Sorting Algorithms

Mohammad Asad Abbasi

Lecture 9

Sorting by Exchange: Shell Sort

- Sorting methods based on comparison:
  - Comparisons and hence movements of data take place between adjacent entries only
  - This leads to a number of redundant comparisons and data movements
  - A mechanism should be followed with which the comparisons can take in long leaps instead of short
    - Donald L. Shell (1959)

  - Use increments:

$$h_t, h_{t-1}, h_{t-2}, \ldots, h_1$$

Shell Sort

- Shell sort, also known as the **diminishing increment sort,** is one of the oldest sorting algorithms

- It improves on insertion sort

- Starts by comparing elements far apart, then elements less far apart, and finally comparing adjacent elements (effectively an insertion sort). By this stage the elements are sufficiently sorted that the running time of the final stage is much closer to $O(N)$ than $O(N^2)$

Shell Sort (Steps)

- Let A be a linear array of *n* numbers A [1], A [2], A [3], ... A [n].

- ***Step* 1:**
  - The array is divided into *k* sub-arrays consisting of every *k*th element. Say *k*= 5, then five sub-array, each containing one fifth of the elements of the original array

Sub array 1 → A[0] A[5] A[10]
Sub array 2 → A[1] A[6] A[11]
Sub array 3 → A[2] A[7] A[12]
Sub array 4 → A[3] A[8] A[13]
Sub array 5 → A[4] A[9] A[14]

- **Note :** The i^{th} element of the j^{th} sub array is located as A $[(i-1) \times k+j-1]$

Shell Sort (Steps)

- ***Step 2:***

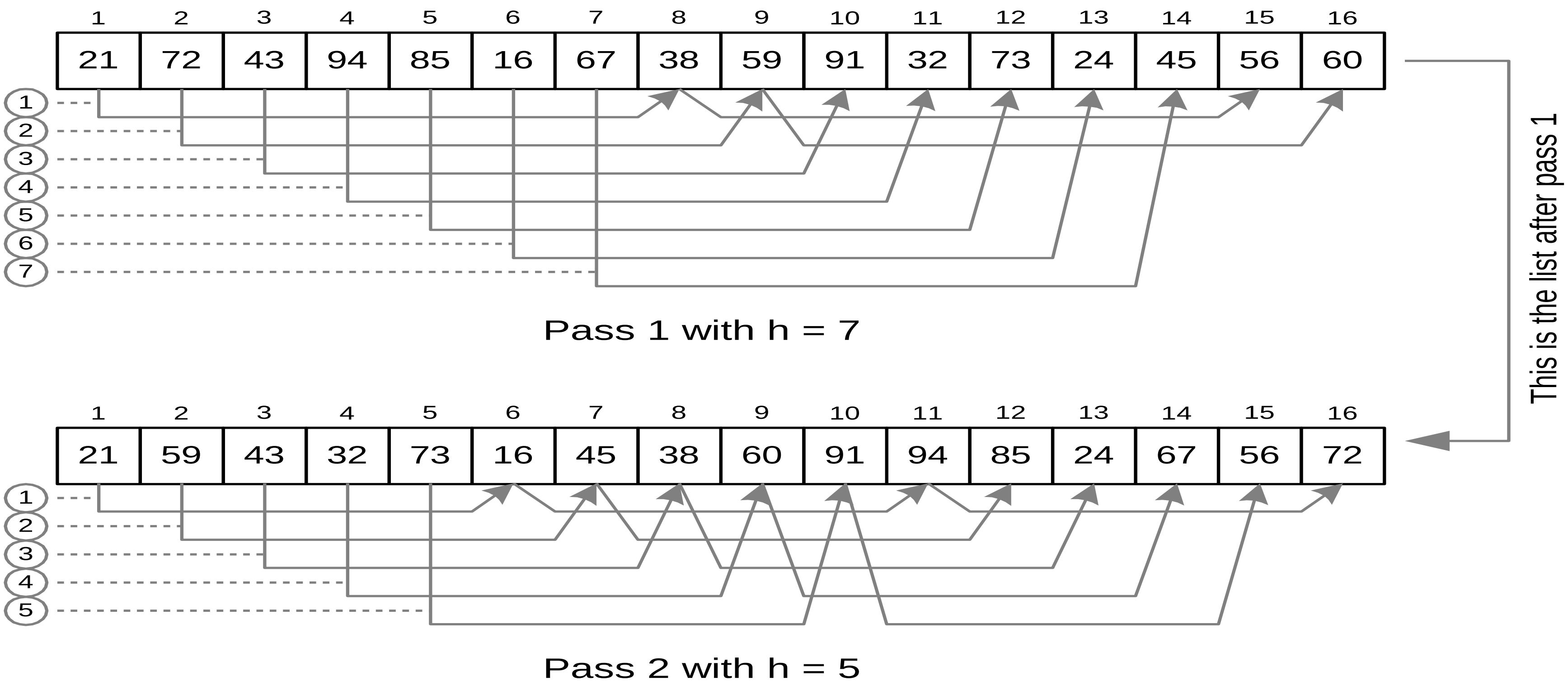
- After the first *k* sub array are sorted (usually by insertion sort) , a new smaller value of *k* is chosen and the array is again partitioned into a new set of sub arrays

