Sorting Algorithms

Sorting

- Rearranging elements of an array in order.
- Various types of sorting are:
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
  - Quick Sort
  - Merge Sort
  - Heap Sort

(We will consider increasing order in these slides)

Sorting

- **In-Place:**

A sorting algorithm is called in-place if it does not use extra memory. It requires only a constant amount (i.e. O(1)) of extra space during the sorting process.

- **Out-Place:**

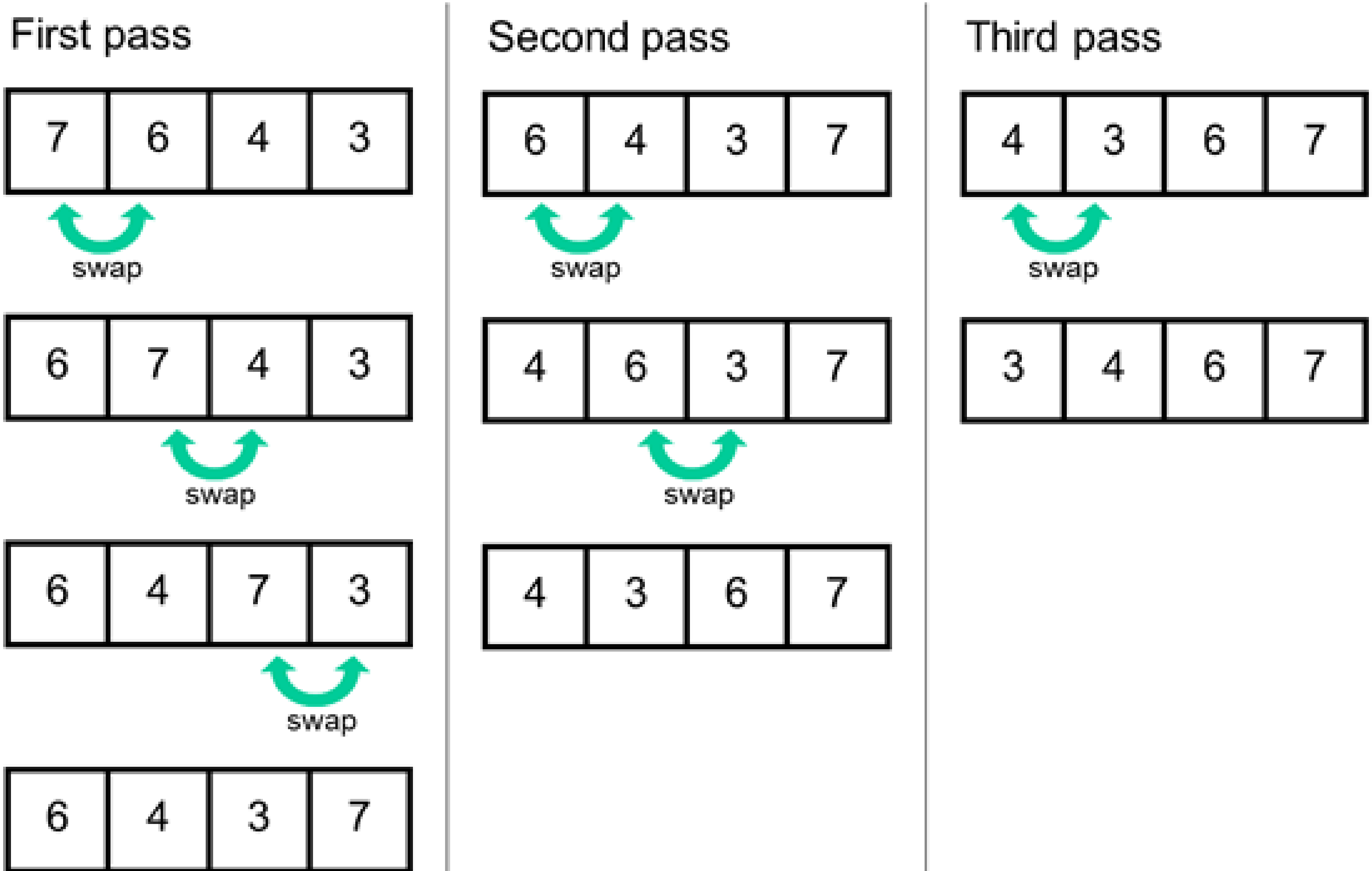
A sorting algorithm is called out-place if it uses extra memory e.g. extra arrays, to sort the given array.

Sorting

- **Stable Sorting:** A sorting algorithm is stable if the relative order of elements with the same key value is preserved by the algorithm.
- Example application of Stable Sort:
  - Assume that names have been sorted in alphabetical order
  - Now, if this list is sorted again by tutorial group number, a stable sort algorithm would ensure that all students in the same tutorial groups still appear in alphabetical order of their names
- **Unstable Sorting:** A sorting algorithm is unstable, if the relative order of elements with the same key value is NOT preserved by the algorithm.

Bubble Sort

Bubble Sort Logic



Algorithm – Bubble Sort

Algorithm bubbleSort(A,n)
Input: An array **A** containing **n** integers.
Output: The elements of **A** get sorted in increasing order.

		cost	time
1.	for i = 1 to n – 1 do	c1	n
2.	for j = 0 to n – i – 1 do	c2	$\sum_{i=1}^{n-1} t_i$
3.	if A[j] > A[j + 1]	c3	$\sum_{i=1}^{n-1} (t_i - 1)$
4.	Exchange A[j] with A[j+1]	c4	$\sum_{i=1}^{n-1} (t_i - 1)$

In all the cases, complexity is of the order of n^2.

