Unit 8 Sorting

The efficiency of data handling can be increased if the data are sorted according to some criteria of order. It is often necessary to sort data before processing. We choose some criteria that is used to order data. The choice will vary from application to application and must be defined by the user. Very often, the sorting criteria are natural, as in case of numbers. A set of numbers can be sorted in ascending or descending order.

The final ordering of data can be obtained in a variety of ways, and only some of them can be considered meaningful and efficient. To decide which method is best, certain criteria of efficiency have to be established and a method for quantitatively comparing different algorithms must be chosen

To make the comparison machine independent, certain critical properties of sorting algorithms should be defined when comparing alternative methods. Two such properties are the number of comparisons and the number of data movements. To sort a set of data, the data have to be compared and moved as necessary; the efficiency of these two operations depends on the size of the data set.

Elementary Sorting Algorithms:

1. Insertion Sort
2. Selection Sort
3. Bubble Sort

Efficient Sorting Algorithm:

1. Heap Sort
2. Quick Sort
3. Merge Sort
4. Radix Sort

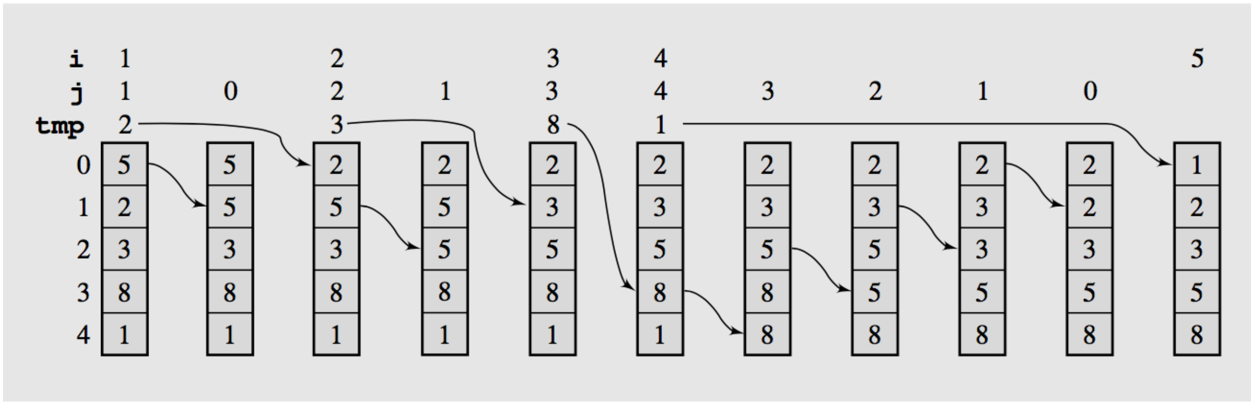
Elementary Sorting Algorithms:

1. **Insertion Sort:** In the insertion sort, elements are inserted into their proper location such that array will be sorted. Assume we have array ARR with 5 elements.
    i. In the first round ARR [0] will be itself sorted.
    ii. In the second round ARR [0] and ARR [1] will be compared and they will be sorted such that the two elements are in order as: ARR [0] <ARR [1].
    iii. In the third round ARR[2] will be inserted among ARR[0] and ARR[1] such that the three will appear in the order as : ARR[0]<ARR[I]<ARR[2].
    iv. In the fourth round ARR[3] will be inserted among ARR[0] to ARR[2] such that the four will appear in order as : ARR[0] <ARR[1]<ARR[2]<ARR[3].
    v. Finally in the last round ARR[4] will be inserted among ARR[0] to ARR[3] such that the five will appear in order as : ARR[0]<ARR[1]<ARR[2]<ARR[3]>ARR[4].

Thus in every pass, proper location of the element to be inserted is found and element is inserted at that location. To better understand let’s sort the following numbers using insertion sort. The Java method for insertion sort is as follows:

```
void Insertion_sort (int [] data)

{
            int i, j, temp;
            for (i=1; i<data.length; i++)
            {
                        temp = data[i];
                        for (j=i; j>0 && temp<data[j-1]; j--)
                                    data[j] = data[j-1];
                        data[j] = temp;
            }
}
```



Complexity of Insertion Sort

The outer for loop runs for N-1 number of times. When the array is already sorted in reverse order then there will be maximum number of comparisons (N-1). In first iteration number of comparisons will be 1, in the second iteration it will be 2, up to N-1,so total number of comparisons will be:

1+2+3+4+………………….(N-1) = N*(N-1)/2.

