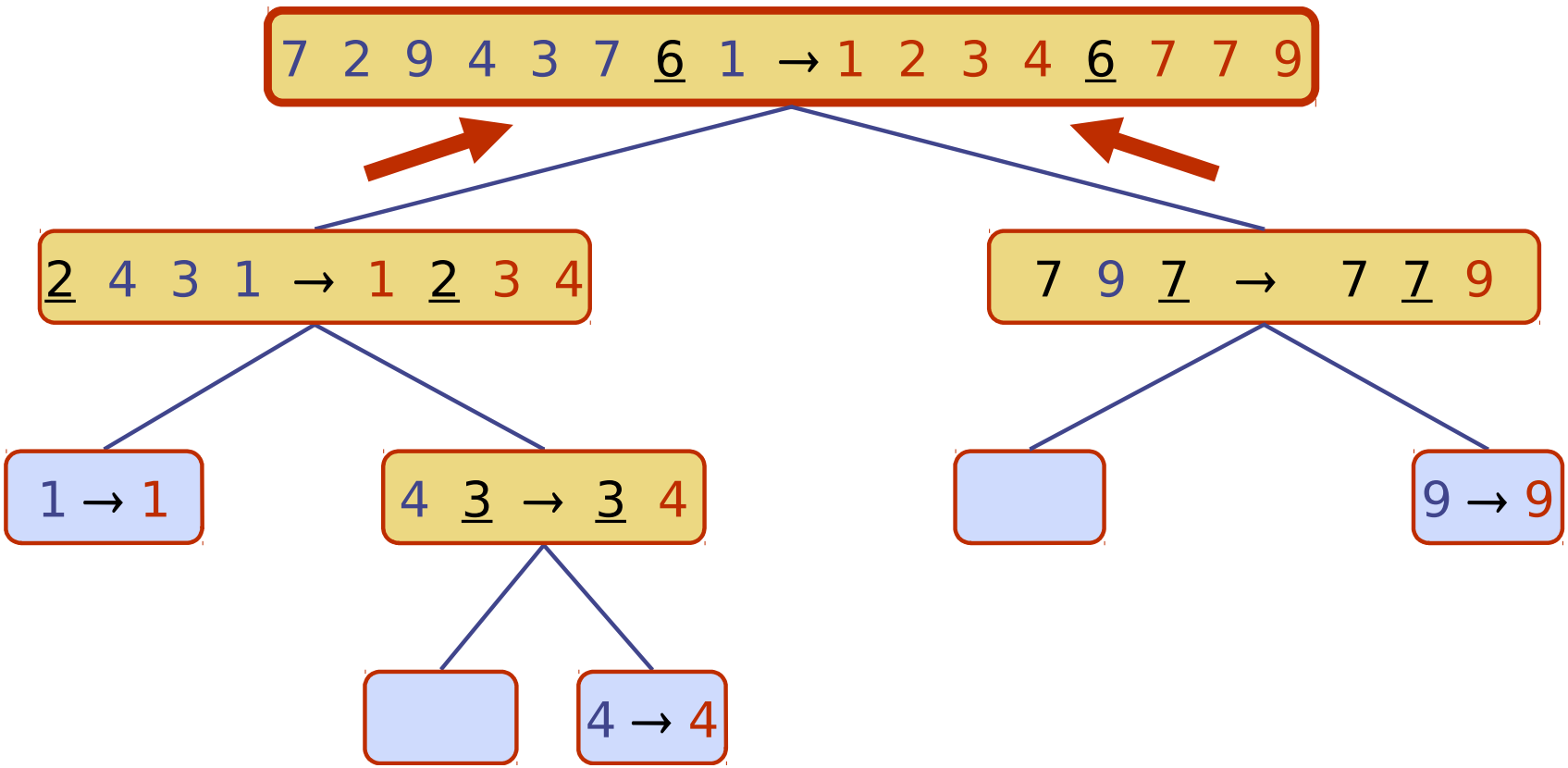
Quick Sort: Illustration VI

Partition, ..., recursive call, base case



Quick Sort: Illustration VII

Join, join



Quick Sort: An Implementation

```
 1  /** Quick-sort contents of array S. */
 2  public static void quickSort(int[ ] S) {
 3    int n = S.length;
 4    if (n < 2) return;                  // array is trivially sorted
 5    // divide
 6    int pivot = S[n-1];                 // using last as arbitrary pivot
 7    int m = 0, k = n;
 8    int[ ] temp = new int[n];
 9    for (int i = 0; i < n - 1; i++)     // divide original into L, E, and G
10      if (S[i] < pivot)                 // element is less than pivot
11        temp[m++] = S[i];
12      else if (S[i] > pivot)            // element is greater than pivot
13        temp[--k] = S[i];
14    int[ ] L = Arrays.copyOfRange(temp, 0, m);
15    int[ ] E = new int[k - m];
16    Arrays.fill(E, pivot);
17    int[ ] G = Arrays.copyOfRange(temp, k, n);
18    // conquer (with recursion)
19    quickSort(L);                       // sort elements less than pivot
20    quickSort(G);                       // sort elements greater than pivot
21    // concatenate results
22    System.arraycopy(L, 0, S, 0, m);