Algorithm – Optimized Bubble Sort

Algorithm bubbleSortOpt(A,n)
Input: An array **A** containing **n** integers.
Output: The elements of **A** get sorted in increasing order.

```
for  i = 1  to  n – 1
    flag = true
    for j = 0  to  n – i – 1  do
        if A[j] > A[j + 1]
            flag = false
            Exchange A[j] with A[j+1]
    if  flag == true
        break;
```

The best case complexity reduces to the order of n, but the worst and average is still n^2. So, overall the complexity is of the order of n^2 again.

Example – Bubble Sort

```
#include<stdio.h>
void bubbleSortOpt(int a[],int n);
int main()
{
   int a[10];
   printf("Enter 10 numbers: \n");
   for(int i = 0; i < 10; i++)
          scanf("%d",&a[i]);
   bubbleSortOpt(a,10);
   printf("\n");
   for(int i = 0; i < 10; i++)
        printf("%d ",a[i]);
   return 0;
}
```

Example – Bubble Sort

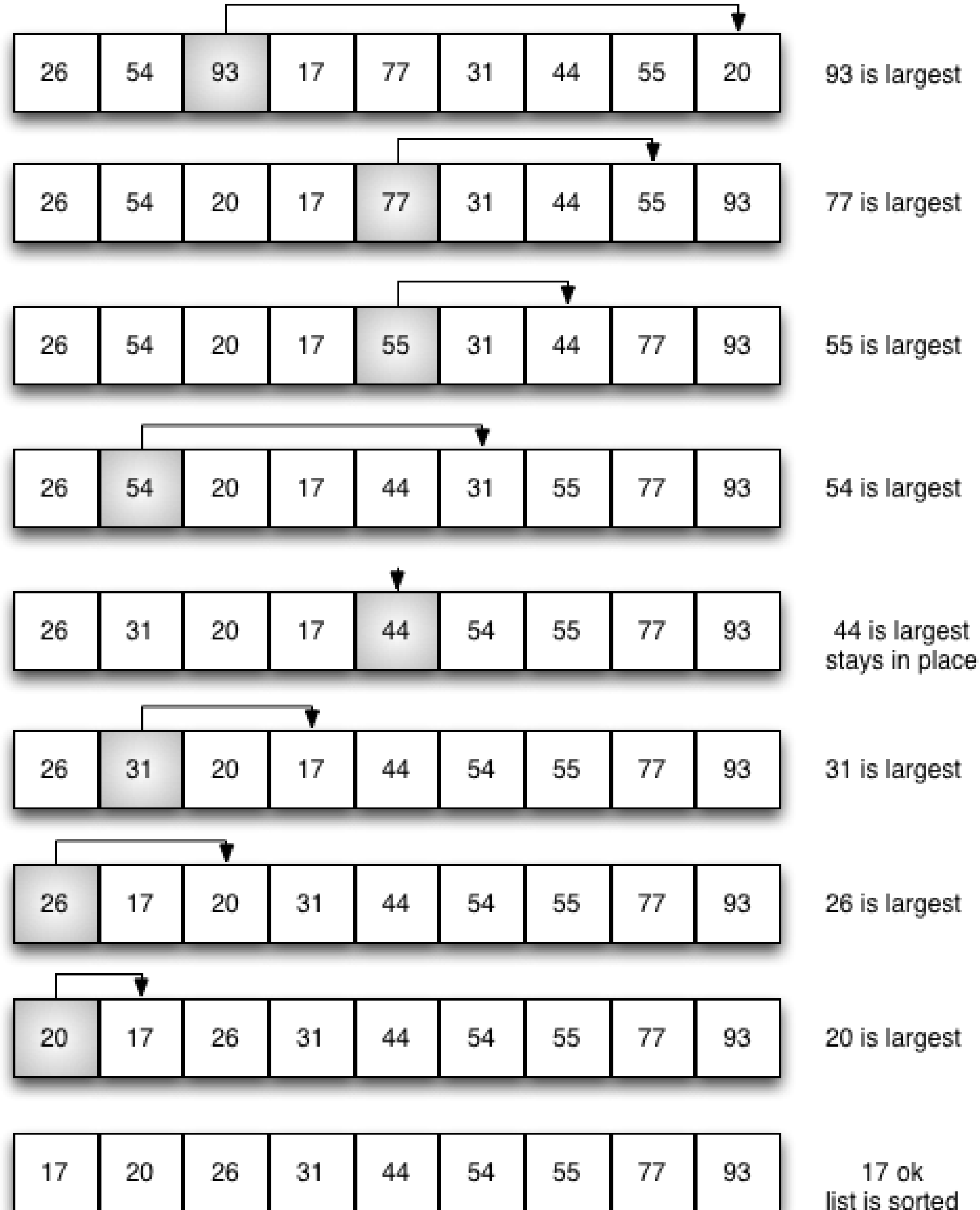
```
void bubbleSortOpt(int a[], int n)
{  int i, j, flag;
   for (i = 1; i <= n-1; i++)
       {     flag = 1;
             for (j = 0; j <= n-1-i; j++)
             {       if (a[j] > a[j+1])
                     {       flag = 0;
                             a[j] = a[j] + a[j+1];
                             a[j+1] = a[j] - a[j+1];
                             a[j] = a[j] - a[j+1];
                     }
             }
             if(flag) break;
      }         }
```

Selection Sort

Selection Sort

- In-place comparison-based algorithm.
- Divides the list into two parts
  - The sorted part, which is built up from left to right at the front (left) of the list, and
  - The unsorted part, that occupy the rest of the list at the right end.
- The algorithm proceeds by
  - Finding the smallest (or the largest) element in the unsorted array
  - Swapping it with the leftmost unsorted element
  - Moving the boundary one element to the right.
  - This process continues till the array gets sorted.
- Not suitable for large data sets.
- Complexity is $\boldsymbol{O(n^2)}$, where $\boldsymbol{n}$ is the number of elements.