Therefore complexity of insertion sort in worst case: O(N^2)

Now if the array is already sorted then comparison will be done but no interchange will be taking place and insertion sort will run in linear time i.e. complexity of insertion sort when array already sorted is: O(N). This is the best case for insertion sort.

After the discussion of worst case and best case, we consider average case. On an average there will be (N-1)/2 comparisons, so total number of comparisons will be:

1+2+3+4+……..+(N-1)/2 = N(N-1)/4

Therefore complexity of insertion sort in average case: O(N^2)

Sort the following set of data items : 54 , 26 , 93 , 17 , 77 , 31 , 44 , 55

Steps	Elements in array								Remarks
1	**54**	26	93	17	77	31	44	55	A[0] is at proper position
2	**26**	**54**	93	17	77	31	44	55	A[0] and A[1] are sorted
3	**26**	**54**	**93**	17	77	31	44	55	A[0] ,A[1] and A[2] are sorted
4	**17**	**26**	**54**	**93**	77	31	44	55	A[0]] through A[3] are sorted
5	**17**	**26**	**54**	**77**	**93**	31	44	55	A[0] through A[4] are sorted
6	**17**	**26**	**31**	**54**	**77**	**93**	44	55	A[0] through A[5] are sorted
7	**17**	**26**	**31**	**44**	**54**	**77**	**93**	55	A[0] through A[6] are sorted
8	**17**	**26**	**31**	**44**	**54**	**77**	**93**	**55**	A[0] through A[7] are sorted

Advantage:

An advantage of using insertion sort is that it sorts the array only when it is really necessary. If the array is already in order, no substantial moves are performed; only the variable tmp is initialized, and the value stored in it is moved back to the same position. The algorithm recognizes that part of the array is already sorted and stops execution accordingly.

Disadvantages:

(i) The fact that elements may already be in their proper positions is overlooked.
(ii) If an item is being inserted, all elements greater than the one being inserted have to be moved.

2. Selection Sort

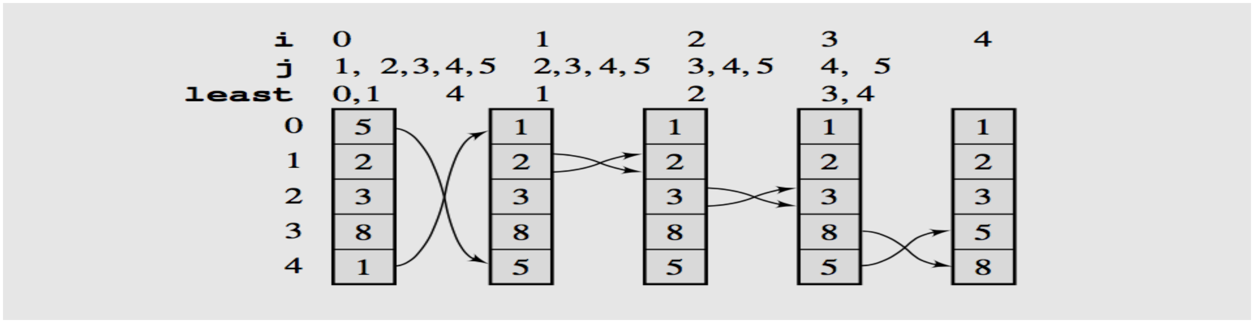
The idea of algorithm is quite simple. Array is imaginary divided into two parts- sorted and unsorted one. At the beginning, sorted part is empty, while unsorted one contains whole array. At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes empty, algorithm stops.