23    System.arraycopy(E, 0, S, m, k - m);
24    System.arraycopy(G, 0, S, k, n - k);
25  }
```

Worst-Case Running Time of Quick Sort

The worst case for quick sort occurs when the pivot is the unique minimum or maximum element

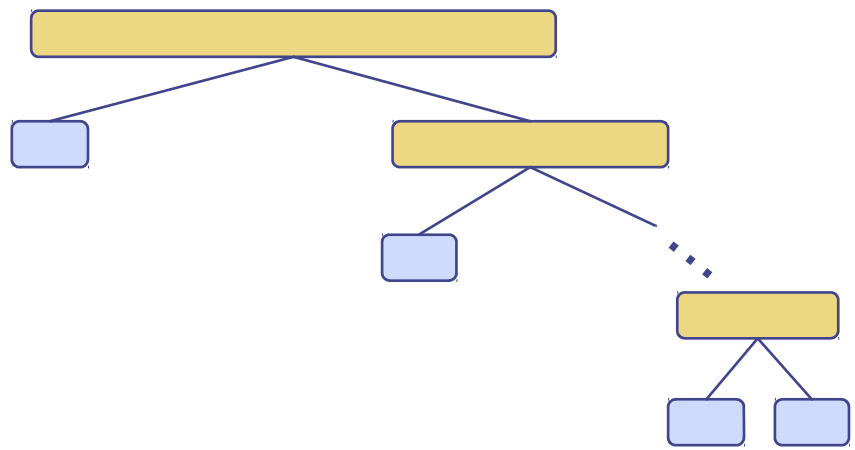
One of L and G has size $n-1$ and the other has size 0

The running time is proportional to the sum

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

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

depth	time
0	n
1	$n-1$
...	...
$n-1$	1



Expected Running Time of Quick Sort

The best case for quick sort occurs when subsequences L and G have roughly the same size

In that case, the tree has height $O(\log n)$ and therefore quick sort runs in $O(n \log n)$

A variation of quick sort, called **randomized quick sort** picks pivots at random.

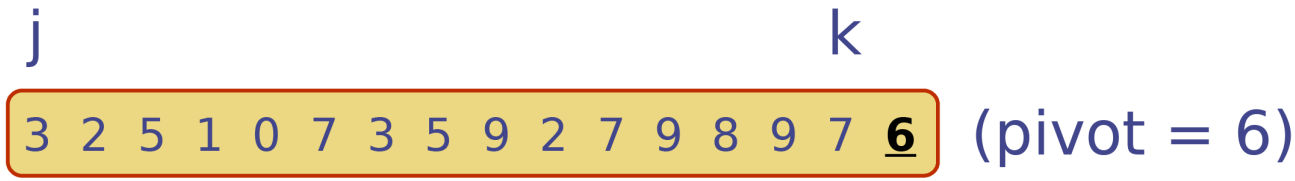
The expected running time of randomized quick sort on a sequence S of size n is $O(n \log n)$

In-Place Quick Sort

An algorithm is **in-place** if it uses only a small amount of memory in addition to that needed for the original input

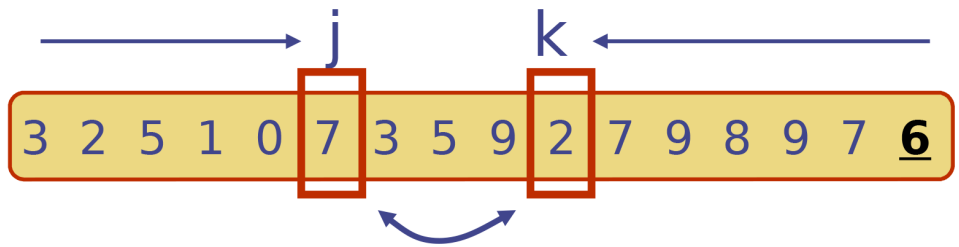
Quick sort can be adapted to be in-place