Shell Sort (Steps)

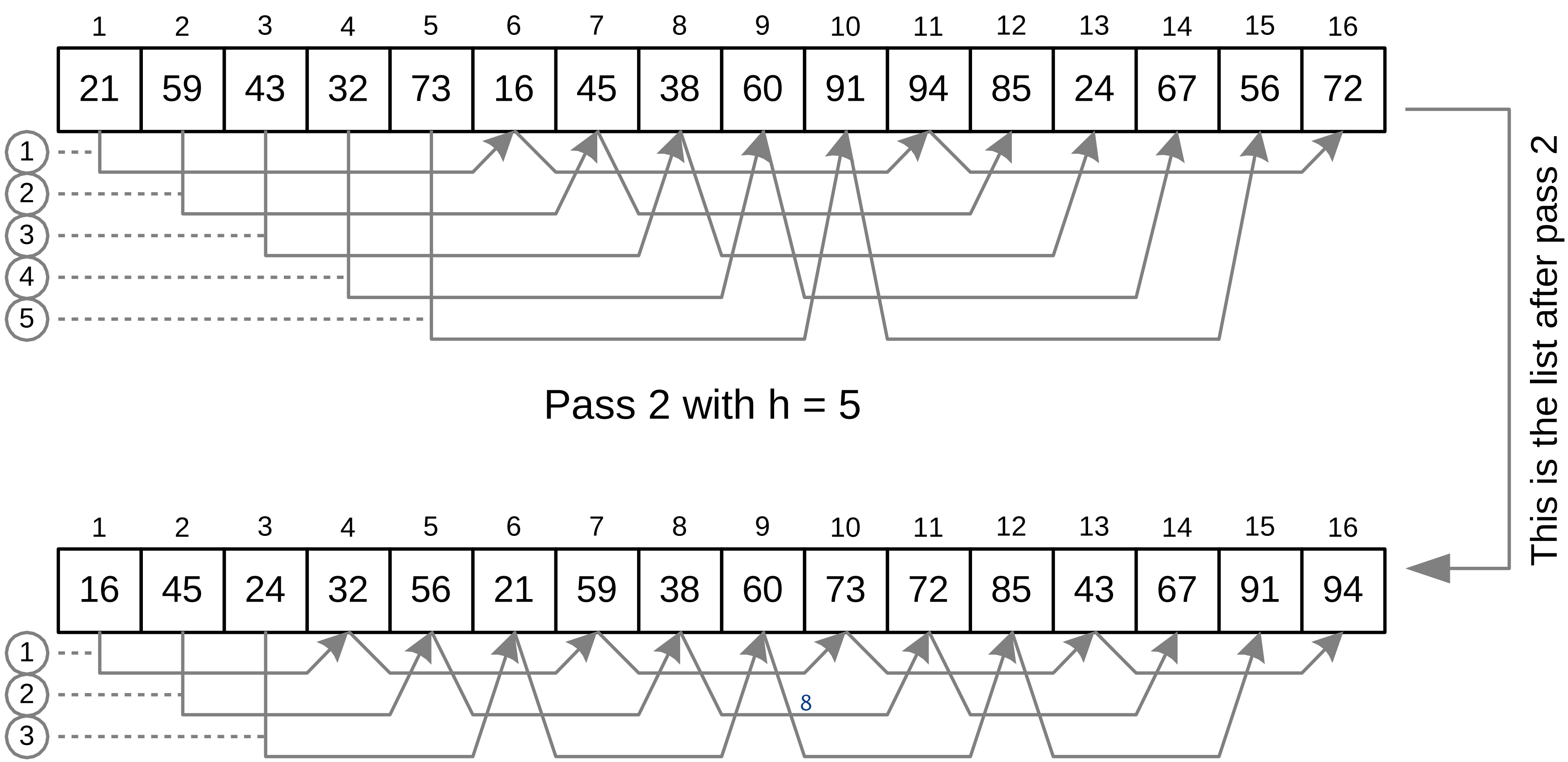
- ***Step 3:***

- And the process is repeated with an even smaller value of *k*, so that A [1], A [2], A [3], ... A [*n*] is sorted