The pseudocode for Selection sort is:

```
Selection_sort (int data[])
for i = 0 to data.length-2
    select the smallest element among data[i], ..., data[data.length-1];
    swap it with data[i];
```

Selection Sort Implementation:

```
void selectionsort(int [] data)
{
        int i, j, least, temp;
        for (i=0; i<data.length-1; i++)
        {
                for (j=i+1, least=i; j<data.length; j++)
                        if(data[j] < (data[least]))
                                least = j;
                if(i != least)
                {
                        swap(data, least, i);
                }
        }
}
```



Complexity of Selection Sort

The for loop runs N-1 times in the function selection sort. For every minimum selected for the array or segment of array just one swapping operation is performed. This is because it is not placed inner for loop. The function min requires N-1 comparisons during first iteration of the for loop, N-2 in the second iteration and so on. Thus total number of comparisons:

N-1+N-2+……+3+2+1 = N(N-1)/2

Thus complexity of the selection sort is O(N^2). If we consider array in sorted or reverse order then this does not affect number of comparisons in the function min. Thus number of comparisons in selection sort is independent of the original order of the elements in the array. Moreover, the complexity of bubble sort and selection sort is same but selection sort executes faster than bubble sort. This is because of less number of swapping operations.

3. Bubble Sort

The bubble sort is the easiest and frequently used sorting algorithm among all the sorting algorithms. The algorithm has got its name as after every pass, the largest element bubbles up and move to end of the array. The logic for bubble sort is as follows.

1. Start comparing a[0] to a[1]
2. If a[0]>a[1] then swap numbers.
3. Compare a[1] to a[2] and repeat till you compare last pair
4. This is referred to as one pass and at the end of first pass largest number is at last
5. Repeat this comparison again starting from a[0] but this time going till second last pair only.

Pseudocode of Bubble sort algorithm:

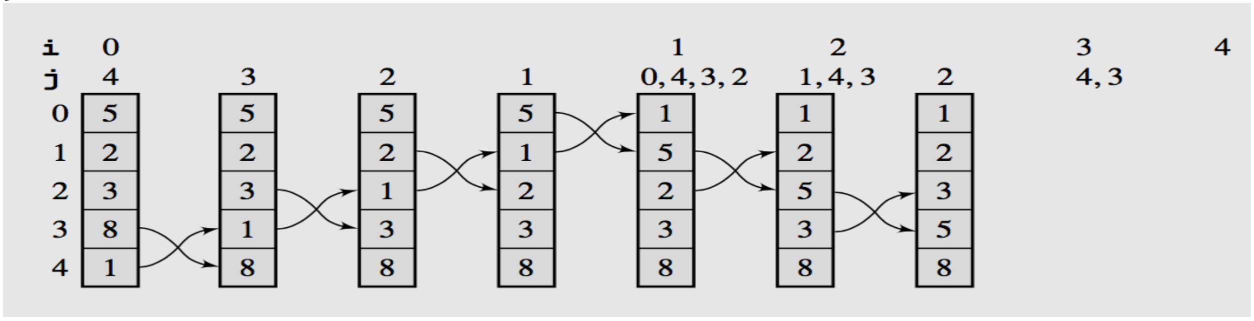
```
bubblesort(data[])
for i = 0 to data.length-2
    for j = data.length-1 downto i+1
        swap elements in positions j and j-1 if they are out of order;
```

The Java Implementation for bubble sort:

```
void bubblesort(int [] data)
{
        for (int i = 0; i < data.length-1; i++)
                for (int j = data.length-1; j > i; --j)

                        if (((Comparable)data[j]).compareTo(data[j-1]) < 0)

                                swap(data, j, j-1);
}
```



Complexity of Bubble Sort

For any array of N elements there are N-1 passes (outer for loop) and in each pass there are N-1 comparisons. Therefore total number of comparisons are (N-1)*(N-1 = N^2+2N+1. This is of the order O(N^2). This complexity is in general not fixed. This is because number of comparisons may differ if the input array is already sorted or sorted in reverse order.

Efficient Sorting Algorithms:

The O(n^2) limit for a sorting method is much too large and must be broken to improve efficiency and decrease run time. The problem is that the time required for ordering an array by three sorting algorithms (insertion, selection, bubble) usually grows faster than the size of the array. In fact, it is customarily a quadratic function of that size. For that we use Heap sort, Quick Sort, Merge Sort, Radix Sort and Shell Sort.

Heap Sort

Selection sort makes O(n^2) comparisons and is very inefficient, especially for large n. But it moves relatively few moves. If the comparison part of the algorithm is improved, the end result can be promising.

Heap sort was invented by John Williams and uses the approach inherent to selection sort. Selection sort finds among the n elements the one that precedes all other n-1 elements, then the least element among those n-1 items, and so forth, until the array is sorted. To have the array in ascending order, heap sort puts the largest element at the end of the array, then the second largest in front of it, and so on. Heap sort starts from the end of the array by finding the largest elements, whereas selection sort starts from the beginning using the smallest element. The final order in both cases is indeed the same.