Perform the partition using two indices to split S into two subarrays



Repeat until j and k cross:

- Scan j to the right until finding an element $\geq x$.
- Scan k to the left until finding an element $\leq x$.
- Swap elements at indices j and k



In-Place Quick Sort: An Implementation

```
 1  /** Sort the subarray S[a..b] inclusive. */
 2  public static void quickSortInPlace(int[ ] S, int a, int b) {
 3    if (a >= b) return;                  // subarray is trivially sorted
 4    int left = a;
 5    int right = b - 1;
 6    int pivot = S[b];                    // using last as arbitrary pivot
 7    int temp;                            // temp object used for swapping
 8    while (left <= right) {
 9      // scan until reaching value equal or larger than pivot (or right marker)
10      while (left <= right && S[left] < pivot) left++;
11      // scan until reaching value equal or smaller than pivot (or left marker)
12      while (left <= right && S[right] > pivot) right--;
13      if (left <= right) {               // indices did not strictly cross
14        // so swap values and shrink range
15        temp = S[left]; S[left] = S[right]; S[right] = temp;
16        left++; right--;
17      }
18    }
19    // put pivot into its final place (currently marked by left index)
20    temp = S[left]; S[left] = S[b]; S[b] = temp;
21    // make recursive calls
22    quickSortInPlace(S, a, left - 1);
23    quickSortInPlace(S, left + 1, b);
24  }
```

Linear-Time Sorting

$\Omega(n \log n)$ time is necessary (refer to the book for further details), in the worst case, to sort an n-element sequence with a *comparison-based* sorting algorithm.

Sorting algorithms that run asymptotically faster than $O(n \log n)$ time exist, but require *special assumptions* about the input sequence to be sorted.

We consider two such sorting algorithms with linear time complexity

- Bucket Sort
- Radix Sort

Bucket Sort

Consider a sequence S of n integers in the range $[0, N-1]$ for some integer $N \geq 2$.

Bucket sort will sort S in $O(n+N)$ time. Furthermore, if N is $O(n)$, then the running time becomes $O(n)$.

Not being based on comparisons, bucket sort uses the integers as indices into a bucket array B that has cells indexed from 0 to $N-1$.

An element with value k is placed in the 'bucket' $B[k]$.

Finally, we enumerate the contents of the buckets $B[0], B[1], \ldots, B[N-1]$ in order to construct the sorted sequence S.

Bucket Sort Algorithm

Algorithm `bucketSort(S)`
Input sequence S of objects with integer keys in the range $[0, N-1]$
Output sequence S sorted in nondecreasing order of the keys

let B be an array of N sequences, each of which is initially empty
for each object o in S **do**
  let k denote the key of o
  remove o from S and insert it at the end of bucket $B[k]$
for $i = 0$ to $N-1$ **do**
  for each object o in sequence $B[i]$ **do**
    remove o from $B[i]$ and insert it at the end of S

Radix Sort

Let us consider pairs of integers (x, y). The **lexicographic** order of pairs (x_1, y_1) and (x_2, y_2) is defined as follows

$$(x_1, y_1) < (x_2, y_2) \iff x_1 < x_2 \ \vee \ x_1 = x_2 \ \wedge \ y_1 < y_2$$

I.e. the pairs are compared by the first dimension, then by the second dimension.

Radix sort produces lexicographic ordering by using bucket sort as the sorting algorithm in each dimension.

Radix sort is applicable to pairs where the values in each dimension are integers in the range $[0, N-1]$ and runs in time $O(2(n+N))$.

In its general form, applicable to d-tuples, radix sort runs in time $O(d(n+N))$.

