Chapter 7

Sorting

Preliminaries

- We discuss the problem of sorting an array of elements.
- We will assume that the array contains only integers, although the code will allow more general objects.
- We will assume that the entire sort can be done in main memory ➔ the number of elements is relatively small (less than a few million)
- External sorting (which must be done on disk or tape), will be discussed at the end of the chapter.
- We will assume the existence of the "<" and ">" operators (besides the assignment operator). They can be used to place a consistent ordering on the input. Sorting under these conditions is known as **comparison-based sorting**.

Sorting in the STL

- The interface that will be used is not the same as in the STL sorting algorithms.
- In STL, sorting (generally **quicksort**) is accomplished by use of the function template sort.

```
void sort( Iterator begin, Iterator end );
void sort( Iterator begin, Iterator end, Comparator cmp );
```

- The iterators must support random access.
- The sort algorithm does not guarantee that equal items retain their original order (if that is important, use stable_sort instead of sort).

```
std::sort( v.begin( ), v.end( ) );
                    // sort the entire container, v, in nondecreasing order
std::sort( v.begin( ), v.end( ), greater<int>{ } );
                    // sort container v in nonincreasing order
std::sort( v.begin( ), v.begin( ) + ( v.end( ) - v.begin( ) ) / 2 );
                   // sort first half of container v in nondecreasing order
```

Insertion Sort

- It is one of the simplest sorting algorithms.
- Insertion sort consists of $N-1$ **passes.** For pass $p=1$ through $N-1$, insertion sort ensures that the elements in positions 0 through p are in sorted order
- It makes use of the fact that elements in positions 0 through $p-1$ are already known to be in sorted order.
- In pass p, we move the element in position p left until its correct place is found among the first $p+1$ elements.
- The element in position p is moved to *tmp*, and all larger elements (prior to position p) are moved one spot to the right. Then tmp is moved to the correct spot.

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

```
// Simple insertion sort.
template <typename Comparable>
void insertionSort( vector<Comparable> & a )
{
        for( int p = 1; p < a.size( ); ++p ) {
            Comparable tmp = std::move( a[ p ] );
            int j;
            for( j = p; j > 0 && tmp < a[ j - 1 ]; --j )
                a[ j ] = std::move( a[ j - 1 ] );
            a[ j ] = std::move( tmp );
    }
}
```

STL Implementation of Insertion Sort

- 2-parameter sort

```
// The two-parameter version calls the three-parameter version,
// using C++11 decltype
template <typename Iterator>
void insertionSort( const Iterator & begin, const Iterator & end )
{       insertionSort( begin, end, less<decltype(*begin)>{ } ); }
```

// the 3rd parameter is a function object

```
// 3-parameter sort
template <typename Iterator, typename Comparator>
void insertionSort( const Iterator & begin, const Iterator & end, Comparator lessThan )
{
        if( begin == end )
                return;
        Iterator j;
        for( Iterator p = begin+1; p != end; ++p ) {
                auto tmp = std::move( *p );
                for( j = p; j != begin && lessThan( tmp, *( j-1 ) ); --j )
                        *j = std::move( *(j-1) );
                *j = std::move( tmp );
        }
}
```

Analysis of Insertion Sort

- Because of the nested loops, each of which can take N iterations, insertion sort is $O(N^2)$.
- A precise calculation shows that the number of tests in the inner loop is at most $p + 1$ for each value of p. Summing over all p gives a total of

$$\sum_{i=2}^{N} i = 2 + 3 + 4 + \cdots + N = \Theta(N^2)$$

- If the input is pre-sorted (or almost sorted), the running time is $O(N)$, because the test in the inner for loop always fails immediately

- An **inversion** in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$.
- In the previous example: the input list 34, 8, 64, 51, 32, 21 had nine inversions, namely (34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21), and (32, 21).
- This is exactly the number of swaps that needed to be (implicitly) performed by insertion sort.
- This is always the case, because swapping two adjacent elements that are out of place removes exactly one inversion, and a sorted array has no inversions.
- Since there is $O(N)$ other work involved in the algorithm, the running time of insertion sort is $O(I + N)$, where I is the number of inversions in the original array
- Thus, insertion sort runs in linear time if the number of inversions is $O(N)$.

- We can compute precise bounds on the average running time of insertion sort by computing the average number of inversions in a permutation.
- We will assume that there are no duplicate elements
- Using this assumption, we can assume that the input is some permutation of the first N integers
- Under these assumptions, we have the following theorem:

Theorem 7.1: The average number of inversions in an array of N distinct elements is $N(N-1)/4$. (Proof: See textbook.)

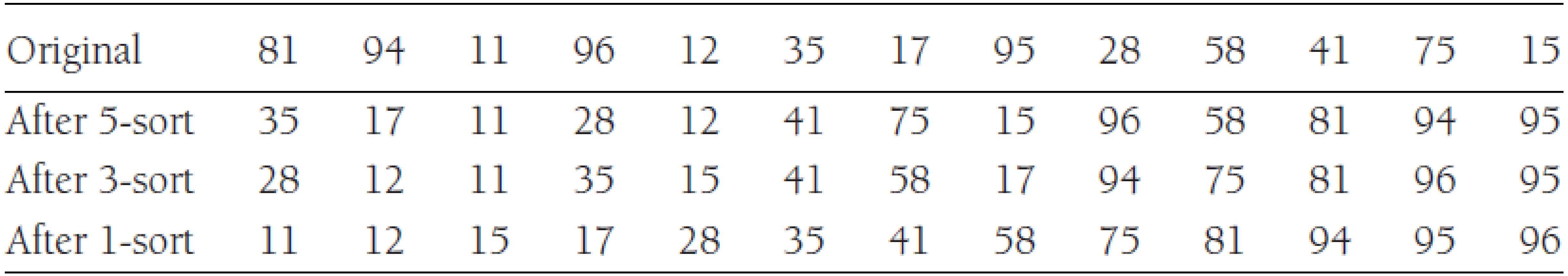
- This theorem implies that insertion sort is quadratic on average. It also provides a very strong lower bound about any algorithm that only exchanges adjacent elements.

Theorem 7.2: Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average. (Proof: See textbook.)

Shellsort

- Shellsort (Donald Shell) was one of the first algorithms to break the quadratic time barrier
- It works by comparing elements that are distant; the distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared.
- For this reason, Shellsort is sometimes referred to as **diminishing increment sort**.
- Shellsort uses a sequence, $h_1, h_2, \ldots, h_t$, called the **increment sequence**.
- Any increment sequence will do as long as $h_1 = 1$, but some choices are better than others.
- After a *phase*, using some increment h_k, for every i, we have $a[i] \leq a[i + h_k]$
- Important property of Shellsort: if you h_{k-1}-sort an h_k-sorted file, then it remains h_k-sorted

- The general strategy to *h_k*-sort is for each position, *i*, sort the subarray starting at i of elements obtained by increments *h_k*; then take the next (smaller) h_k and so on.
- Suppose the increment sequence is 5, 3, 1



Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

- A popular (but poor) choice for increment sequence is to use the sequence suggested by Shell: h_t = floor(N / 2), and h_k = floor(h_{k+1} / 2).

```
// Shellsort routine using Shell's increments (better increments are possible)
template <typename Comparable>
void shellsort( vector<Comparable> & a )
{
        for( int gap = a.size( ) / 2; gap > 0; gap /= 2 )
                for( int i = gap; i < a.size( ); ++i )
                {
                        Comparable tmp = std::move( a[ i ] );
                        int j = i;
                        for( ; j >= gap && tmp < a[ j - gap ]; j -= gap )
                                a[ j ] = std::move( a[ j - gap ] );
                        a[ j ] = std::move( tmp );
                }
}
```

Worst-Case Analysis of Shellsort

- The running time of Shellsort depends on the choice of increment sequence, and the proofs can be rather involved.
- The average-case analysis of Shellsort is a long-standing open problem, except for the most trivial increment sequences.

Theorem 7.3: The worst-case running time of Shellsort using Shell's increments is $\Theta(N^2)$.

- Hibbard suggested a slightly different increment sequence, which gives better results in practice (and theoretically): 1, 3, 7, . . . , $2^k - 1$.
- Key difference: consecutive increments have no common factors.
- For this increment sequence, we have the following theorem:

Theorem 7.4: The worst-case running time of Shellsort using Hibbard's increments is $\Theta(N^{3/2})$.

- The performance of Shellsort is quite acceptable in practice, even for N in the tens of thousands. The simplicity of the code makes it the algorithm of choice for sorting up to moderately large input.