A heap is a binary tree with the following two properties.

- The value of each node is not less than the values stored in each of its children.
- The tree is perfectly balanced and the leaves in the last level are all in the leftmost positions.

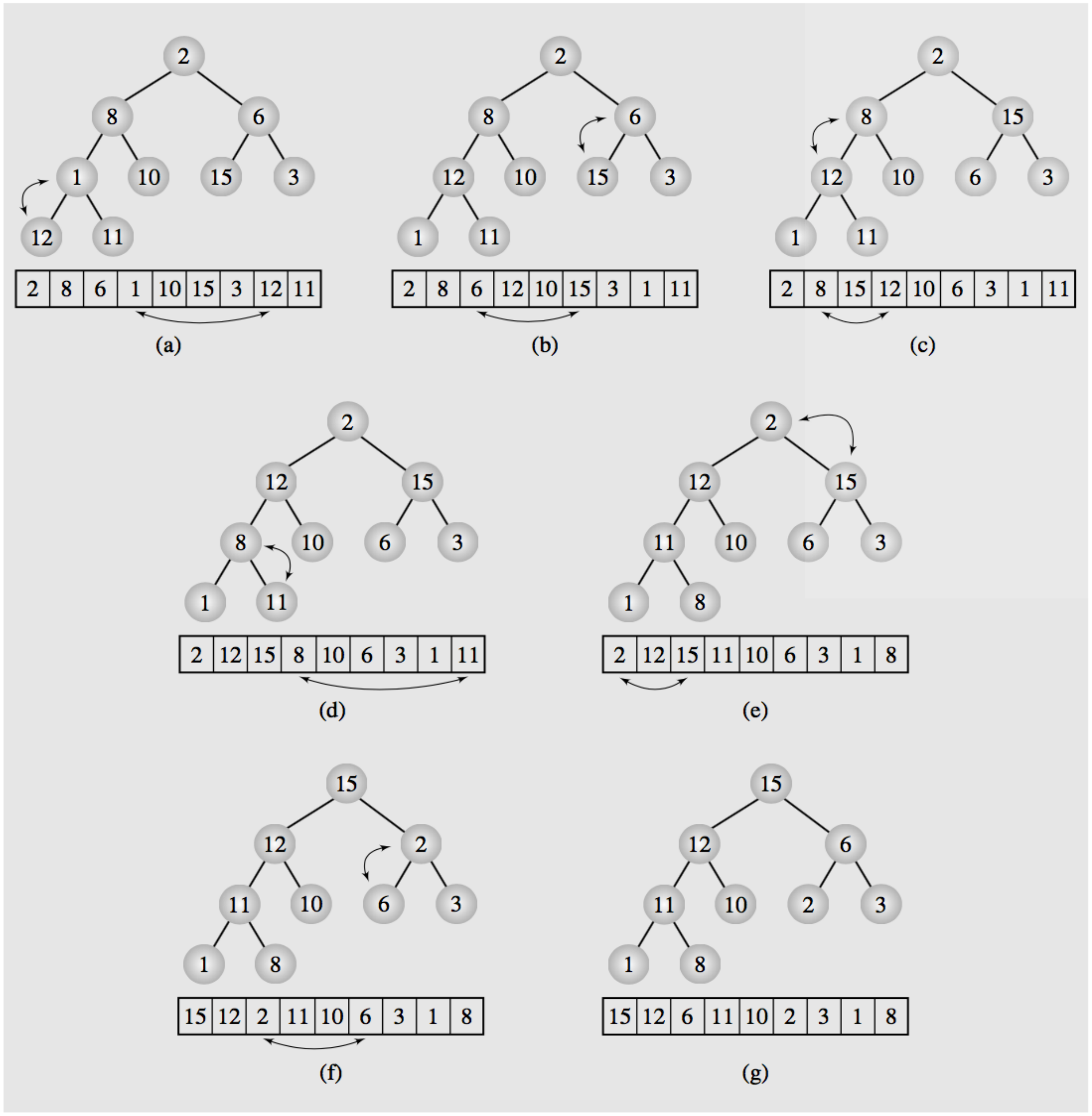
A tree has the heap property if it satisfies condition 1. Both conditions are useful for sorting, although this is not immediately apparent for the second condition. The goal is to use only the array being sorted without using additional storage for the array elements; by condition 2, all elements are located in consecutive positions in the array starting from position 0, with no unusual position inside the array. In other words, condition 2 reflects the packing of an array with no gaps.