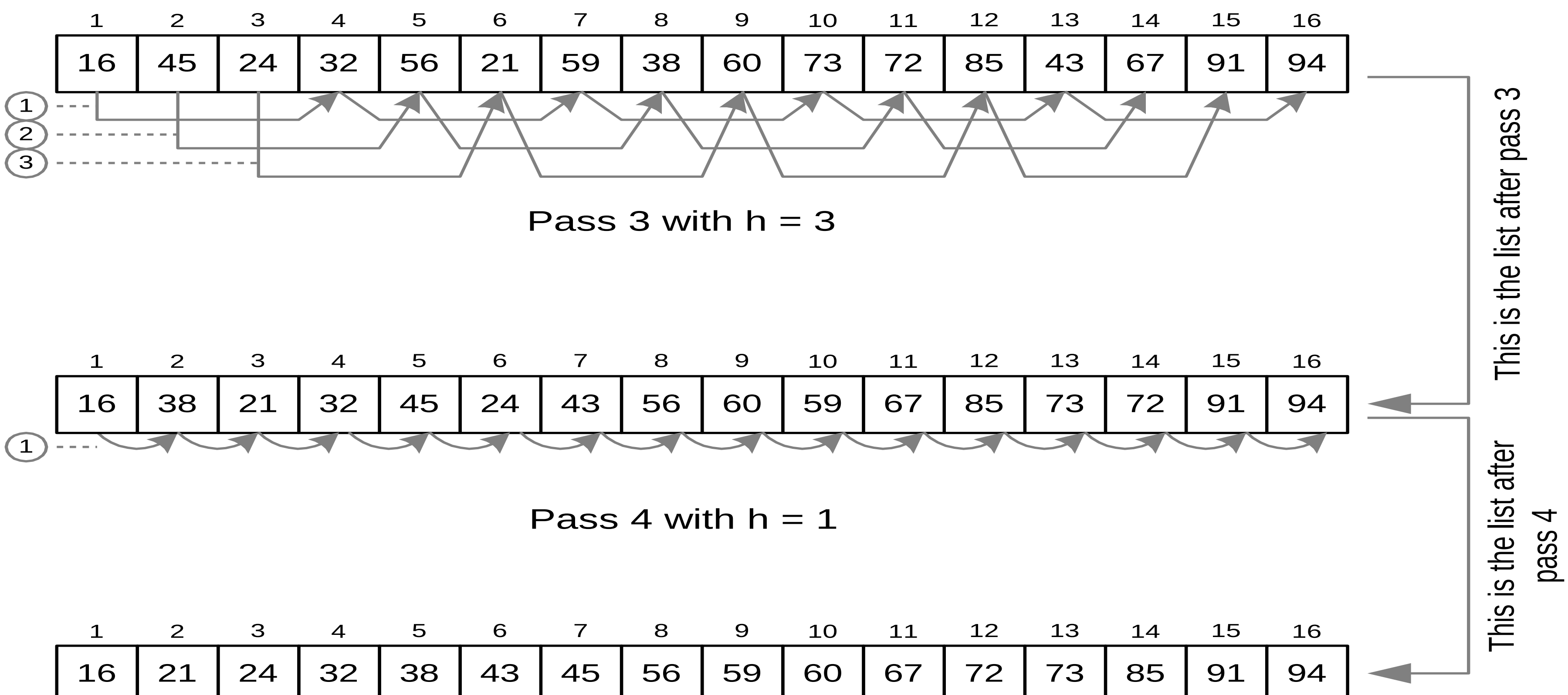
Shell Sort: Illustration



Shell Sort: Illustration



Shell Sort: Illustration



Output list

Shell Sort: Illustration

To illustrate the shell sort, consider the following array with 7 elements 42, 33, 23, 74, 44, 67, 49 and the sequence K = 4, 2, 1 is chosen.

Pass = 1

Span = k = 4

42, 33, 23, 74, 44, 67, 49

Pass = 2

span = k = 2

42, 33, 23, 74, 44, 67, 49

Pass = 3

Span = k = 1

23, 33, 42, 67, 44, 74, 49

23, 33, 42, 44, 49, 67, 74

ALGORITHM

- Let A be a linear array of *n* elements, A [1], A [2], A [3], ... A[*n*] and *Incr* be an array of sequence of span to be incremented in each pass. X is the number of elements in the array *Incr*. *Span* is to store the span of the array from the array Incr.