Heapsort

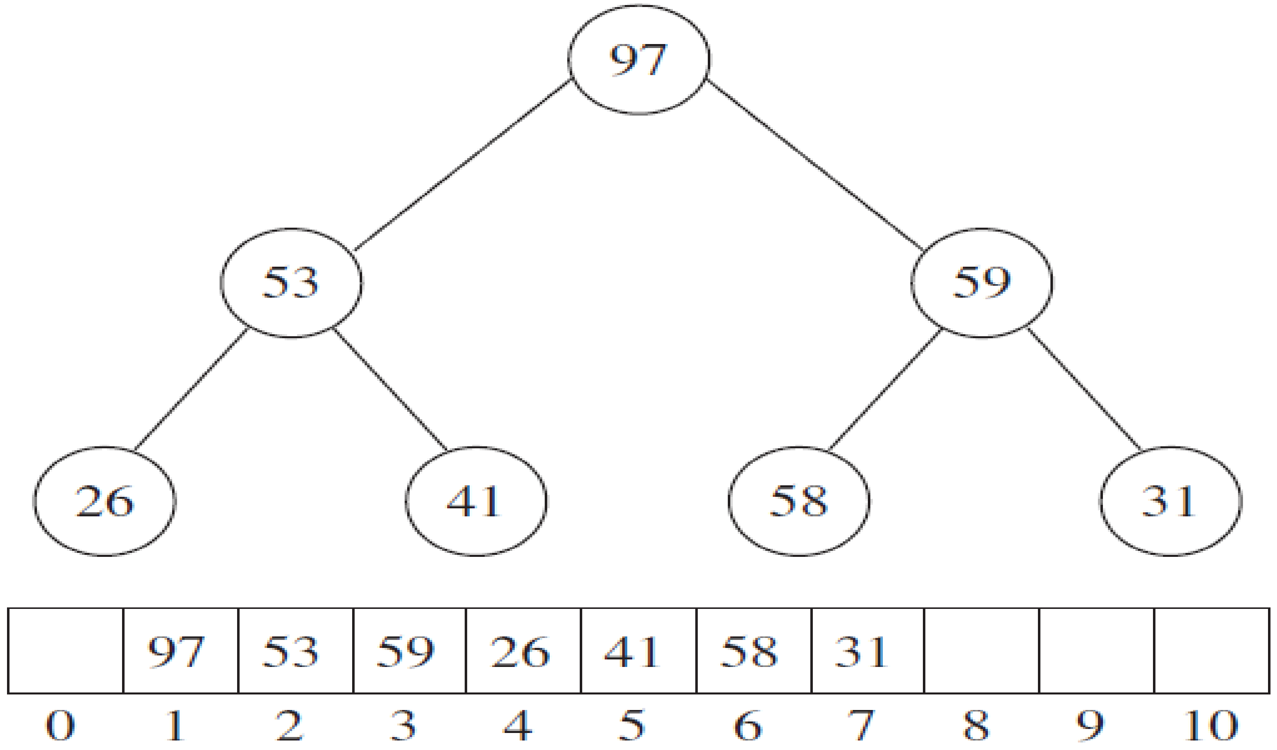
- Priority queues can be used to sort in $O(N \log N)$ time. The algorithm based on this idea is known as **heapsort**
- Reminder from Chapter 6: basic strategy is
  - ✓ ***build*** a binary heap of N elements. This stage takes $O(N)$ time.
  - ✓ then perform N ***deleteMin*** operations.
- The elements leave the heap smallest first, in sorted order.
- By recording these elements in a second array and then copying the array back, we sort N elements.
- Since each deleteMin takes $O(\log N)$ time, the total running time is $O(N \log N)$.
- The main problem with this algorithm is that it uses an extra array. Thus, the memory requirement is doubled.

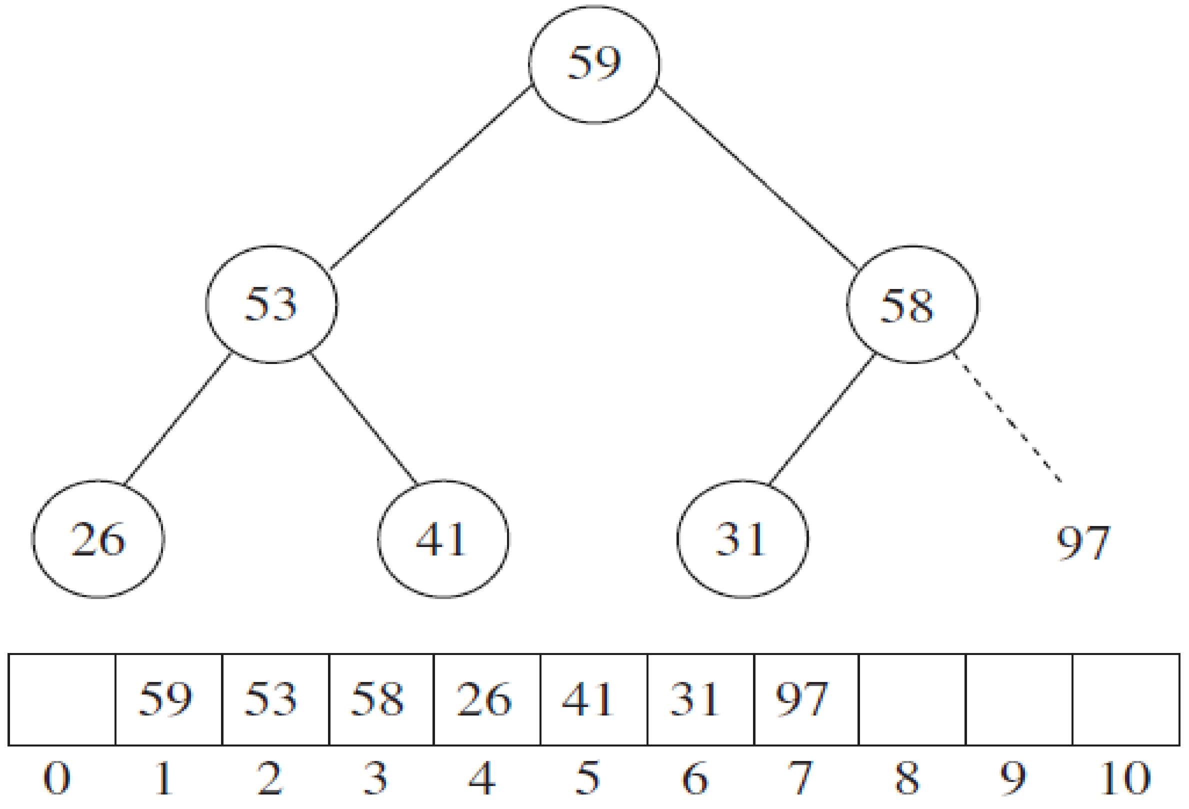
Alternative to doubling the array

- Make use of the fact that after each *deleteMin*, the heap shrinks by 1.
- ➔ the cell that was last in the heap can be used to store the element that was just deleted.
- Example, suppose we have a heap with six elements.
  - The first *deleteMin* produces a_1.
  - Now the heap has only five elements, ➔ we can place a_1 in position 6.
  - The next *deleteMin* produces a_2. The heap will now only have four elements ➔ we can place a_2 in position 5.
  - And so on
- After the last *deleteMin* the array will contain the elements in *decreasing* sorted order. If we want the elements in the more typical *increasing* sorted order, we can change the ordering property so that the parent has a larger element than the child. Thus, we have a (*max*)heap.