Elements in a heap are not perfectly ordered. It is known only that the largest element is in the root node and that, for each other node, all its descendants are not greater than the element in this node. Heap sort thus starts from the heap, puts the largest element at the end of the array, and restores the heap that now has one less element. From the new heap, the largest element is removed and put in its final position and then the heap property is restored for the remaining elements. Thus, in each round, one element of the array ends up in its final position, and the heap becomes smaller by this one element. The process ends with exhausting all elements from the heap.

Pseudocode for Heap Sort

```
heapsort (data [])
    transform data into a heap;
    for i = data.length-1 downto 2
        swap the root with the element in position i;
        restore the heap property for the tree data[0],...,data[i-1];
```

Array into heap transformation



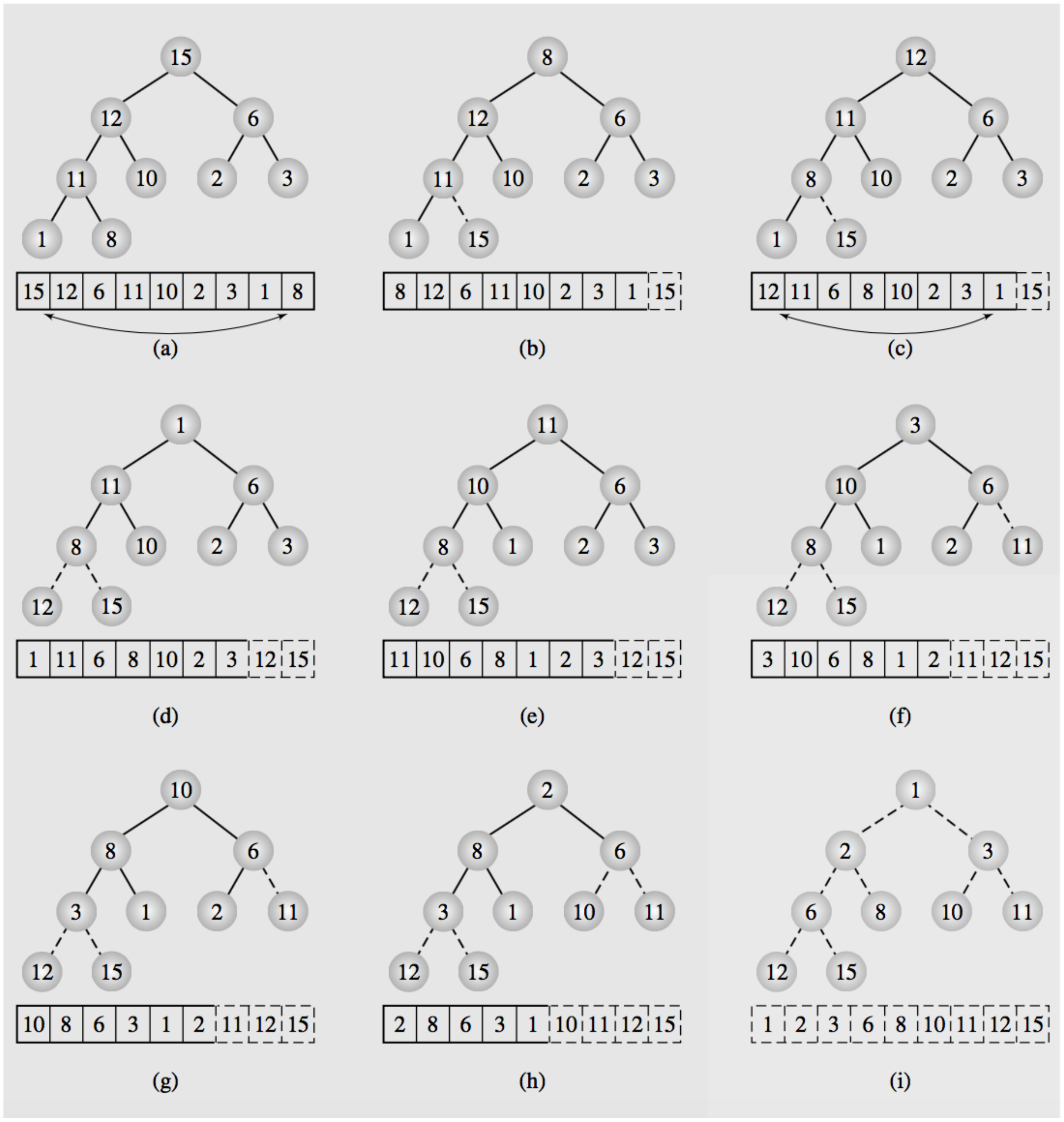


Fig: Heap Sort

Complexity of Heapsort

Complexity of function heapify is O($\log_2$N). This is because total running time of heapify depend upon sub tree rooted at one of the children at index I and setting the property of heap among parent i, left and right of I (constant time O(1). The number of nodes in the children's sub tree can at most be 2 N/3. Thus total running time T(N) can be denoted as:

T(N) = T(2N/3)+O(1)

Solution of the above is $O(\log_2 N)$.

The Buildheap function takes O(n). Function Heapsort used Buildheap and heapify function. A call to buildheap function takes O(N) time and there are N-1 calls to heapify that takes O(log2N) time. Therefore running time of Heapsort is O(NlogN).

Quick Sort

The quick sort divides the original array into two subarrays, the first of which contains elements less than or equal to a chosen key called pivot or bound. The second subarray includes elements equal to or greater than the bound. The two subarrays can be sorted separately but before this is done, the partition process is repeated for both subarrays. As a result, two new bounds are chosen, one for each subarray. The four subarrays are created because each subarray obtained in first phase is now divided into two segments. This process of partitioning is carried down until there are only one cell arrays that do not need to be sorted at all.

To partition an array, two operations have to be performed. As bound has to be found and the array has to be scanned to place the elements in the proper subarrays. However, choosing a good bound is not a trivial task. The problem is that the subarrays should be approximately the same length. If an array contains the numbers 1 through 100 (in any order) and 2 is chosen as a bound, then an imbalance results. The first subarray contains only 1 number after portioning, whereas the second has 99 numbers.

Quick sort is recursive in nature because it is applied to both subarrays of the array at each level of partitioning. The technique is summarized in the following pseudo code:

```
algo quicksort ( a[], lb,ub)
{
        if(lb < ub)
        {
                loc = partition(a, lb, ub);
                quicksort(a, lb, loc-1);
                quicksort(a, loc+1, ub);
        }
}

algo partition(a[], lb,ub)
{
        pivot = a[lb];
        start = lb; end = ub;
        while(start < end)
        {
                while(a[start] <= pivot && start < end)
                {
                        start = start + 1;
                }
                while(a[end] > pivot)
                {
                        end = end – 1;
                }
                if(start < end)
                {
                        swap(a[start], a[end]);
                }
        }
        a[lb] = a[end];
        a[end] = pivot;
        return end;
}
```