Example:
Each pass places the largest element in its proper location.



Algorithm

- **Algorithm selectionSort(a[], n)**
- Input: An array **a** containing **n** elements.
- Output: The elements of **a** get sorted in increasing order.

```
for i = 0 to n – 2
        min = i
        for j = i+1 to n – 1
                if a[j] < a[min]
                        min = j
        if min != i
                Exchange a[min] with a[i]
```

Implementation

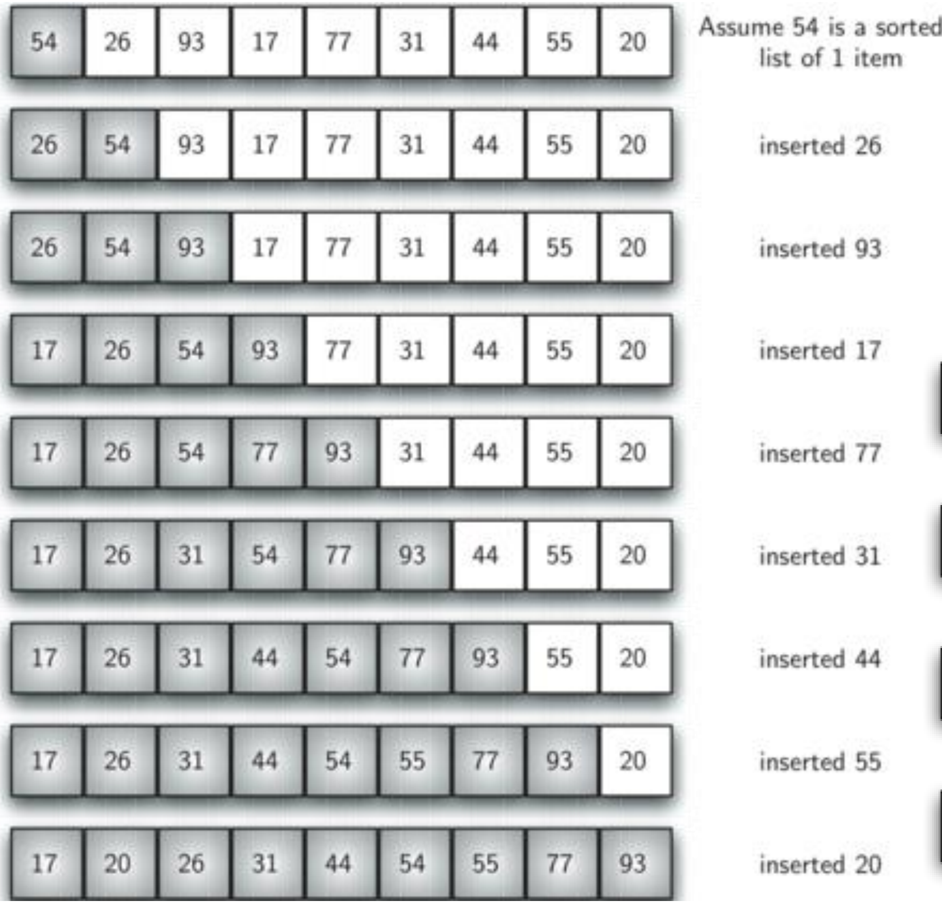
```
#include<stdio.h>
void selectionSort(int a[], int n)
{   int i, j, min, temp;
    for(i=0; i <= n-2; i++ )
    {   min = i;
        for(j=i+1; j <= n-1; j++)
            if(a[j] < a[min])
                min = j;
        if(min != i)
        {  temp = a[i];
            a[i] = a[min];
            a[min] = temp;
        }
    }           }
int main()
{   int a[10];
    printf("Enter 10 numbers: \n");
    for(int i = 0; i < 10; i++)
       scanf("%d",&a[i]);
    selectionSort(a,10);
    printf("\n");
    for(int i = 0; i < 10; i++)
       printf("%d ",a[i]);
    return 0;
}
```