1. Input *n* numbers of an array A
2. Initialise *i* = 0 and repeat through step 6 if (*i* < *x*)
3. Span = Incr[*i*]
4. Initialise *j* = span and repeat through step 6 if (*j* < *n*)
   (*a*) Temp = A [*j*]
5. Initialise *k* = *j*-span and repeat through step 5 if (*k* > = 0) and (temp < A [*k*])
   (*a*) A [*k* + span] = A [*k*]
6. A [*k* + span] = temp
7. Exit

	Pass # 1				i= num / 2			(i= 7/2 = 3)	
	index								
	0	1	2	3	4	5	6		
1	**42**	**33**	**23**	**74**	**44**	**67**	**49**	j = i	J=3
	k			K+i				K=j-i	K=3-3=0
									K+i = 3
2	42	33	23	74	44	67	49		J=4
		k			K+I				K=4-3=1
3	42	33	23	74	44	67	49		J=5
			k			K+i			K=5-3=2
4	42	33	23	74	44	67	49		J=6
				k	swap		K+i		K=6-3=3
	42	33	23	49	44	67	74		

```
for(i=num/2; i>0; i=i/2)
{
 for(j=i; j<num; j++)
 {
  for(k=j-i; k>=0; k=k-i)
  {
   if(arr[k+i]>=arr[k])
      break;
   else
   {
      tmp=arr[k];
      arr[k]=arr[k+i];
      arr[k+i]=tmp;
   }
  }
 }
}
```

	Pass # 2					i= i / 2			(i= 3 / 2 = 1)
	index								
	0	1	2	3	4	5	6		
1	42	33	23	49	44	67	74	j = i	J=1
	k	K+i						K=j-i	K=1-1=0
									K+i = 1
2	33	42	23	49	44	67	74		J=2
		k	K+i						K=2-1= 1
3	33	23	42	49	44	67	74		
	k	K+i	K=k-i						
4	23	33	42	49	44	67	74		J=3
			k	K+i					K=3-1=2
5	23	33	42	49	44	67	74		J=4
				K	K+i				K=4-1=3

```
for(i=num/2; i>0; i=i/2)
{
 for(j=i; j<num; j++)
 {
  for(k=j-i; k>=0; k=k-i)
  {
   if(arr[k+i]>=arr[k])
     break;
   else
   {
     tmp=arr[k];
     arr[k]=arr[k+i];
     arr[k+i]=tmp;
   }
  }
 }
}
```

	Pass # 2				i= i / 2				(i= 3 / 2 = 1)
	index								
	0	1	2	3	4	5	6		
6	23	33	42	49	44	67	74		J=4
				K	K+i				K=4-1=3
7	23	33	42	44	49	67	74		J=5
					K	K+i			K=5-1=4
8	23	33	42	44	49	67	74		J=6
						K	K+i		K=6-1=5
SORTED									

```
for(i=num/2; i>0; i=i/2)
{
 for(j=i; j<num; j++)
 {
  for(k=j-i; k>=0; k=k-i)
  {
    if(arr[k+i]>=arr[k])
       break;
    else
    {
       tmp=arr[k];
       arr[k]=arr[k+i];
       arr[k+i]=tmp;
    }
  }
 }
}
```

The Complexity

- If an appropriate sequence of increments is classified, then the order of the shell sort is:
  - $f(n) = O(n(\log n))$

Issues in Shell Sort

- Algorithm to be used to sort subsequences in shell sort
  - Straight insertion sort
  - Shell sort is better than the insertion sort
    - Lower number of passes than n number of passes in insertion sort

- Deciding the values of increments
    - Several choices have been made

RADIX SORT

- Radix sort or bucket sort is a method that can be used to sort a list of numbers by its base
- If we want to sort list of English words, where radix or base is 26, then 26 buckets are used to sort the words

RADIX SORT EXAMPLE

Input: 478, 537, 9, 721, 3, 38, 123, 67

BucketSort on 1's

0	1	2	3	4	5	6	7	8	9
	721		03 123				537 67	478 38	09

BucketSort on 10's

0	1	2	3	4	5	6	7	8	9
003 009		721 123	537 038			067	478		

BucketSort on 100's

0	1	2	3	4	5	6	7	8	9
3 9 38 67	123			478	537		721		

Output: 3, 9, 38, 67, 123, 478, 537, 721

RADIX SORT EXAMPLE (1^{st} Pass)

Bucket sort
by 1's digit

Input data

478
537
9
721
3
38
123
67

0	1	2	3	4	5	6	7	8	9
	721		3 123				537 67	478 38	9

After 1^{st} pass

721
3
123
537
67
478
38
9

RADIX SORT EXAMPLE (2^{nd} Pass)

After 1^{st} pass

721
3
123
537
67
478
38
9

Bucket sort
by 10's
digit

0	1	2	3	4	5	6	7	8	9
03		721	537			67	478		
09		123	38						

After 2^{nd} pass

3
9
721
123
537
38
67
478

RADIX SORT EXAMPLE (3rd Pass)

After 2nd pass

3
9
721
123
537
38
67
478

Bucket sort by 100's digit

0	1	2	3	4	5	6	7	8	9
003 009 038 067	123			478	537		721		

After 3rd pass

3
9
38
67
123
478
537
721

Invariant: after k passes the low order k digits are sorted.