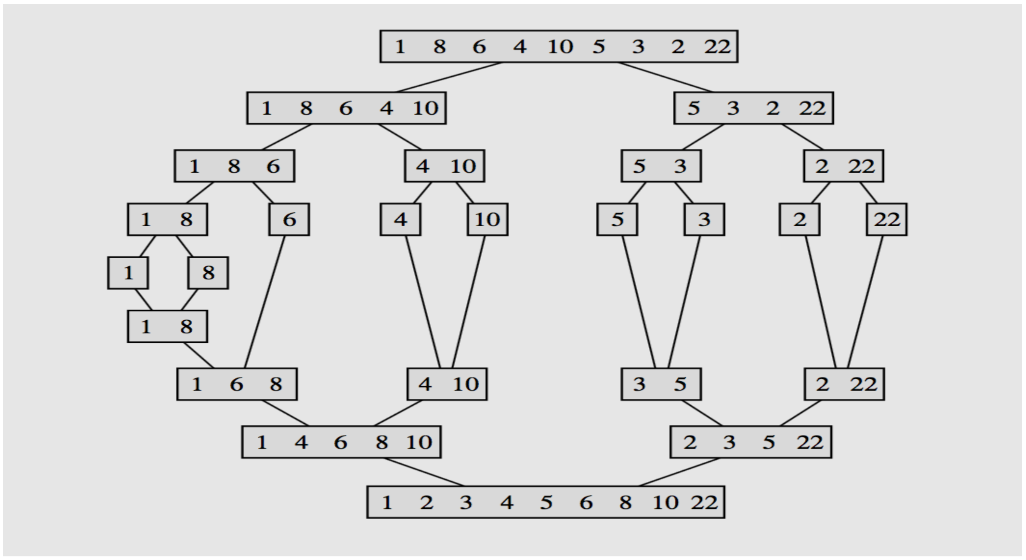
Merge sort

The problem with quick sort is that its complexity in the worst case is O(n^2) because it is difficult to control the partitioning process. Different methods of choosing a bound attempt to make the behavior of this process fairly regular; however, there is no guarantee that portioning results in arrays of approximately the same size. Another strategy is to make portioning as simple as possible and concentrate on merging the two sorted arrays. This strategy is characteristic of merge sort. It was one of the first sorting algorithms used on a computer was developed by John von Neumann.

The key process in merge sort is merging sorted halves of an array into one sorted array. However, these halves have to be sorted first, which is accomplished by merging the already sorted halves of these halves. The process of dividing arrays into two halves stops when the array has fewer than two elements. The algorithm is recursive in nature and can be summarized in the following pseudocode:

```
algo mergesort(data, first, last)
{
        if (first<last)
        {
                mid = (first+last)/2;
                mergesort(data, first, mid);
                mergesort(data, mid+1, last);
                merge(data, first, last);
        }
}
```

The array [1 8 6 4 10 5 3 2 22] sorted by merge sort as follows:



Radix Sort

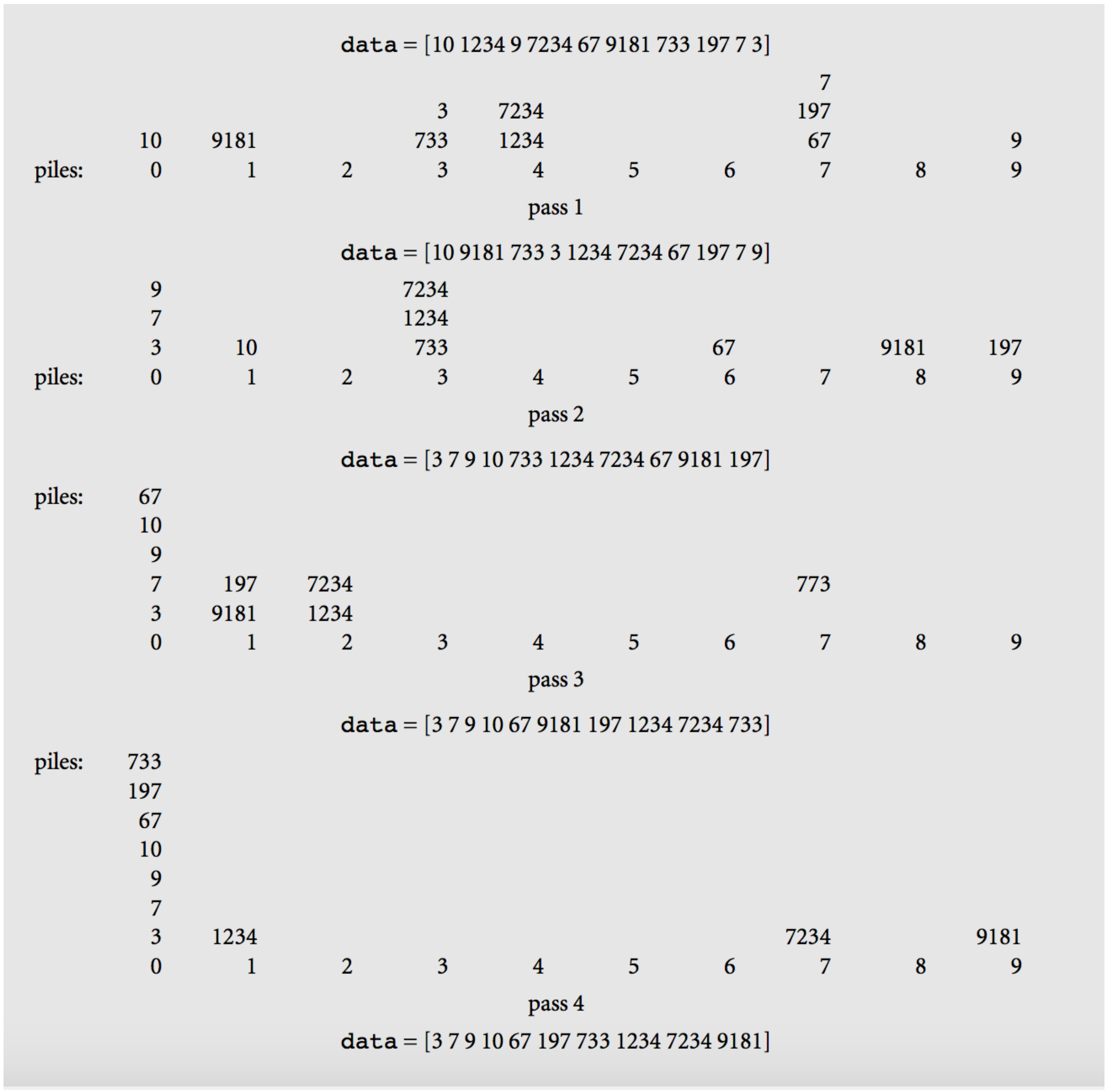
Radix sort is a popular way of sorting used in everyday life. To sort library cards, we may create as many piles of cards as letters in the alphabet, each pile containing authors whose names start with the same letter. Then, each pile is sorted separately using the same method; namely piles are created according to the second letter of the authors' name. This process continues until the number of times the piles are divided into smaller piles equals the number of letters of the longer name.