Insertion Sort

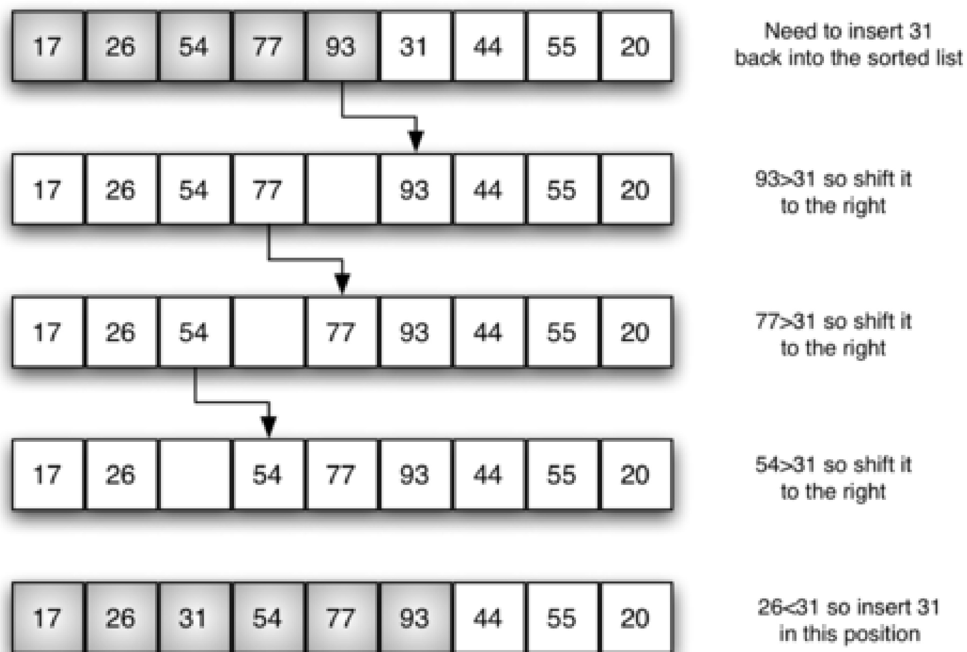
Insertion Sort

- An in-place comparison-based sorting algorithm.
- Always keeps the lower part of an array in the sorted order.
- A new element will be inserted in the sorted part at an appropriate place.
- The algorithm searches sequentially, move the elements, and inserts the new element in the array.
- Not suitable for large data sets
- Complexity is $\boldsymbol{O}(\boldsymbol{n}^2)$, where n is the number of elements.
- Best case complexity is $\boldsymbol{O(n)}$.

Example



Elaborated 5th pass, i.e. inserted 31.



Algorithm

- **Algorithm insertionSort(a[], n)**
- Input: An array **a** containing **n** elements.
- Output: The elements of **a** get sorted in increasing order.

```
for i = 1 to n – 1
        temp = a[i]
        j = i
        while j > 0 and a[j-1] > temp
                a[j] = a[j-1]
                j = j – 1
        a[j] = temp
```

Implementation

```
#include<stdio.h>
void insertionSort(int a[], int n)
{   int i, j, temp;
    for(i=1; i <= n-1; i++ )
    {   temp = a[i];
        j = i;
        while (j > 0 &&
                a[j-1] > temp)
        {  a[j] = a[j-1];
           j = j - 1;
        }
        a[j] = temp;
    }
}
int main()
{   int a[10];
    printf("Enter 10 numbers: \n");
    for(int i = 0; i < 10; i++)
      scanf("%d",&a[i]);
    insertionSort(a,10);
    printf("\n");
    for(int i = 0; i < 10; i++)
      printf("%d ",a[i]);
    return 0;
}
```