Build a heap from the list 58, 41, 59, 26, 53, 97, 31

(*Max*) heap after buildHeap phase



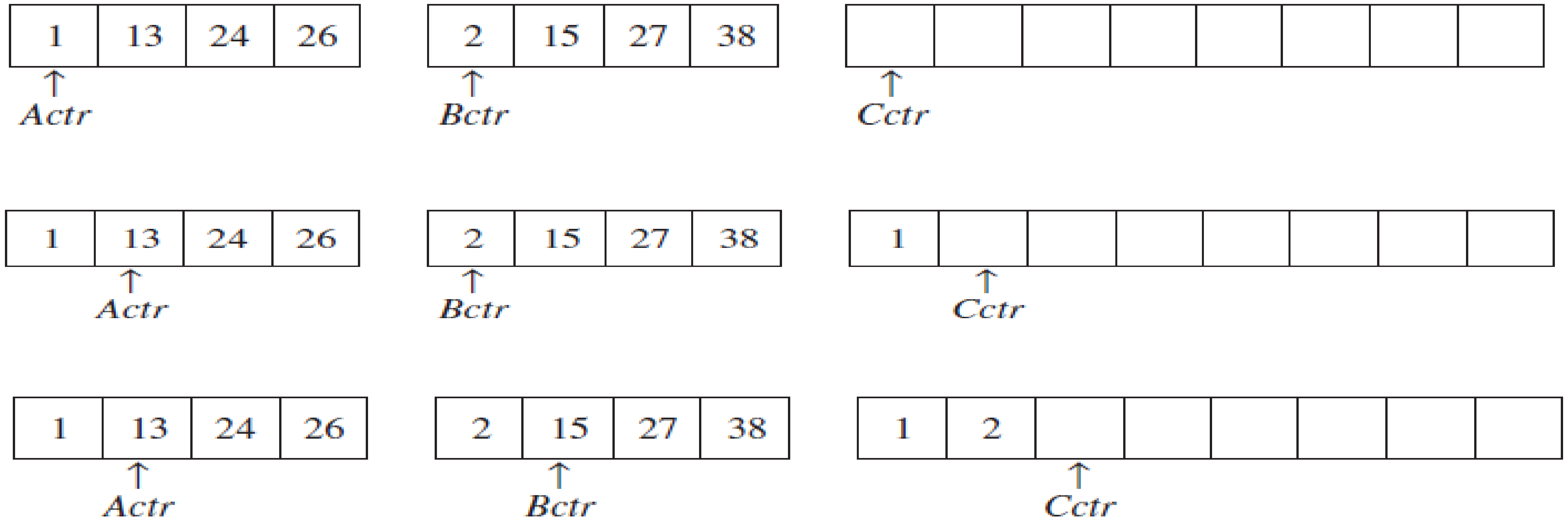


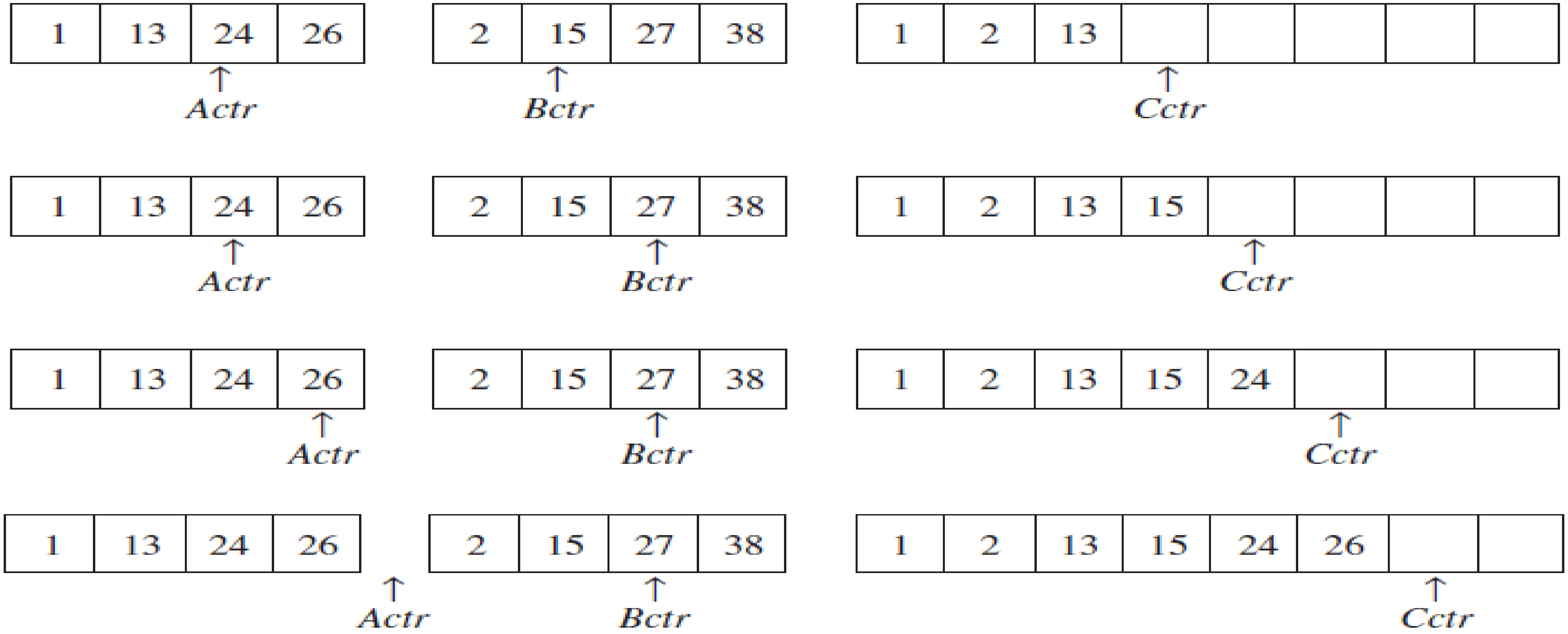
Heap after first deleteMax

Mergesort

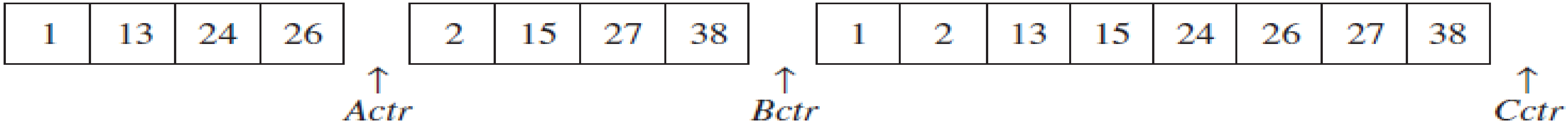
- The fundamental operation in Mergesort is merging two sorted lists.
- Since the lists are sorted, this can be done in one pass through the input if the output is put in a third list.
- Basic **merging** algorithm:
  - Input arrays *A* and *B*, an output array *C*,
  - Use three counters, *Actr*, *Bctr*, and *Cctr*, initially set to the beginning of their respective arrays.
  - The smaller of *A*[*Actr*] and *B*[*Bctr*] is copied to the next entry in *C*, and the appropriate counters are advanced.
  - When either input list is exhausted, the remainder of the other list is copied to *C*.
- The merge operation is clearly linear O(N)

Merge the following two arrays





The remainder of the *B* array is then copied to *C*



Mergesort algorithm

- If $N = 1$, there is only one element to sort, and the answer is the element itself.
- Otherwise, recursively ***mergeSort*** the first half and the second half.
- This gives two sorted halves, which can then be merged together using the ***merging*** algorithm described above.
- For instance, to sort the eight-element array 24, 13, 26, 1, 2, 27, 38, 15
  - Recursively sort the first four and last four elements, obtaining 1, 13, 24, 26, 2, 15, 27, 38.
  - Then merge the two halves as above, obtaining the final list 1, 2, 13, 15, 24, 26, 27, 38.
- This algorithm is a classic divide-and-conquer strategy.

- The problem is *divided* into smaller problems and solved recursively.
- The *conquering* phase consists of patching together the answers.
- Divide-and-conquer is a very powerful use of recursion that we will see many times.

  Mergesort routines

  Merge routine

- The merge routine is subtle. If a temporary array is declared locally for each recursive call of merge, then there could be logN temporary arrays active at any point.
- A close examination shows that since merge is the last line of *mergeSort*, there only needs to be one temporary array active at any point, and that the temporary array can be created in the public mergeSort driver.
- Further, we can use any part of the temporary array; we will use the same portion as the input array a.

Analysis of Mergesort

- Mergesort is a classic example of the techniques used to analyze recursive routines: We have to write a recurrence relation for the running time.
- We will assume that N is a power of 2 so that we always split into even halves.
- For $N = 1$, the time to mergesort is constant, which we will denote by 1.
- Otherwise, the time to mergesort N numbers is equal to the time to do two recursive mergesorts of size N/2, plus the time to merge, which is linear. So

$T(1) = 1$

$T(N) = 2T(N/2) + N$

Since we can substitute N/2 into the main equation,

$2T(N/2) = 2(2(T(N/4)) + N/2) = 4T(N/4) + N$

We have

$T(N) = 4T(N/4) + 2N$

Again, by substituting N/4 into the main equation, we see that

$4T(N/4) = 4(2T(N/8) + N/4) = 8T(N/8) + N$

So we have

$T(N) = 8T(N/8) + 3N$

Continuing in this manner, we obtain

$T (N) = 2^k T (N/ 2^k) + k \cdot N$

Using $k = \log N$, we obtain

$T(N) = N\,T(1) + N \log N = N \log N + N$

Remarks on MergeSort

- Though we assumed $N = 2^k$, the analysis can be refined to handle cases when N is not a power of 2. (Answer almost identical).
- Although *mergeSort*'s running time is $O(N \log N)$, it has the significant problem that
  - merging two sorted lists uses linear extra memory; and
  - the additional work involved in copying to the temporary array and back, throughout the algorithm, slows the sort considerably.
- This copying can be avoided by judiciously switching the roles of *a* and *tmpArray* at alternate levels of the recursion.
- A non-recursive implementation of mergeSort is also possible.
- The running time of *mergeSort*, when compared with other $O(N \log N)$ alternatives, depends heavily on the relative costs of comparing elements and moving elements in the array (and the temporary array).
- These costs are language dependent. (See the discussion Java vs C++ in textbook.)

QuickSort

- For C++, **quicksort** has historically been the fastest known generic sorting algorithm in practice.
- Its average running time is $O(N \log N)$.
- It has $O(N^2)$ worst-case performance.
- By combining **quicksort** with **heapsort**, we can achieve quicksort's fast running time on almost all inputs, with heapsort's $O(N \log N)$ worst-case running time. (Left as Exercise 7.27, which describes this approach).
- Like **mergeSort**, **quickSort** is a divide-and-conquer recursive algorithm.