Radix Sort Algorithm

Algorithm `radixSort`(S, N)
Input sequence S of d-tuples such that $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \leq (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S
Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1 **do**
`bucketSort`(S, N, i)

Radix Sort for Binary Numbers

Consider a sequence of n b-bit integers $x = x_{b-1} \ldots x_1 x_0$

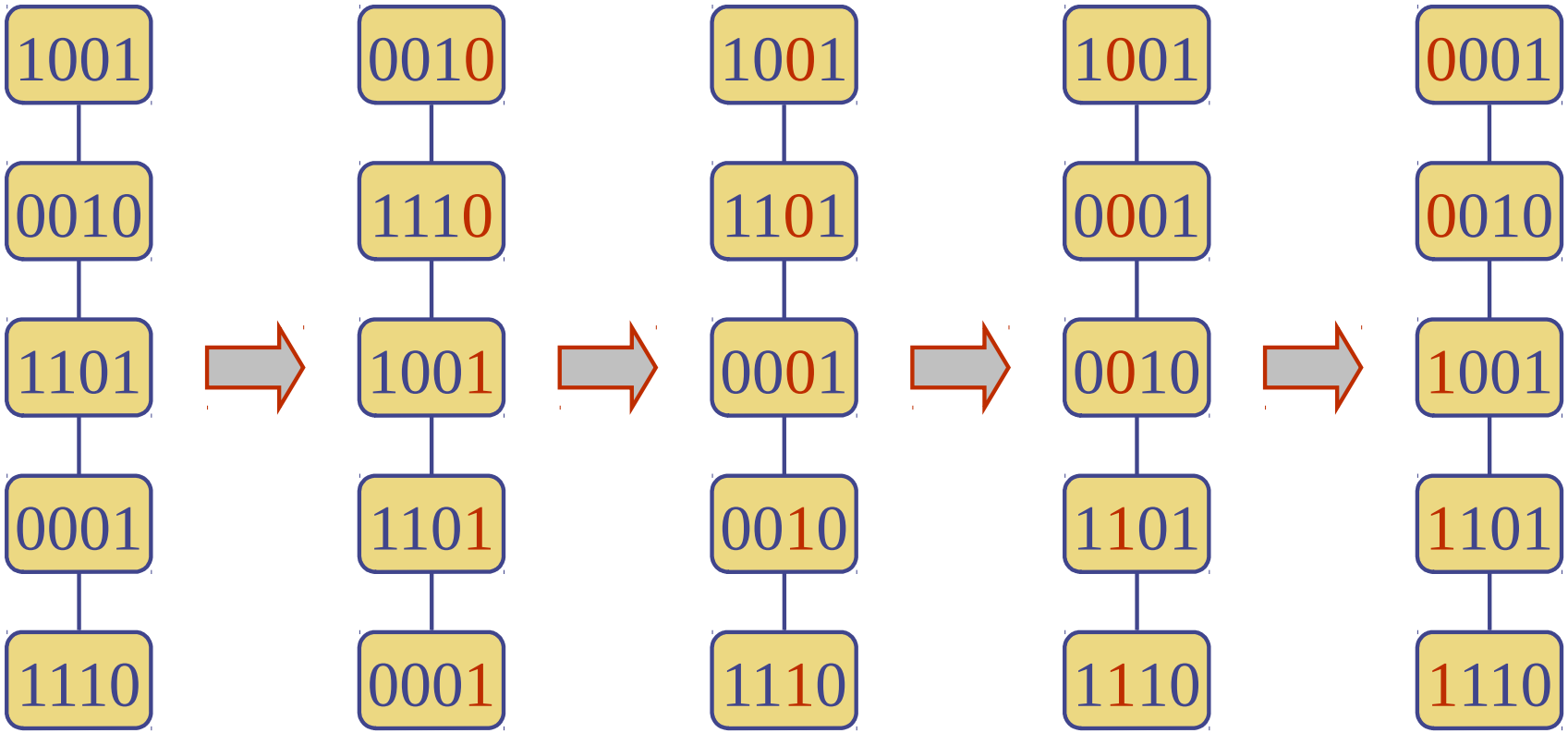
We represent each element as a b-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$

This application of the radix-sort algorithm runs in $O(bn)$ time

For example, we can sort a sequence of 32-bit integers in linear time

Radix Sort Example

Sorting a sequence of 4-bit integers



Comparison of Sorting Algorithms

Algorithm	Time	Notes
selection sort	$O(n^2)$	slow (good for small inputs)
insertion sort	$O(n^2)$	slow (good for small inputs)
bubble sort	$O(n^2)$	slow (good for small inputs)
quick sort	$O(n \log n)$ expected	fastest (good for large inputs)
merge sort	$O(n \log n)$	fast (good for huge inputs)
bucket sort	$O(n + N)$	fastest; extra requirements
radix sort	$O(d(n + N))$	fastest; extra requirements

Trade-offs involving efficiency, memory usage, ...

Summary

Reading

Chapter 12 Sorting and Selection

Questions?