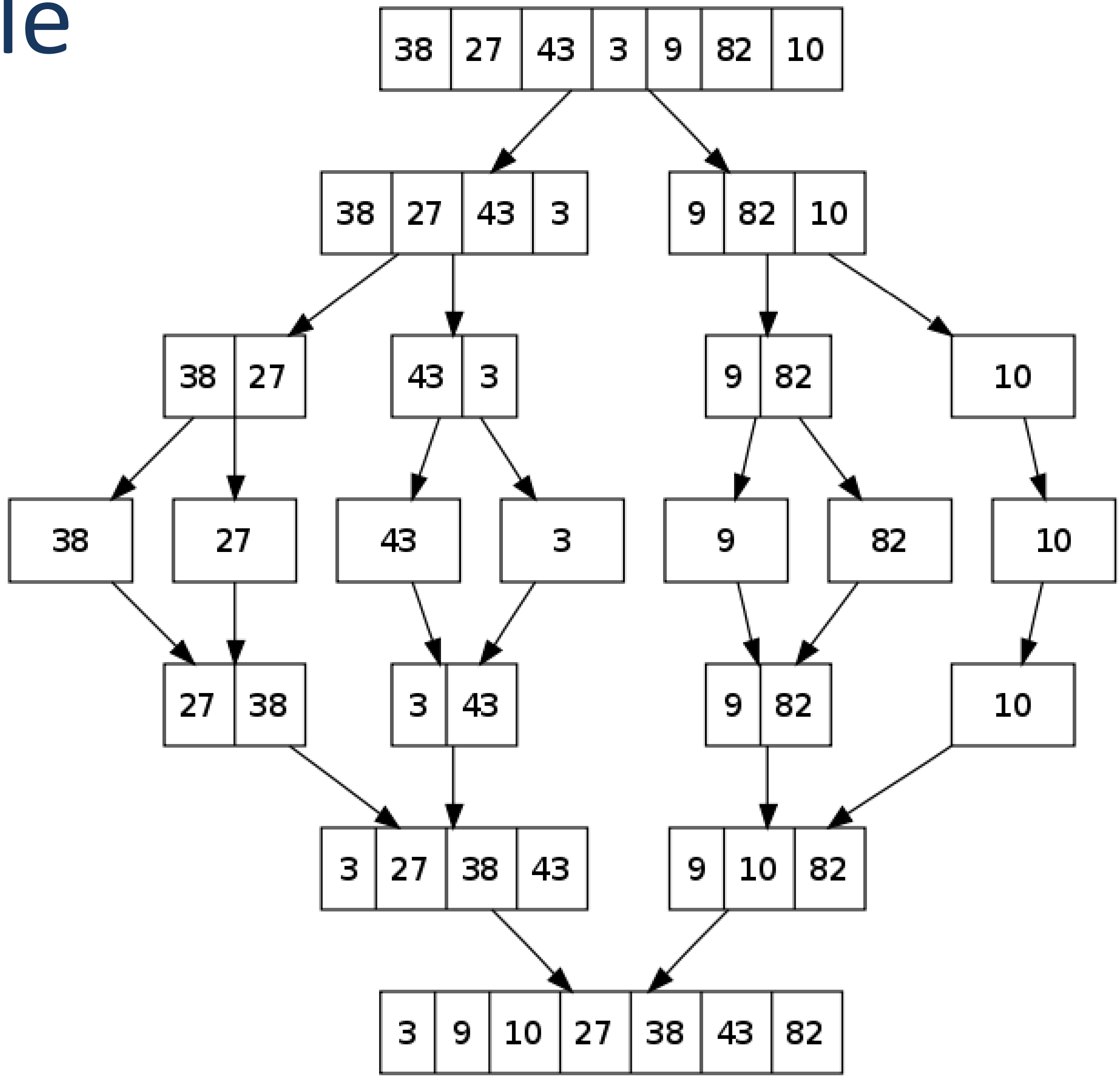
Merge Sort

Merge Sort

- Based on the divide-and-conquer paradigm.
- To sort an array A[p .. r], (initially p = 0 and r = n-1)

1. Divide Step
   - If a given array A has zero or one element, then return as it is already sorted.
   - Otherwise, split A[p…r] into two subarrays A[p…q] and A[q + 1... r], each containing about half of the elements of A[p…r]. That is, q is the halfway point of A[p...r].
2. Conquer Step
   - Recursively sort the two subarrays A[p…q] and A[q + 1…r].
3. Combine Step
   - Combine the elements back in A[p…r] by merging the two sorted subarrays A[p…q] and A[q + 1…r] into a sorted sequence.

Example


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Algorithm

- MERGE-SORT (A, p, r)

1. If $p < r$
2. $q = \text{FLOOR}[(p + r)/2]$
3. MERGE-SORT(A, p, q)
4. MERGE-SORT(A, q + 1, r)
5. MERGE (A, p, q, r)

- To sort an array A with n elements, the first call to MERGE-SORT is made with $p = 0$ and $r = n – 1$.

Call sequence for an array with size 8

- MERGE-SORT (A, 0, 7)
  - MERGE-SORT (A, 0, 3)
    - MERGE-SORT (A, 0, 1)
      - MERGE-SORT (A, 0, 0)
      - MERGE-SORT (A, 1, 1)
      - **MERGE (A, 0, 0, 1)**
    - MERGE-SORT (A, 2, 3)
      - MERGE-SORT (A, 2, 2)
      - MERGE-SORT (A, 3, 3)
      - **MERGE (A, 2, 2, 3)**
    - **MERGE (A, 0, 1, 3)**
  - MERGE-SORT (A, 4, 7)
    - MERGE-SORT (A, 4, 5)
      - MERGE-SORT (A, 4, 4)
      - MERGE-SORT (A, 5, 5)
      - **MERGE (A, 4, 4, 5)**
    - MERGE-SORT (A, 6, 7)
      - MERGE-SORT (A, 6, 6)
      - MERGE-SORT (A, 7, 7)
      - **MERGE (A, 6, 6, 7)**
    - **MERGE (A, 4, 5, 7)**
  - **MERGE (A, 0, 3, 7)**

Contd…

- Algorithm MERGE (A, p, q, r)
- Input: Array A and indices p, q, r such that $p \leq q \leq r$. Subarrays A[p…q] and A[q + 1…r] are sorted.
- Output: The two subarrays are merged into a single sorted subarray in A[p .. r].

```
1.  n1 = q − p + 1
2.  n2 = r − q
3.  Create arrays L[n1] and R[n2]
4.  For i = 0 to n1 – 1
5.          L[i] = A[p + i]
6.  For j = 0 to n2 – 1
7.          R[j] = A[q + 1 + j ]
```

Contd…

```
i = 0, j = 0, and k = p.
While i < n1 and j < n2
        if L[i] ≤ R[j]
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
        k++
While i < n1
        A[k] = L[i]
        i++
        k++
While j < n2
        A[k] = R[j];
        j++;
        k++;
```

Implementation

```
#include<stdio.h>
void merge(int arr[], int l, int m, int r)
{   int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
    i = 0;    j = 0;    k = l;
while (i < n1 && j < n2)
    {  if (L[i] <= R[j])
        {  arr[k] = L[i];     i++;       }
        else
        {  arr[k] = R[j];     j++;       }
        k++;    }
    while (i < n1)
    {   arr[k] = L[i];
        i++;        k++;    }
    while (j < n2)
    {  arr[k] = R[j];
        j++;        k++;    }        }
```