QuickSort Algorithm

- Let us begin with the following simple sorting algorithm to sort a list.
  - Given a list of items to sort
  - Arbitrarily choose any item
  - Form three groups: those smaller than the chosen item, those equal to the chosen item, and those larger than the chosen item.
  - Recursively sort the first and third groups
  - Concatenate the three groups.

Implementation of this simple recursive sorting algorithm

- This algorithm forms the basis of quicksort. But it does not change much from **mergeSort**, especially in terms of extra memory.
- **quicksort** is commonly written in a way that avoids creating the second group (the equal items), and the algorithm has numerous subtle details that affect the performance; therein lie the complications.

“Classic quicksort”

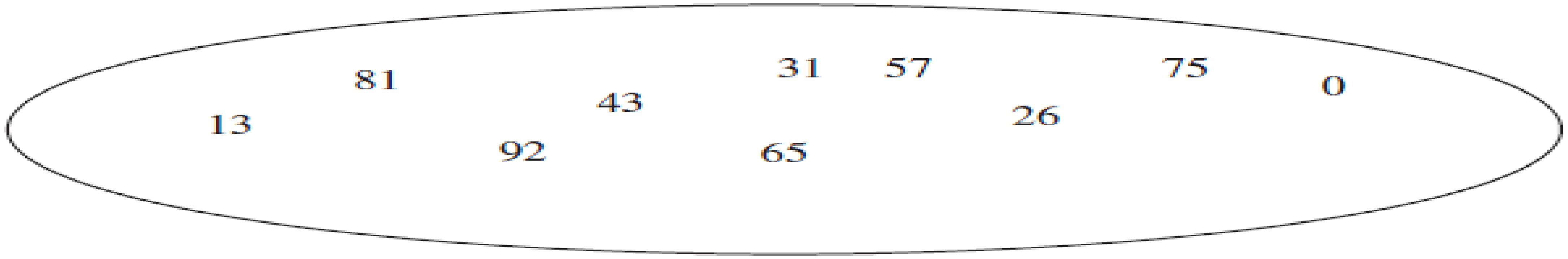
- The following is the most common implementation of quicksort.
- Only one array S is used; it is the input array.

Four steps:

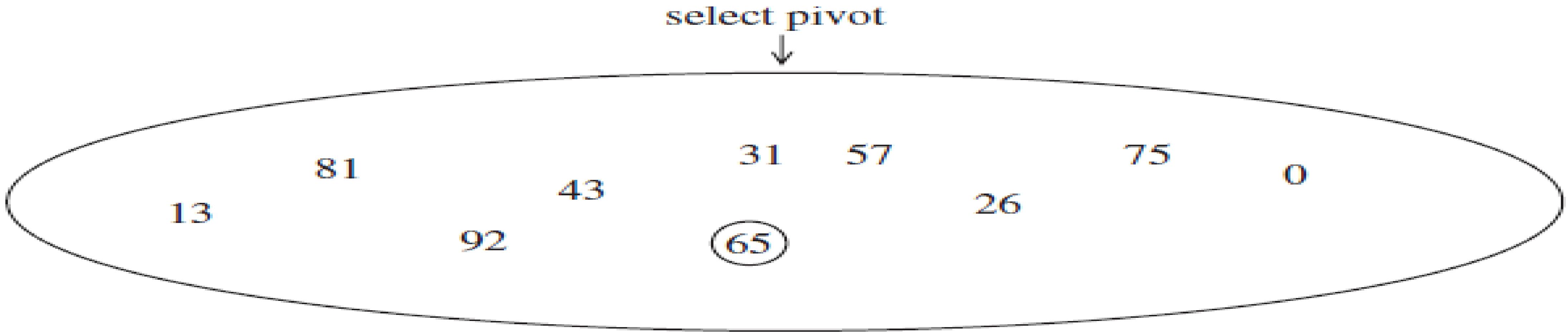
1. If the number of elements in S is 0 or 1, then return.
2. Pick any element v in S. This is called the **pivot.**
3. **Partition** $S - \{v\}$ (the remaining elements in S) into two disjoint groups: $S_1 = \{x \in S - \{v\} \mid x \leq v\}$, and $S_2 = \{x \in S - \{v\} \mid x \geq v\}$.
4. Return {quicksort(S_1) followed by v followed by quicksort(S_2)}.

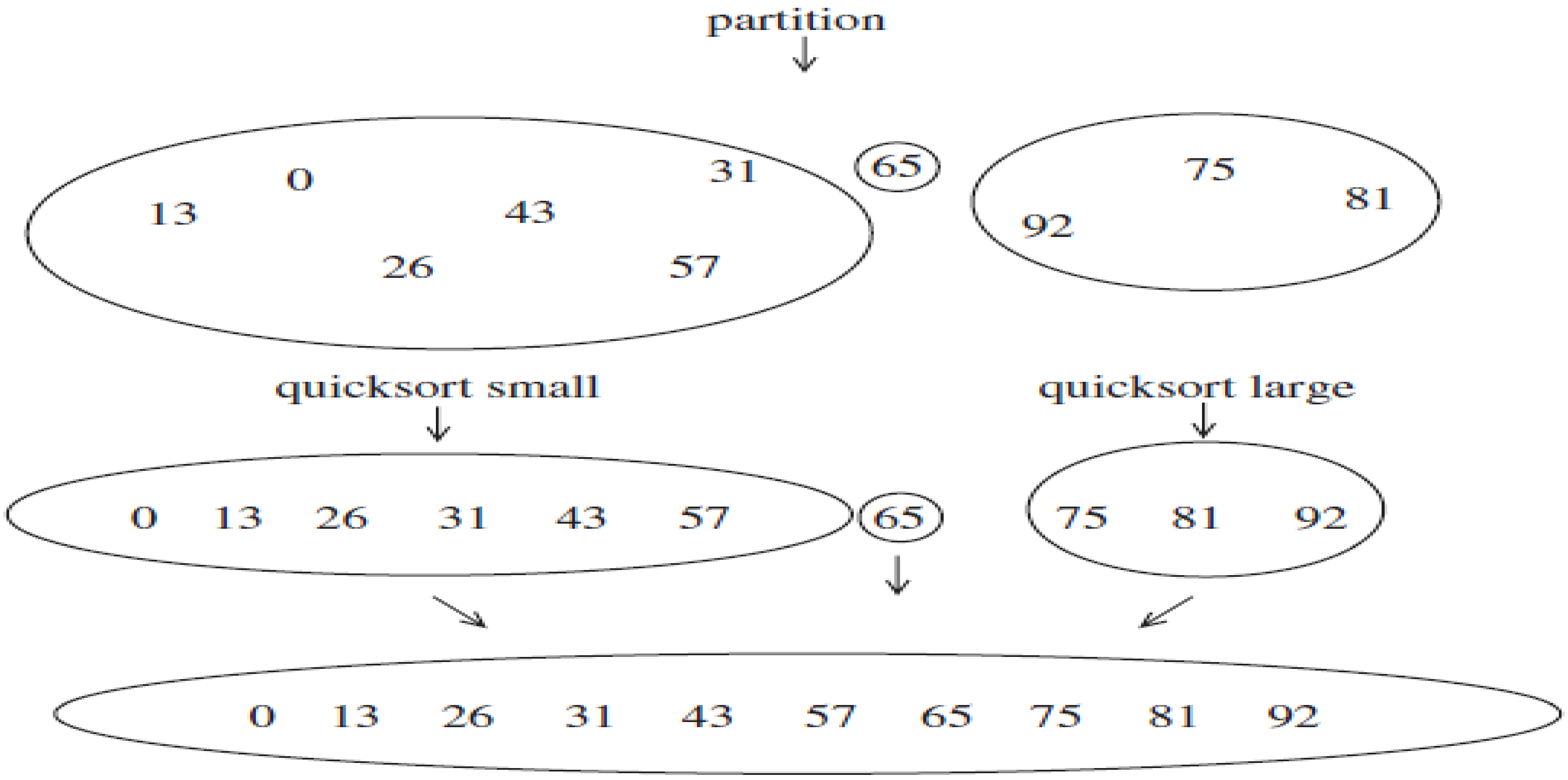
- The devil lies in the detail (of parts 2 and 3 of the algorithm)!

Example to sort a list of numbers



The pivot is selected randomly to be 65




partition
0
31
65
13
43
26
57
75
81
92
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Discussion

- The previous algorithm works, but is it any faster than *mergesort*?
- Like *mergesort*, it recursively solves two subproblems and requires linear additional work.
- But, unlike *mergesort*, the subproblems are not guaranteed to be of equal size, which is potentially bad.
- *quicksort* is faster because the partitioning step can actually be performed in place and very efficiently.
- This efficiency more than makes up for the lack of equal-sized recursive calls.

Picking the Pivot

- The algorithm as described works whichever element is chosen as pivot; some choices are obviously better than others;
- The choice to use the first element as the pivot is acceptable if the input is random; if the input is pre-sorted (or input with a large pre-sorted section) or in reverse order, then the pivot provides a poor partition, because either all the elements go into S_1 or they go into S_2.
- If the first element is used as the pivot and the input is pre-sorted, then quicksort will take quadratic time to do essentially nothing at all,.
- Pre-sorted input is quite frequent, so using the first element as pivot *is a very bad idea.*
- A safe course is to choose the pivot randomly, unless the random number generator has a flaw.