When sorting integers, 10 piles numbered 0 through 9 are created, and initially, integers are put in a given pile according to their rightmost digit so that 93 is put in pile 3. Then piles are combined and the process is repeated, this time with the second rightmost digit; in this case, 93 ends up on pile 9. The process ends after the leftmost digit of the longest number is processed.

Pseudocode:

```
algo Radixsort(a[], n)
{
        maxdata = getMax(a[], n);
        for(exp= 1; maxdata/exp > 0; exp= exp*10);
                countsort(a[], n, exp);
}

algo countsort(a[], n, exp)
{
        initialize count[0….9} with 0s;
        for(i= n-1; i> 0; i--)
        {
                output[count{a[i]/ exp) % 10] = a[i];
        }
}
```



```
                        data = [10 1234 9 7234 67 9181 733 197 7 3]
                                                          7
                                  3    7234              197
          10    9181             733   1234               67          9
piles:     0       1      2       3      4      5      6   7      8   9
                                     pass 1
                        data = [10 9181 733 3 1234 7234 67 197 7 9]
           9                     7234
           7                     1234
           3      10              733                67         9181  197
piles:     0       1      2       3      4      5      6   7      8   9
                                     pass 2
                        data = [3 7 9 10 733 1234 7234 67 9181 197]
piles:    67
          10
           9
           7     197   7234                                 773
           3    9181   1234
           0       1      2       3      4      5      6   7      8   9
                                     pass 3
                        data = [3 7 9 10 67 9181 197 1234 7234 733]
piles:   733
         197
          67
          10
           9
           7
           3    1234                                       7234       9181
           0       1      2       3      4      5      6   7      8   9
                                     pass 4
                        data = [3 7 9 10 67 197 733 1234 7234 9181]
```

To better understand the radix sorting we sort the following number.

001, 234, 456 , 654, 697, 874, 243, 385, 902, 023

B0	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**
	001	902	243	234	385	456	697		
			023	654					
				874					

Pass 1, placing number in bucket on the basis of unit digit

B0	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**
	001	023	234	243	654		874	385	697
	902				456				

Pass 2, placing number in bucket on the basis of 10th digit

B0	**B1**	**B2**	**B3**	**B4**	**B5**	**B6**	**B7**	**B8**	**B9**
001		234	385	456		654		874	902
023		243				697			

Pass 3, placing number in bucket on the basis of 100th digit

Complexity of Radix Sort: The complexity of radix sort depends upon number of elements (N) and number of digits in the maximum element of the array (say M). It is clear that number of passes will be equal to M. As outer loop runs for M times and inner loop runs for N times, the running time of the radix sort can be approximated to O(N*M). Now in worst case if M=N then running time will be $O(N^2)$ and in case M = log10N, then running time will be $O(N\log_{10}N)$.