Implementation

```
void mergeSort(int arr[], int l, int r)
{   if (l < r)
    {   int m = l+(r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```

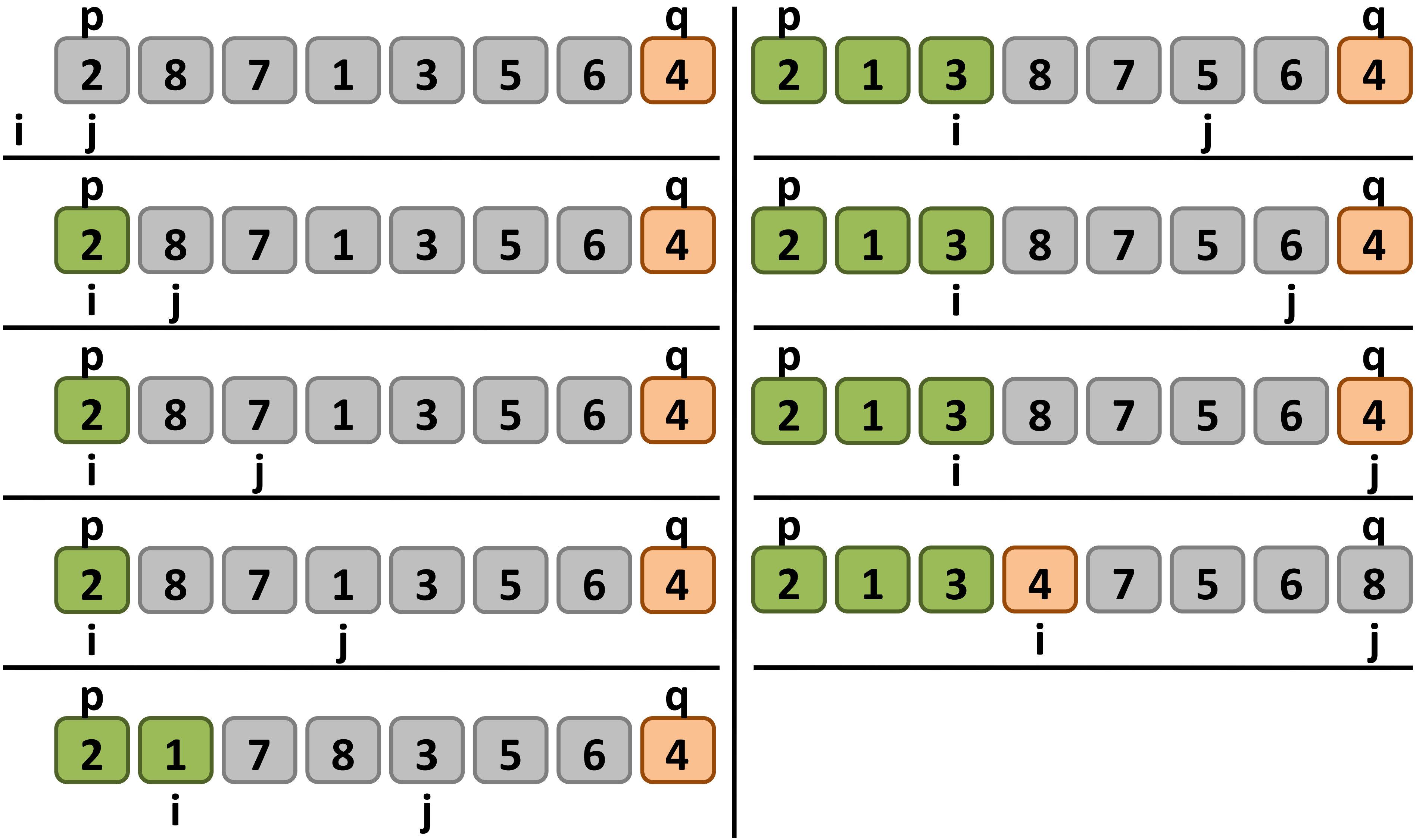
```
int main()
{   int a[10];
    printf("Enter 10 numbers: \n");
    for(int i = 0; i < 10; i++)
      scanf("%d",&a[i]);
    mergeSort(a,0,9);
    printf("\n");
    for(int i = 0; i < 10; i++)
      printf("%d ",a[i]);
    return 0;
}
```

Quick Sort

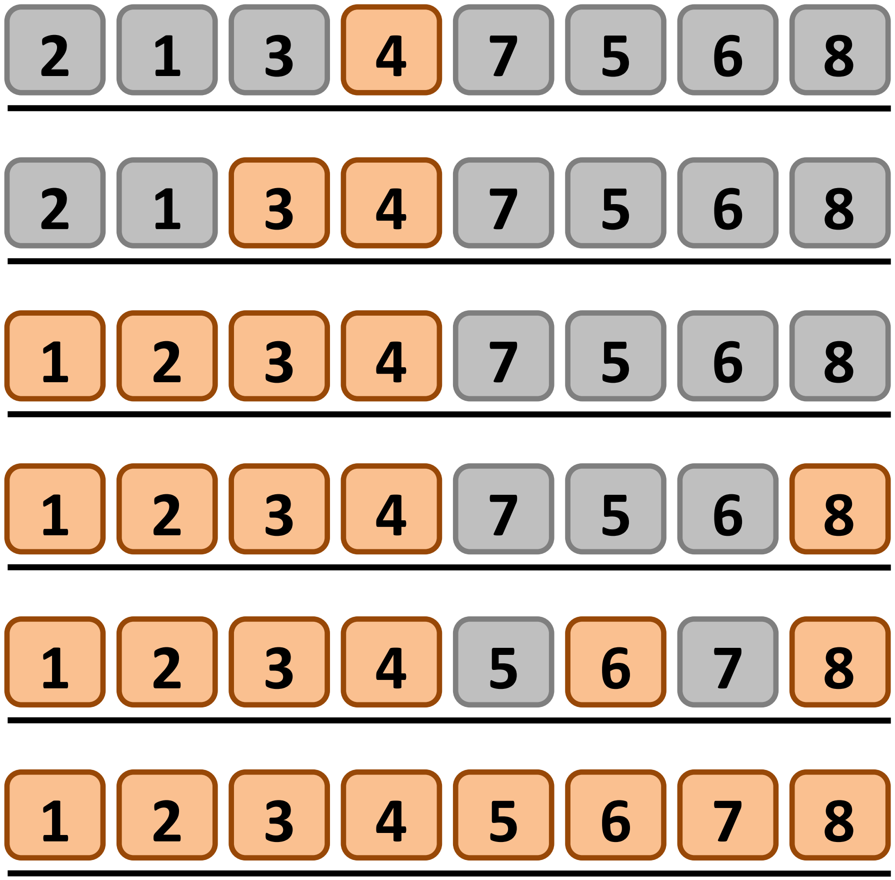
Quick Sort

- Divide and Conquer algorithm.
- Picks an element as pivot and partitions the given array around the picked pivot, such that
  - The pivot is placed at its correct position
  - All elements smaller than the pivot are placed before the pivot.
  - All elements greater than the pivot are placed after the pivot.
- Several ways to pick a pivot.
  - The first element.
  - The last element.
  - Any random element.
  - The median.

Example: 2, 8, 7, 1, 3, 5, 6, 4



Contd…



Algorithm

- QUICKSORT(A, p, r)