Median-of-three Partitioning

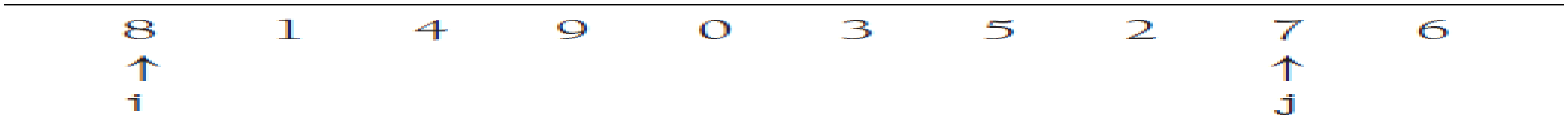
- The median of a group of N numbers is the ceiling($N/2$)th largest number.
- The best choice of pivot would be the median of the array.
- This is hard to calculate and would slow down *quicksort* considerably.
- A good estimate can be obtained by picking three elements randomly and using the median of these three as pivot.
- The randomness turns out not to help much, so the common course is to use as pivot the median of the left, right, and centre elements.
- Example: for the input 8, 1, 4, 9, 6, 3, 5, 2, 7, 0

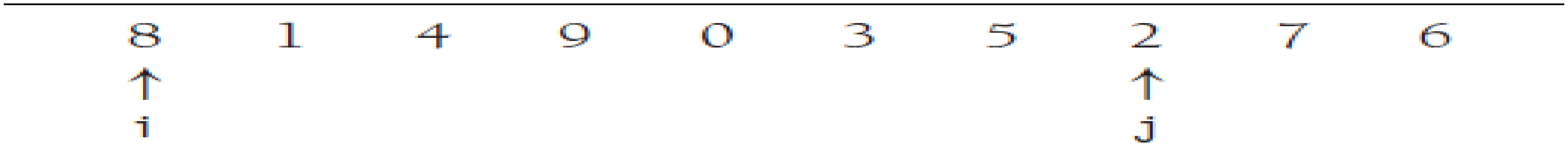
leftElt is 8; rightElt is 0; centerElt is in position floor(left + right)/2) i.e. 6

- Using median-of-three partitioning reduces the number of comparisons by 14%.

Partitioning Strategy

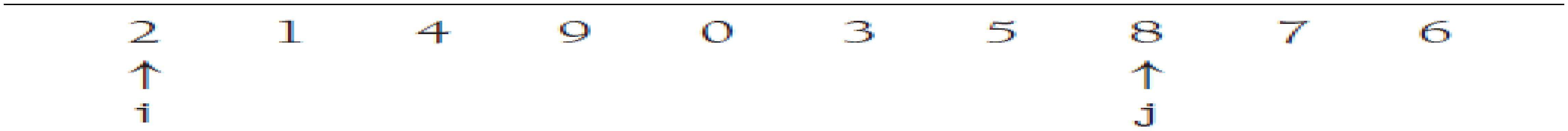
- Several partitioning strategies are used in practice; the one described here is known to give good results.
- The first step is to get the pivot element out of the way by swapping it with the last element.
- *i* starts at the first element and *j* starts at the next-to-last element.
- The partitioning stage wants to move all the small elements to the left part of the array and all the large elements to the right part. (“Small” and “large” are relative to the pivot.)
- While *i* is to the left of *j*, it is moved right, skipping over elements that are smaller than the pivot.
- *j* is moved left, skipping over elements that are larger than the pivot.
- When *i* and *j* have stopped, *i* is pointing at a large element and *j* is pointing at a small element.
- If *i* is to the left of *j*, those elements are swapped



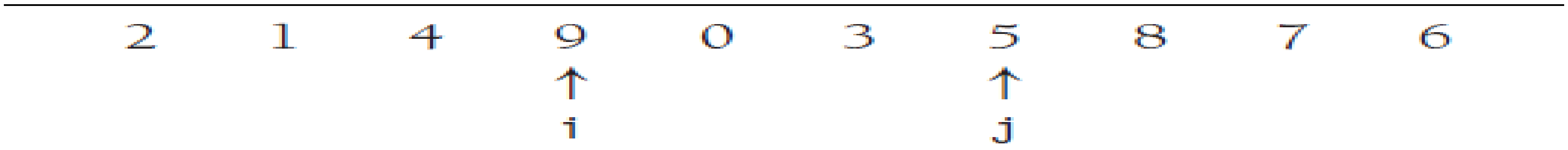


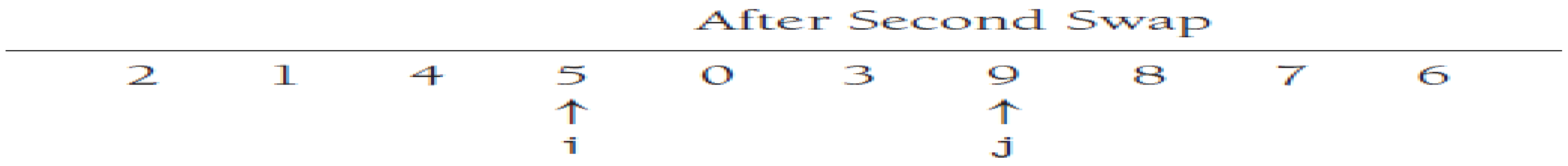
swap the elements pointed to by i and j. Repeat the process until i and j cross.

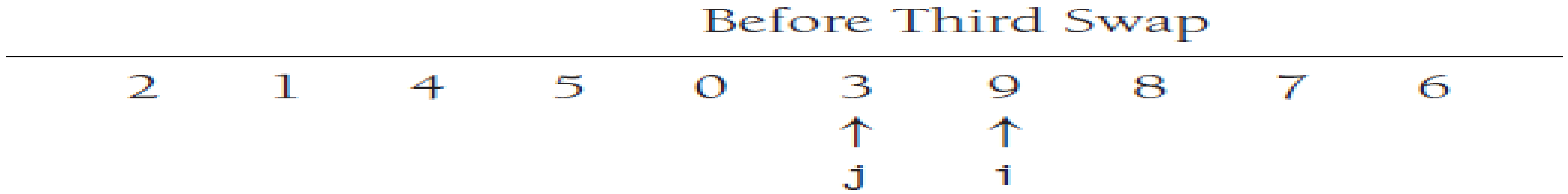
After First Swap



Before Second Swap

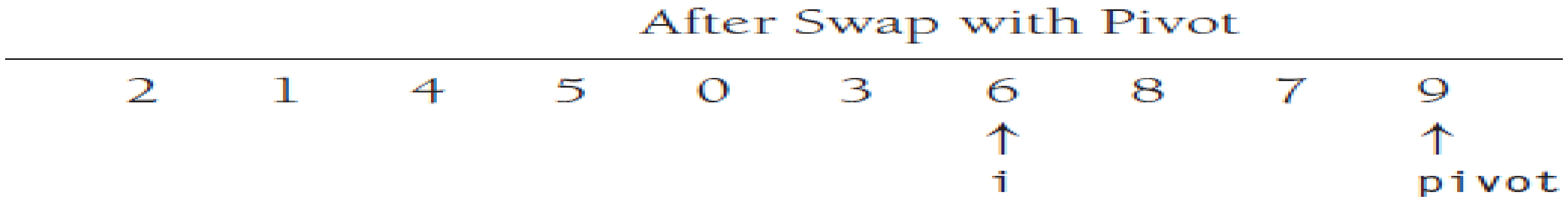






Now, *i* and *j* have crossed, so no swap is performed.

Partitioning final part: swap the pivot element with the element pointed to by *i* :



Handling equal elements

- One important detail we must consider is how to handle elements that are equal to the pivot (➔ will be in S_2).
- Consider the case where all the elements in the array are identical.
- If both i and j stop, there will be many swaps between identical elements.
- Although this seems useless, the positive effect is that i and j will cross in the middle, so when the pivot is replaced, the partition creates two nearly equal subarrays.
- The mergesort analysis tells us that the total running time would then be $O(N \log N)$.

- If neither *i* nor *j* stops, and code is present to prevent them from running off the end of the array, no swaps will be performed.
- Although this seems good, a correct implementation would then swap the pivot into the last spot that *i* touched, which would be the next-to last position (or last, depending on the exact implementation).
  ➔ This would create very uneven subarrays.
- If all the elements are identical, the running time is $O(\mathrm{N}^2)$.
- It is better to do the unnecessary swaps and create even subarrays than to risk wildly uneven subarrays.
- Therefore, we will have both *i* and *j* stop if they encounter an element equal to the pivot.
- This is the only one of the four possibilities that does not take quadratic time for this input.