ALGORITHM

- Let A be a linear array of *n* elements A [1], A [2], A [3],...... A [*n*]. Digit is the total number of digits in the largest element in array A.

1. Input *n* number of elements in an array A.
2. Find the total number of Digits in the largest element in the array.
3. Initialize $i = 1$ and repeat the steps 4 and 5 until ($i <= \text{Digit}$).
4. Initialize the buckets $j = 0$ and repeat the steps (*a*) until ($j < n$)

   (*a*) Compare *i*th position of each element of the array with bucket number and place it in the corresponding bucket.
5. Read the element(s) of the bucket from 0th bucket to 9th bucket and from first position to higher one to generate new array A.
6. Display the sorted array A.
7. Exit.

RADIX SORT EXAMPLE

	index									
	0	1	2	3	4	5	6	7	8	9
	170	**45**	**75**	**90**	**802**	**24**	**2**	**66**		

count of occurrences in count[]	Change count[i]	Output array
count[(arr[i]/exp)%10]++	count[i] += count[i - 1]	output[count[(arr[i]/exp)%10] - 1]
Count[0] ~~1~~ 2	2	~~2~~ ~~1~~ 0
Count[1] 0	2	2
Count[2] ~~1~~ 2	4	~~4~~ ~~3~~ 2
Count[3] 0	4	4
Count[4] 1	5	~~5~~ 4
Count[5] ~~1~~ 2	7	~~7~~ ~~6~~ 5
Count[6] 1	8	~~8~~ 7
Count[7] 0	8	8
Count[8] 0	8	8
count[9] 0	8	8

Out put array

	0	1	2	3	4	5	6	7	8	9
	170	90	802	2	24	45	75	66		

Pass # 1

```
// Store count of occurrences in
   count[]

 for (i = 0; i < n; i++)
    count[ (arr[i]/exp)%10 ]++;

 // Change count[i]

 for (i = 1; i < 10; i++)
    count[i] += count[i - 1];
// Build the output array
 for (i = n - 1; i >= 0; i--)
 {
    output[count[ (arr[i]/exp)%10
   = arr[i];
    count[ (arr[i]/exp)%10 ]--;
 }
```

count of occurrences in count[]	Change count[i]	Output array
count[(arr[i]/exp)%10]++	count[i] += count[i - 1]	output[count[(arr[i]/exp)%10] - 1]
Count[0] ~~1~~ 2	2	~~2~~ ~~1~~ 0
Count[1] 0	2	2
Count[2] 1	3	~~3~~ 2
Count[3] 0	3	3
Count[4] 1	4	~~4~~ 3
Count[5] 0	4	4
Count[6] 1	5	~~5~~ 4
Count[7] ~~1~~ 2	7	~~7~~ ~~6~~ 5
Count[8] 0	7	7
count[9] 1	8	~~8~~ 7

Out put array

	0	1	2	3	4	5	6	7	8	9
	802	2	24	95	66	170	75	90		

Pass # 2

```
// Store count of occurrences in
   count[]
  for (i = 0; i < n; i++)
     count[ (arr[i]/exp)%10 ]++;

  // Change count[i]

  for (i = 1; i < 10; i++)
     count[i] += count[i - 1];
// Build the output array
  for (i = n - 1; i >= 0; i--)
  {
     output[count[ (arr[i]/exp)%10
   = arr[i];
     count[ (arr[i]/exp)%10 ]--;
  }
```

count of occurrences in count[]	Change count[i]	Output array
count[(arr[i]/exp)%10]++	count[i] += count[i - 1]	output[count[(arr[i]/exp)%10] - 1]
Count[0] ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6	6	~~6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0
Count[1] 1	7	~~7~~ 6
Count[2] 0	7	7
Count[3] 0	7	7
Count[4] 0	7	7
Count[5] 0	7	7
Count[6] 0	7	7
Count[7] 0	7	7
Count[8] 1	8	~~8~~ 7
count[9] 0	8	8

Out put array

	0	1	2	3	4	5	6	7	8	9
	2	24	45	66	75	90	170	802		

Pass # 3

```
// Store count of occurrences in
   count[]

for (i = 0; i < n; i++)
    count[ (arr[i]/exp)%10 ]++;

// Change count[i]

for (i = 1; i < 10; i++)
    count[i] += count[i - 1];
// Build the output array
for (i = n - 1; i >= 0; i--)
{
    output[count[ (arr[i]/exp)%10
  = arr[i];
    count[ (arr[i]/exp)%10 ]--;
}
```