```
1. if p < r
2.     q = PARTITION(A, p, r)
3.     QUICKSORT(A, p, q – 1)
4.     QUICKSORT(A, q + 1, r)
```

- To sort an array A with n elements, the first call to QUICKSORT is made with $p = 0$ and $r = n – 1$.

```
1. PARTITION(A, p, r)
2. x = A[r]
3. i = p – 1
4. for j = p to r – 1
5.     if A[j] ≤ x
6.         i = i + 1
7.         Exchange A[i] with A[j]
8. Exchange A[i + 1] with A[r]
9. return i + 1
```

Implementation

```
#include<stdio.h>
int partition (int arr[], int low, int high)
{   int pivot = arr[high];
    int temp, i = (low - 1);
     for (int j = low; j <= high- 1; j++)
     {  if (arr[j] <= pivot)
        {   i++;
            temp = arr[i];          arr[i] = arr[j];          arr[j] = temp;
        }       }
    temp = arr[i + 1];    arr[i + 1] = arr[high];    arr[high] = temp;
    return (i + 1);
}
```

Implementation

```
void quickSort(int arr[], int low, int high)
{   if (low < high)
    {   int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);    }}
int main()
{   int n, k = 0, A[15], i;
    printf("Enter 10 numbers: \n");
    for (i = 0; i < 10; i++)
    {   scanf("%d", &A[i]);
        if (A[i] > k)            k = A[i];        }
    quickSort(A, 0, 9);    printf("\n");
    for(int i = 0; i < 10; i++)      printf("%d ",A[i]);
    return 0;            }
```

Counting Sort

Counting Sort

- Assumes that the input consists of integers in a small range 1 to k, for some integer k.
- Runs in $O(n + k)$ time.
- For each element x, the algorithm
  - First determines the number of elements less than x.
  - Then directly place the element into its correct position.

Example

A[]

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

k = 6

n = 8

- Compute frequency of k elements, i.e. array C.

C[]

0	1	2	3	4	5	6
0	**2**	**0**	**2**	**3**	**0**	**1**

- Update C to store cumulative frequency.

C[]

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

A[] | C[] | B[]

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

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

0	1	2	3	4	5	6	7	8
							4	

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

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

0	1	2	3	4	5	6	7	8
		1					4	

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Algorithm

- Algorithm countingSort(A,n,k)
- Input: Array A, its size n, and the maximum integer k in the list.
- Output: The elements of A get sorted in increasing order.

1. for i = 0 to k
2. C[i] = 0
3. for i = 0 to n – 1
4. C[A[i]] = C[A[i]] + 1
5. for i = 1 to k
6. C[i] = C[i] + C[i-1]
7. for i = n – 1 to 0
8. B[C[A[i]]] = A[i]
9. C[A[i]] = C[A[i]] - 1
10. for i = 0 to n – 1
11. A[i] = B[i+1]

Implementation

```
#include <stdio.h>
void countingSort(int A[], int k, int n)
{   int i, j;
    int B[n+1], C[100];
    for (i = 0; i <= k; i++)            C[i] = 0;
    for (j = 0; j < n; j++)             C[A[j]] = C[A[j]] + 1;
    for (i = 1; i <= k; i++)            C[i] = C[i] + C[i-1];
    for (j = n-1; j >= 0; j--)
    {   B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] – 1;    }
    for (i = 0; i < n; i++)             A[i] = B[i+1];}
```

Contd...

```
int main()
{  int k = 0, A[15], i;
    printf("Enter 10 numbers: \n");
    for (i = 0; i < 10; i++)
    {  scanf("%d", &A[i]);
        if (A[i] > k)   k = A[i];    }
    countingSort(A, k, 10);
    printf("\n");
    for(int i = 0; i < 10; i++)                  printf("%d ",A[i]);
    return 0;        }
```

Radix Sort

Radix Sort

- Similar to alphabetizing a large list of names.
  - List of names is first sorted according to the first letter of each names, that is, the names are arranged in 26 classes.
  - Then sort on the next most significant letter, and so on.
- Radix sort do counter-intuitively by sorting on the least significant digits first.
  - First pass sorts entire list on the least significant digit.
  - Second pass sorts entire list again on the second least-significant digits and so on.

Example

INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Algorithm

- Assumption: Each element in the n-element array A has d digits, where digit 1 is the least-significant digit and d is the most-significant digit.

- radixSort(A, d)

1. for i = 1 to d
2. use a stable sort to sort A on digit i
   // counting sort will do the job

Implementation