Handling Small Arrays

- For very small arrays ($N \leq 20$), *quicksort* does not perform as well as *insertionSort*.
- Also, because *quicksort* is recursive, these cases will occur frequently.
- A common solution is not to use *quicksort* recursively for small arrays, but instead use a sorting algorithm that is efficient for small arrays, such as *insertionSort*.
- Using this strategy can actually save about 15 percent in the running time (over doing no cutoff at all).
- A good cutoff range is $N = 10$, although any cutoff between 5 and 20 is likely to produce similar results.

Quicksort routines

Analysis of Quicksort

- **The worst-case bound for quicksort is $O(N^2)$.**
- **Best-Case Analysis gives $\Theta(N \log N)$.**
- **Average-case $O(N \log N)$.**

- For the proofs, we will assume:
  - a random pivot (no median-of-three partitioning) and
  - no cutoff for small arrays
- Obviously, we will take T(0) = T(1) = 1
- Since running time of quicksort is equal to running time of the two recursive calls plus the linear time spent in the partition (the pivot selection takes only constant time), the basic quicksort relation

$$T(N) = T(i) + T(N - i - 1) + cN$$

where $i = |S_1|$ is the number of elements in S_1.

3 cases for the Proofs of the Analyses

Worst-Case Analysis:

- The pivot is the smallest element, all the time. Then $i = 0$, and if we ignore $T(0) = 1$ (insignificant), the recurrence is $T(N) = T(N-1) + cN, \quad N > 1$

Using the basic quicksort relation

$$T(N-1) = T(N-2) + c(N-1)$$

$$T(N-2) = T(N-3) + c(N-2)$$

$$\dots$$

$$T(2) = T(1) + c(2)$$

Adding up all these equations yields

$$T(N) = T(1) + c\sum_{i=2}^{N} i = \Theta(N^2)$$

Best-Case Analysis:

- Best case: the pivot is in the middle.
- We assume that the two subarrays are each exactly half the size of the original (a slight overestimate; but ok for Big-Oh complexity).

$$T(N) = 2\,T(N/2) + c\,N \quad \Rightarrow \quad T(N) / \mathrm{N} = T(N/2) / (N/2) + c$$

Likewise, $T(N/2) / (\mathrm{N}/2) = T(N/4) / (N/4) + c$

$T(N/4) / (\mathrm{N}/4) = T(N/8) / (N/8) + c$

Etc. until $T(2) / 2 = T(1) / 1 + c$

*Adding up all these equations (*there are log N of them), *we get:*

$$T(N) / (\mathrm{N}) = T(1) / 1 + c \log N$$

which yields (same results as mergeSort, Section 7.8):

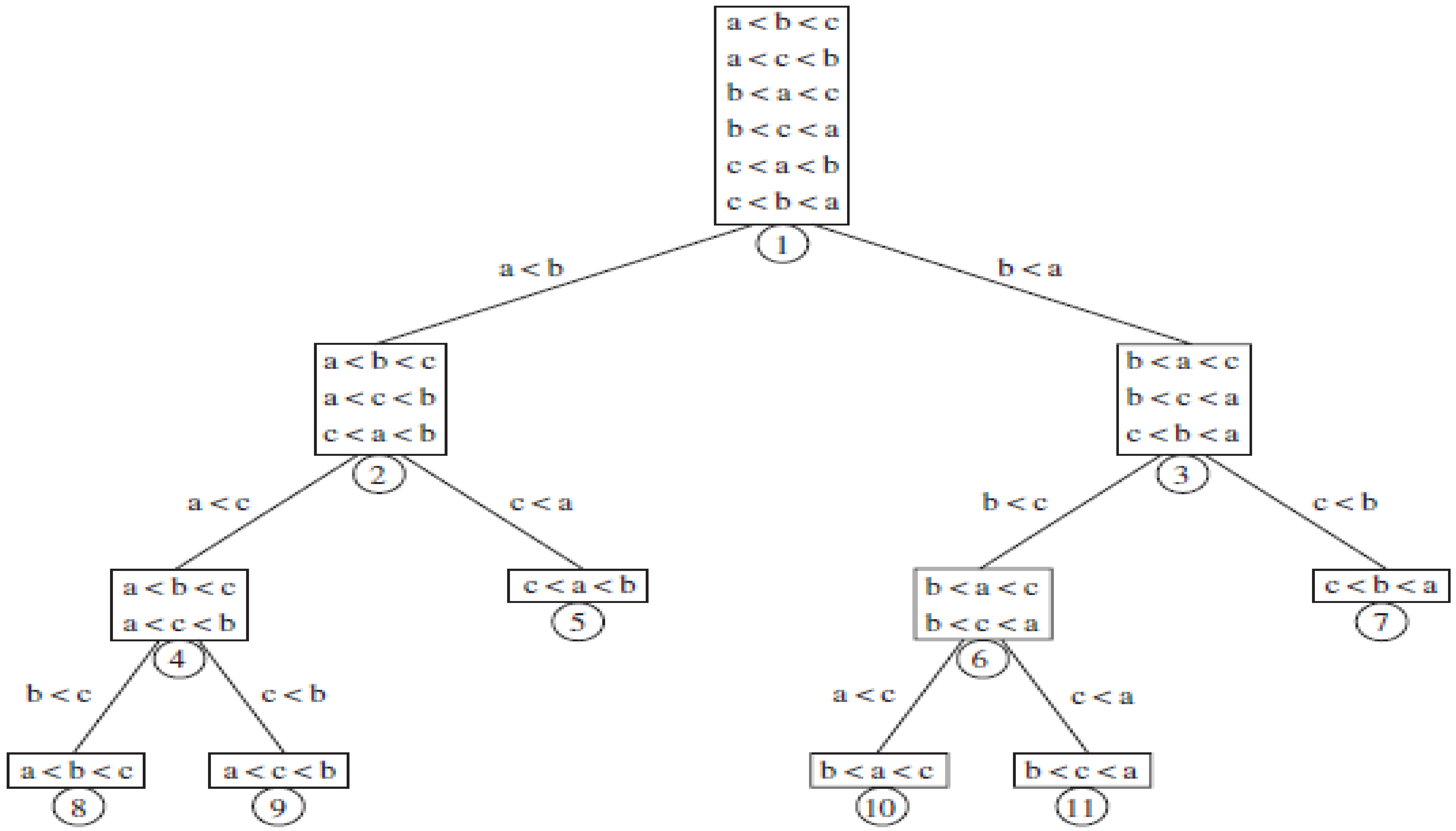
$$T(N) = c\,N \log N + N = \Theta\,(N \log N)$$

Average-Case *Analysis:*

- The analysis starts from equation $T(N) = T(i) + T(N - i - 1) + cN$ seen earlier
- It is a bit lengthier but with simple algebra steps
- It eventually leads to $T(N) = O(N \log N)$

Decision Trees

- In our context, a decision tree is a binary tree.
- Each node represents a set of possible orderings, consistent with comparisons that have been made, among the elements.
- The results of the comparisons are the tree edges.


a < b < c
a < c < b
b < a < c
b < c < a
c < a < b
c < b < a
1
a < b
b < a
a < b < c
a < c < b
c < a < b
2
b < a < c
b < c < a
c < b < a
3
a < c
c < a
b < c
c < b
a < b < c
a < c < b
4
c < a < b
5
b < a < c
b < c < a
6
c < b < a
7
b < c
c < b
a < c
c < a
a < b < c
8
a < c < b
9
b < a < c
10
b < c < a
11

Decision Trees

- Decision trees are used here to prove lower bounds.
- The average number of comparisons used is equal to the average depth of the leaves.
- Since a decision tree is large, it follows that there must be some long paths.

Basic Tree Properties and Lower Bounds

- **Lemma 7.1:** Let *T* be a binary tree of depth *d*. Then *T* has at most 2^d leaves.
- **Lemma 7.2:** A binary tree with *L* leaves must have depth at least *ceiling(log L)*.
- **Theorem 7.6:** Any sorting algorithm that uses only comparisons between elements requires at least *ceiling(log(N!))* comparisons in the worst case.
- **Theorem 7.7:** Any sorting algorithm that uses only comparisons between elements requires *Ω(N log N)* comparisons.

A General Lower Bound for Sorting

- We have $O(N \log N)$ algorithms for sorting; but is it as good as we can do?
- As just stated, any algorithm for sorting that uses only comparisons requires $\Omega(N \log N)$ comparisons (and hence time) in the worst case.

==> So ***mergesort*** and ***heapsort*** are optimal to within a constant factor.

- $\Omega(N \log N)$ comparisons are required, even on average, for any sorting algorithm that uses only comparisons.

==> So ***quicksort*** is optimal on average to within a constant factor.

Linear-Time Sorts: Bucket Sort (programiz.com)

- Bucket Sort divides the unsorted array elements into several groups called buckets.
- Each bucket is then sorted by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm.
- Finally, the sorted buckets are combined to form a final sorted array.
- The process of bucket sort can be understood as a **scatter-gather approach:**
    - elements are first scattered into buckets;
    - then the elements in each bucket are sorted;
    - finally, the elements are gathered in order.