Radix Sort Program

```
// C++ implementation of Radix Sort
#include<iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}
```

Radix Sort Program

```
// A function to do counting sort of arr[] according to the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;

    // Change count[i] so that count[i] now contains actual position of
    // this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
```

Radix Sort Program

```
    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

Radix Sort Program

```
// The main function to that sorts arr[] of size n using Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead of passing digit
    // number, exp is passed. exp is 10^i where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

Advantages and Disadvantages

- **Advantages**
  - Radix and bucket sorts are stable, preserving existing order of equal keys
  - They work in linear time, unlike most other sorts. In other words, they do not bog down when large numbers of items need to be sorted. Most sorts run in O(n log n) or O(n^2) time.
  - The time to sort per item is constant, as no comparisons among items are made. With other sorts, the time to sort per time increases with the number of items.
  - Radix sort is particularly efficient when you have large numbers of records to sort with short keys

- **Drawbacks**
  - Radix and bucket sorts do not work well when keys are very long, as the total sorting time is proportional to key length and to the number of items to sort
  - They are not "in-place", using more working memory than a traditional sort

Running time analysis of Radix sort

- How many passes?
- How much work per pass?
- Total time?

- Conclusion
  - Not truly linear if K is large
- In practice
  - Radix Sort only good for large number of items, relatively small keys
  - Hard on the cache, vs. MergeSort/QuickSort

Running time analysis of Radix sort

- Time requirement for the radix sorting method depends on the number of digits and the elements in the array

- **WORST CASE**
  - $f(n) = O(n^2)$

- **BEST CASE**
  - $f(n) = O(n \log n)$

- **AVERAGE CASE**
  - $f(n) = O(n \log n)$

MERGE SORT

- Merge sort is based on the **divide-and-conquer** paradigm

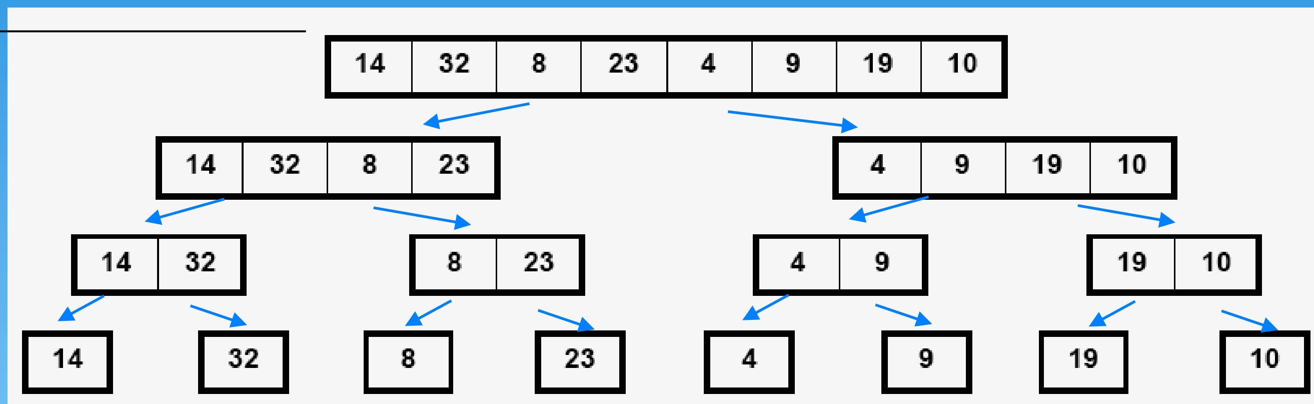
- Conceptually, a merge sort works as follows:

  1. Divide the unsorted list into *n* sub lists, each containing 1 element (a list of 1 element is considered sorted)

  2. Repeatedly merge sub lists to produce new sorted sub lists until there is only 1 sub list remaining. This will be the sorted list