```
#include<stdio.h>
void countSort(int A[], int n, int exp)
{   int B[n+1], i, C[10] = {0};
    for (i = 0; i < n; i++)
        C[ (A[i]/exp)%10 ]++;
    for (i = 1; i < 10; i++)
        C[i] = C[i] + C[i - 1];
    for (i = n - 1; i >= 0; i--)
    {   B[C[ (A[i]/exp)%10 ]] = A[i];
        C[ (A[i]/exp)%10 ]--;    }
    for (i = 0; i < n; i++)
        A[i] = B[i+1];          }
```

Contd…

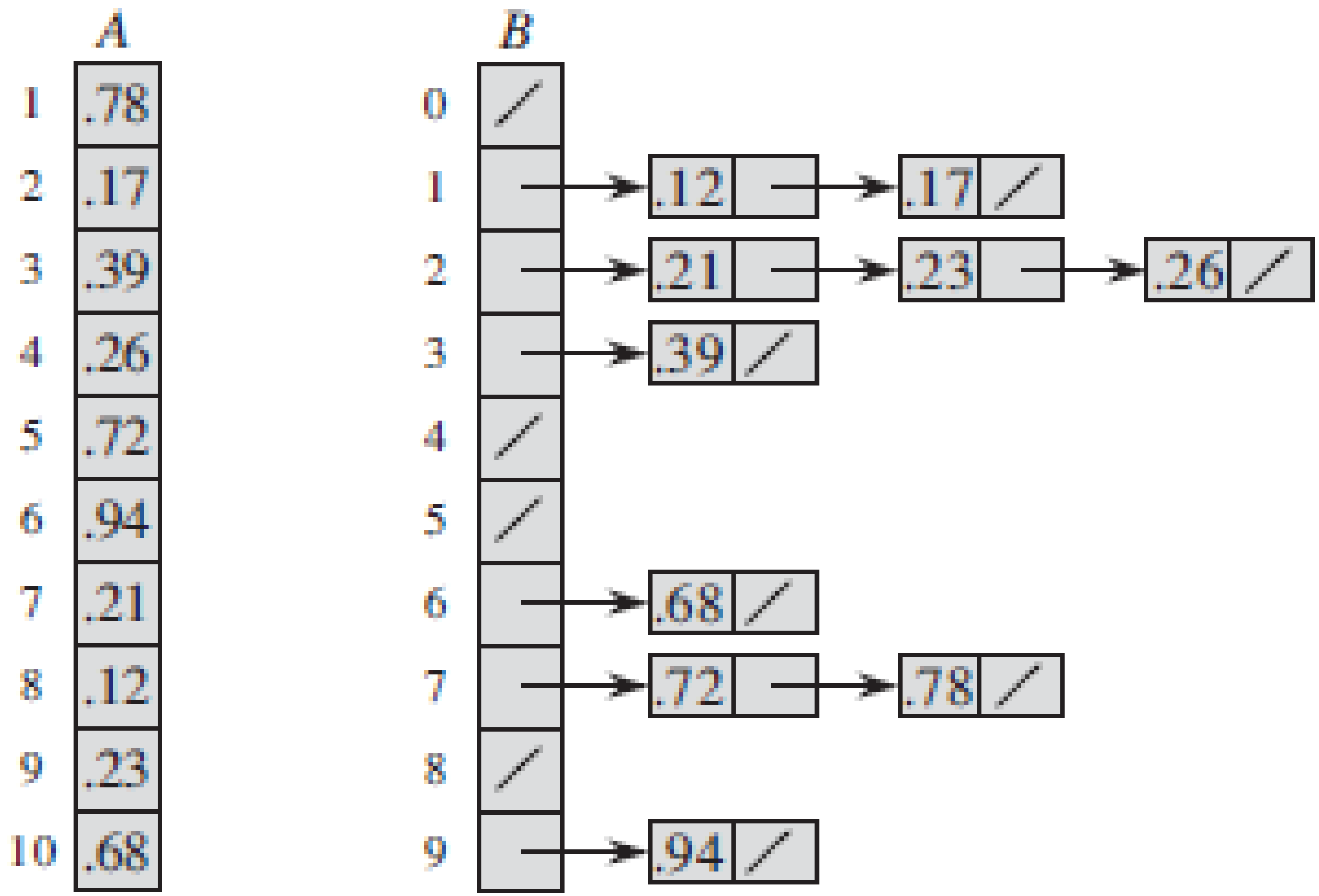
```
void radixsort(int A[], int k, int n)
{    for (int exp = 1; k/exp > 0; exp *= 10)
          countSort(A, n, exp);     }
int main()
{   int A[10], k = 0, i;
    printf("Enter 10 numbers: \n");
    for (i = 0; i < 10; i++)
    {   scanf("%d", &A[i]);
        if (A[i] > k)          k = A[i];}
    radixsort(A, k, 10);   printf("\n");
    for(int i = 0; i < 10; i++)      printf("%d ",A[i]);
    return 0;        }
```

Bucket Sort

Bucket Sort

- Assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0,1).
- Average-case running time is O(n).
- Divides the interval [0,1) into n equal-sized subintervals, or buckets.
- Distributes the n input numbers into the buckets and sort the numbers in each bucket.
- Lastly list elements of all buckets in order.

Example



A
1
.78
2
.17
3
.39
4
.26
5
.72
6
.94
7
.21
8
.12
9
.23
10
.68
B
0
1
2
3
4
5
6
7
8
9
.12
.17
.21
.23
.26
.39
.68
.72
.78
.94

Algorithm

- bucketSort(A[], n)

1. Create B[n], i.e. an array of linked list (or buckets).
2. for i = 0 to n – 1
3. make B[i] an empty list
4. for i = 0 to n – 1
5. insert A[i] into list B[Floor(n*A[i])]
6. for i = 0 to n – 1
7. sort list B[i] with insertion sort
8. Concatenate B[0], B[1], … B[n – 1] together in order.

Thank You