Working of Bucket Sort

Suppose, the input array is:

0.42	0.32	0.23	0.52	0.25	0.47	0.51

Create an array of size 10. Each slot of this array is used as a bucket for storing elements.

0	0	0	0	0	0	0	0	0	0

- Insert elements into the buckets from the array. The elements are inserted according to the range of the bucket.
- Suppose we have buckets each of ranges from 0 to 1, 1 to 2, 2 to 3,......, (n-1) to n.
- Suppose, input element is 0.23. It is multiplied by size = 10 (i.e. 0.23*10 = 2.3).
- Then, it is converted into an integer (i.e. floor(2.3) = 2).
- Finally, 0.23 is inserted into **bucket-2**.
- If integer taken as input, then divide it by the interval (10 here) and take the floor value.

Insert all the elements into the buckets from the array

0	0	0.23	0.32	0.42	0.52	0	0	0	0
		0.25		0.47	0.51				

Sort the elements in each bucket separately using some sorting algorithm (insertionsort, quicksort, etc.)

0	0	0.23	0.32	0.42	0.51	0	0	0	0
		0.25		0.47	0.52				

The elements from each bucket are gathered by iterating through the buckets and copying the bucket elements into the original array. (Each element from the bucket is erased once it is copied into the original array.)

0.23	0.25	0.32	0.42	0.47	0.51	0.52

Analysis of Bucket Sort

- The input $A_1, A_2, \ldots, A_N$ must be only positive integers smaller than M.
- The array of buckets of size M, is initialized to all 0s. So it has M buckets, initially empty.

<u>Worst Case Complexity</u>: ***$O(N^2)$***

- When there are elements of close range in the array, they are likely to be placed in the same bucket. ➔ some buckets may have more elements than others.
  ➔ The complexity will depend on the sorting algorithm used to sort the elements of the bucket.
  The complexity becomes even worse when the elements are in reverse order.
- If insertion sort is used to sort elements of the bucket, then the time complexity becomes ***$O(N^2)$.***

<u>Best Case Complexity</u>: ***O(N+M)***

- When there are elements of close range in the array, they are likely to be placed in the same bucket. ➔ some buckets may have more elements than others.
- It occurs when the elements are uniformly distributed in the buckets with a nearly equal number of elements in each bucket.
- The complexity becomes even better if the elements inside the buckets are already sorted.
- If insertion sort is used to sort elements of a bucket then the overall complexity in the best case will be linear i.e. ***O(N+M)***.

where ***O(N)*** is the complexity for making the buckets and ***O(M)*** is the complexity for sorting the elements of the bucket using algorithms having linear time complexity at the best case.

Average Case Complexity: **O(n)**

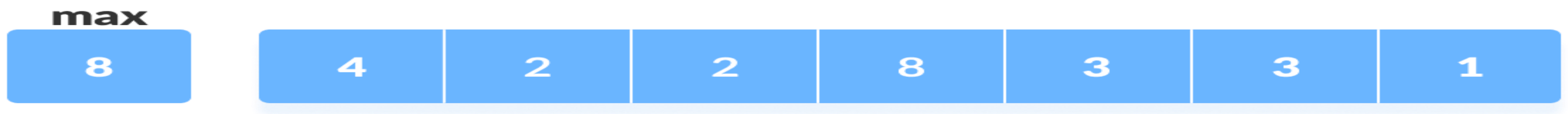
- It occurs when the elements are distributed randomly in the array.
- Even if the elements are not distributed uniformly, bucket sort runs in linear time.
- This remains true until the sum of the squares of the bucket sizes is linear in the total number of elements.

Counting Sort

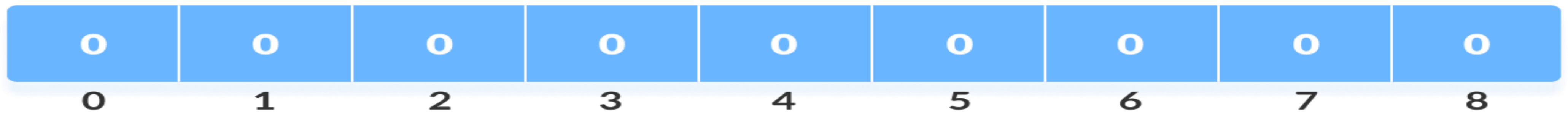
- Sorts the elements of an array by counting the number of occurrences of each unique element in the array.
- The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Working of Counting Sort:

Find out the maximum element, *max*, from the given array.



Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.



Store the count of each element at their respective index in count array

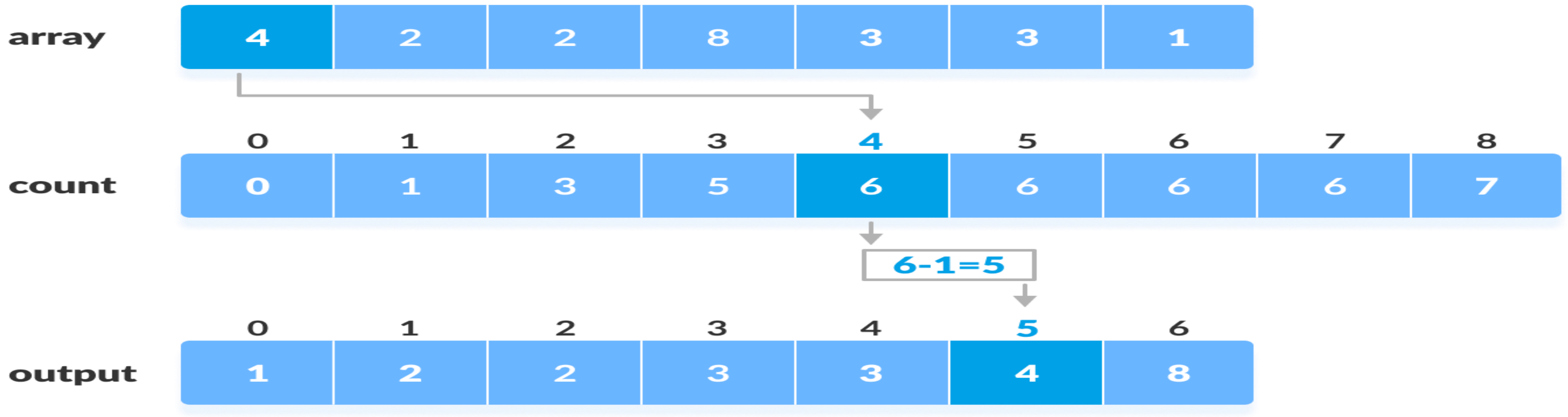
0	1	2	2	1	0	0	0	1
0	1	2	3	4	5	6	7	8

Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.

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

Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.

Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.



After placing each element at its correct position, decrease its count by one.

Counting Sort complexities

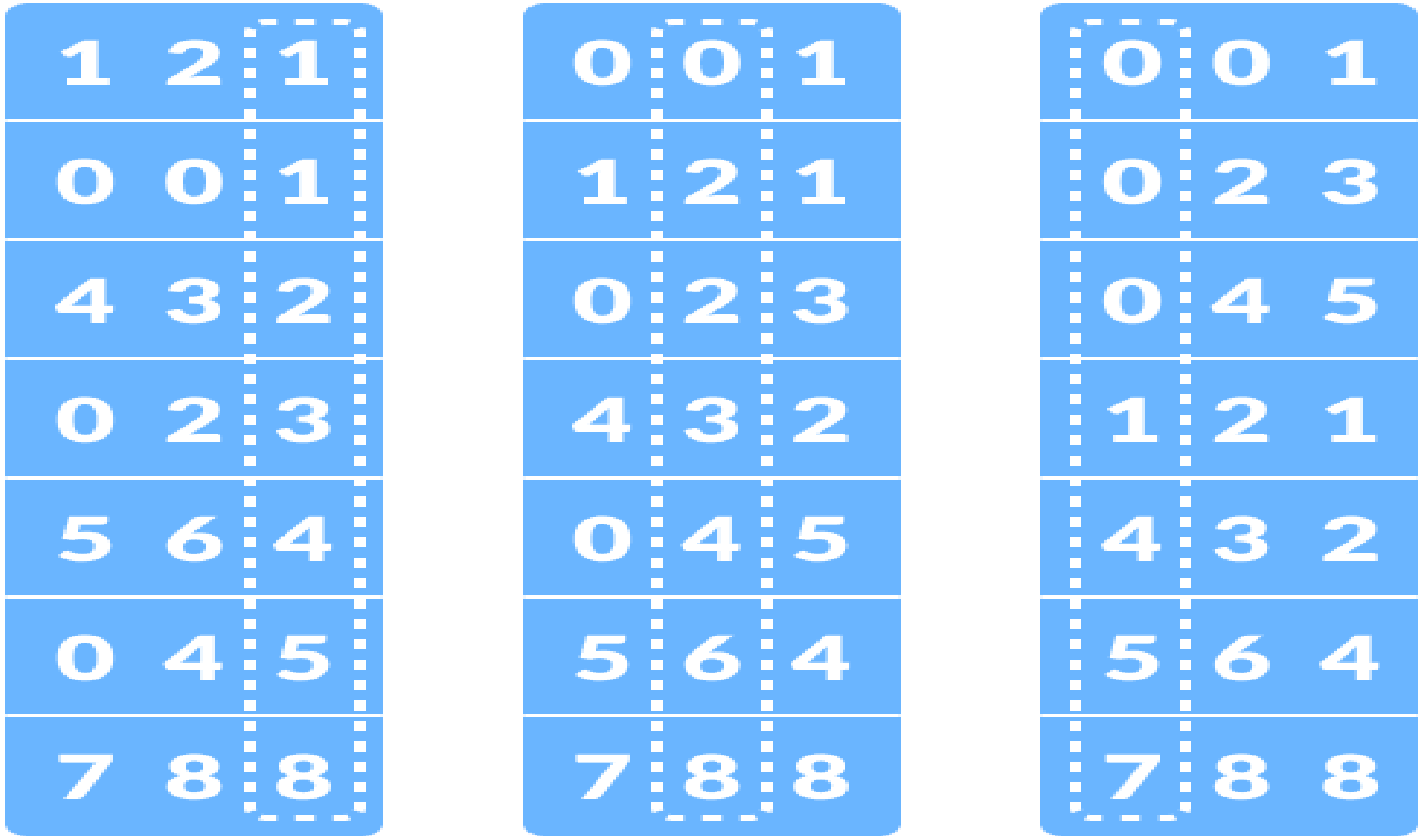
Worst Case Complexity: O(n+k)
Best Case Complexity: O(n+k)
Average Case Complexity: O(n+k)
K is max.

In all the above cases, the complexity is the same because no matter how the elements are placed in the array, the algorithm goes through n+k times.

Linear-Time Sorts: Radix Sort (programiz.com)

Radix sort sorts the elements by

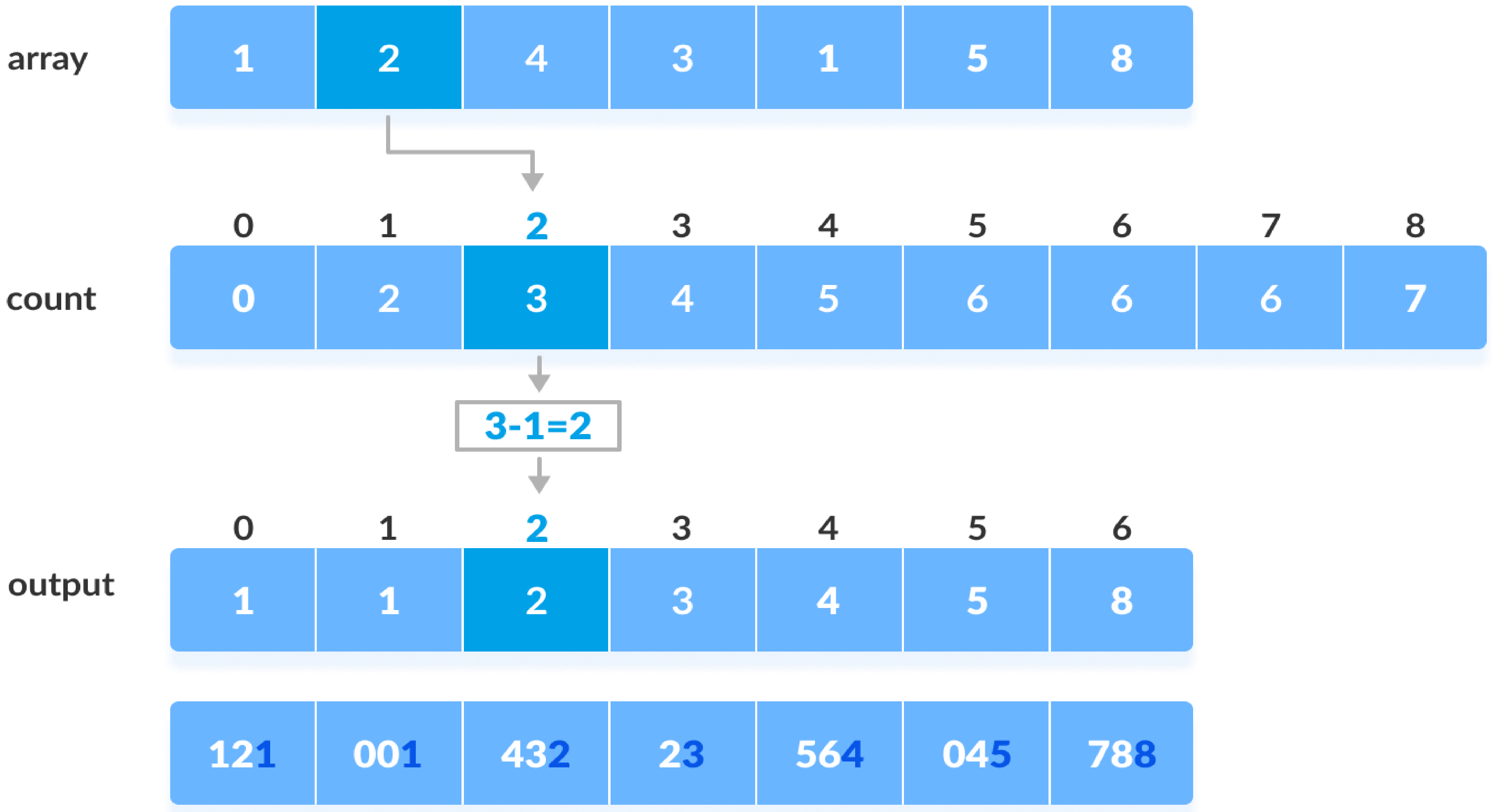
- first grouping the individual digits of the same **place value**,
- then, sorting the elements according to their increasing/decreasing order.



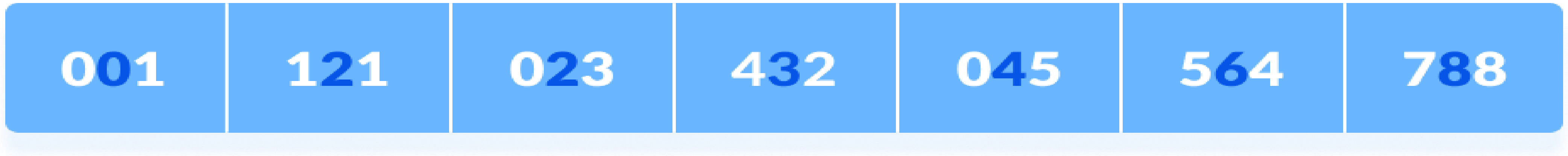
sorting the integers according to units, tens and hundreds place digits

Working of Radix Sort

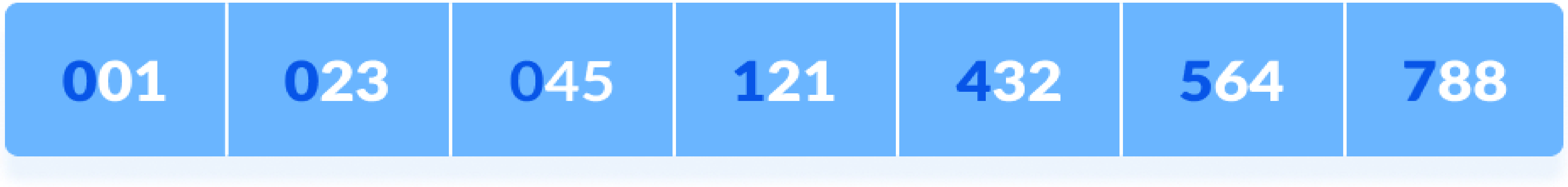
1. Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.
   In example [121, 432, 564, 23, 1, 45, 788], 788 is the largest. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).
2. Now, go through each significant place one by one.
   Use any stable sorting technique to sort the digits at each significant place. E.g. ***counting sort***.
   Sort the elements based on the unit place digits (X=0).


array
1
2
4
3
1
5
8
0
1
2
3
4
5
6
7
8
count
0
2
3
4
5
6
6
6
7
3-1=2
0
1
2
3
4
5
6
output
1
1
2
3
4
5
8
121
001
432
23
564
045
788

Now, sort the elements based on digits at tens place



Finally, sort the elements based on the digits at hundreds place.



Analysis of Radix Sort

- Since ***radix sort*** is a non-comparative algorithm, it has advantages over comparative sorting algorithms.
- For the ***radix sort*** that uses counting sort as an intermediate stable sort, the time complexity is O(d (n+k)). Where d is the number cycle and *O(n+k)* is the time complexity of ***counting sort***.
- ***radix sort*** has linear time complexity which is better than *O(n log n)* of comparative sorting algorithms.
- If we take very large digit numbers or the number of other bases like 32-bit and 64-bit numbers then it can perform in linear time however the intermediate sort takes large space.
- This makes ***radix sort*** space inefficient. This is the reason why this agorithm is not used in software libraries.