Merge Sort

Input

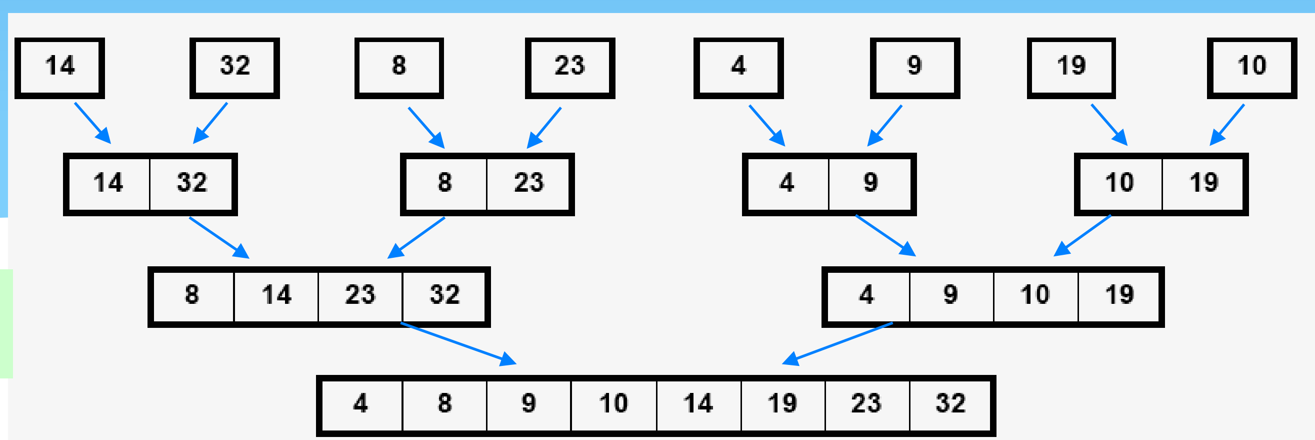


(divide)

How much work at every step?

O(lg n) steps

O(n) sub-problems



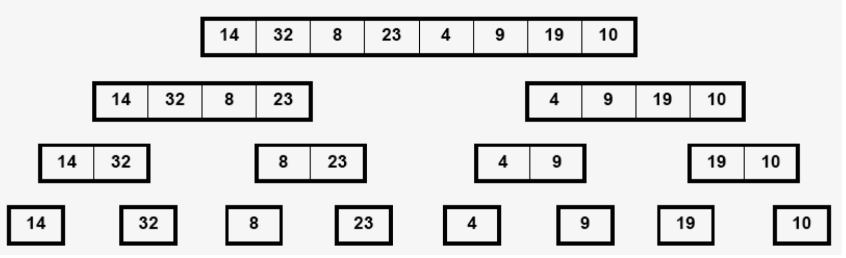
(conquer)

How much work at every step?

O(lg n) steps

MERGE SORT

```
void MergeSort(int *A,int n) {
    int mid, i, *L, *R;

    if(n < 2)
        return; // base condition. If the array has less than two element, do nothing.

    mid = n/2; // find the mid index.

    // create left and right subarrays
    // mid elements (from index 0 till mid-1) should be part of left sub-array
    // and (n-mid) elements (from mid to n-1) will be part of right sub-array

    L = (int*)malloc(mid*sizeof(int));
    R = (int*)malloc((n - mid)*sizeof(int));

    for(i = 0;i<mid;i++)
        L[i] = A[i]; // creating left subarray

    for(i = mid;i<n;i++)
        R[i-mid] = A[i]; // creating right subarray

    MergeSort(L,mid); // sorting the left subarray
    MergeSort(R,n-mid); // sorting the right subarray

    Merge(A,L,mid,R,n-mid); // Merging L and R into A as sorted list.
}
```



Merging

L:

3	10	23	54

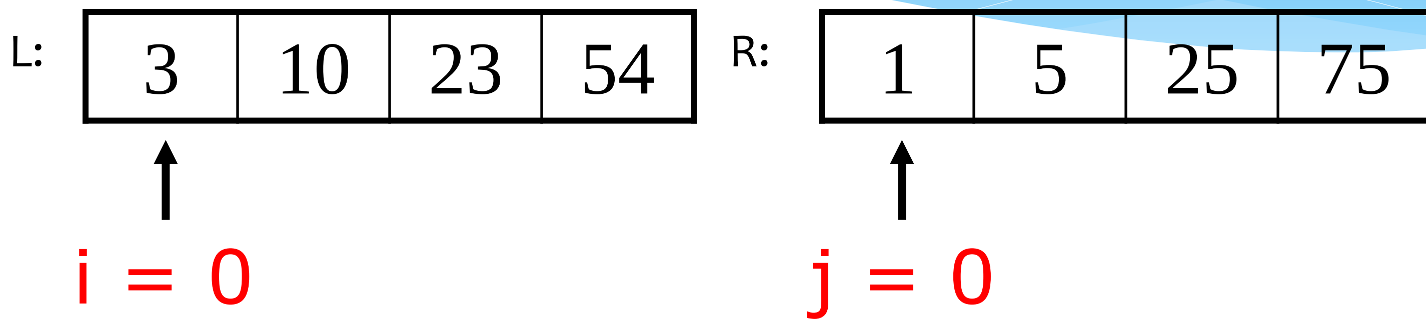
R:

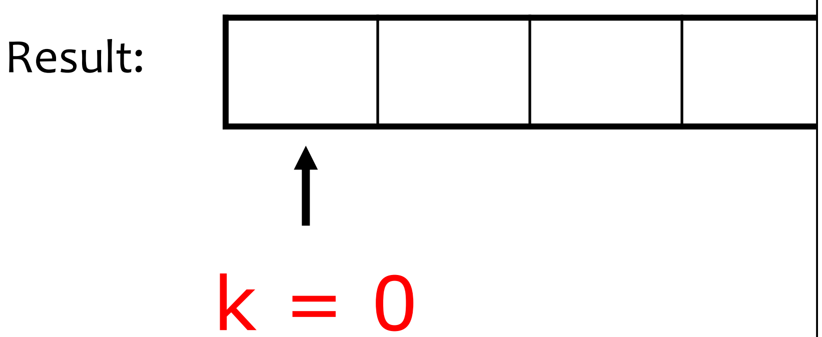
1	5	25	75

Result:

1	3	5	10	23	25	54	75

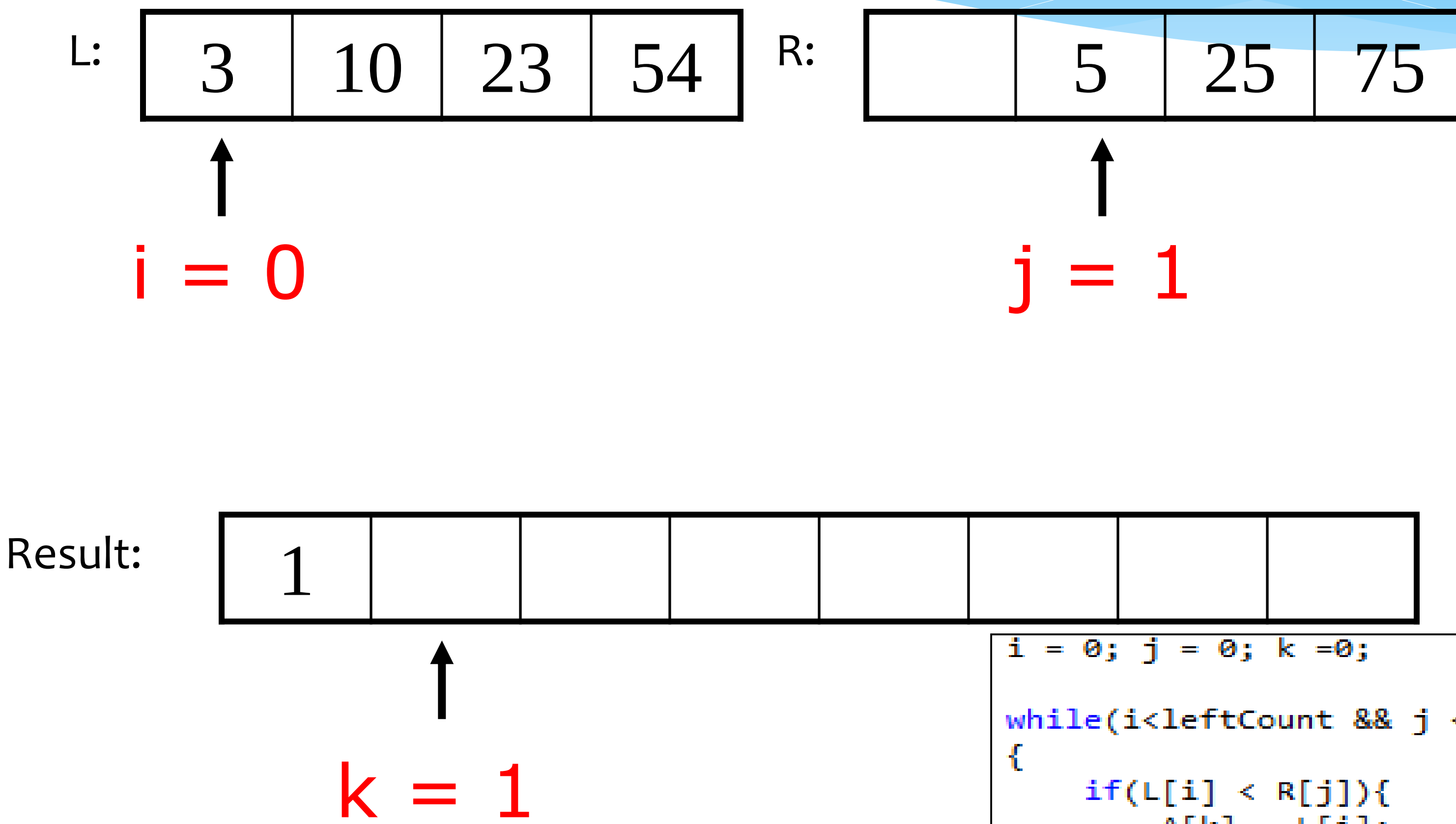
Merging





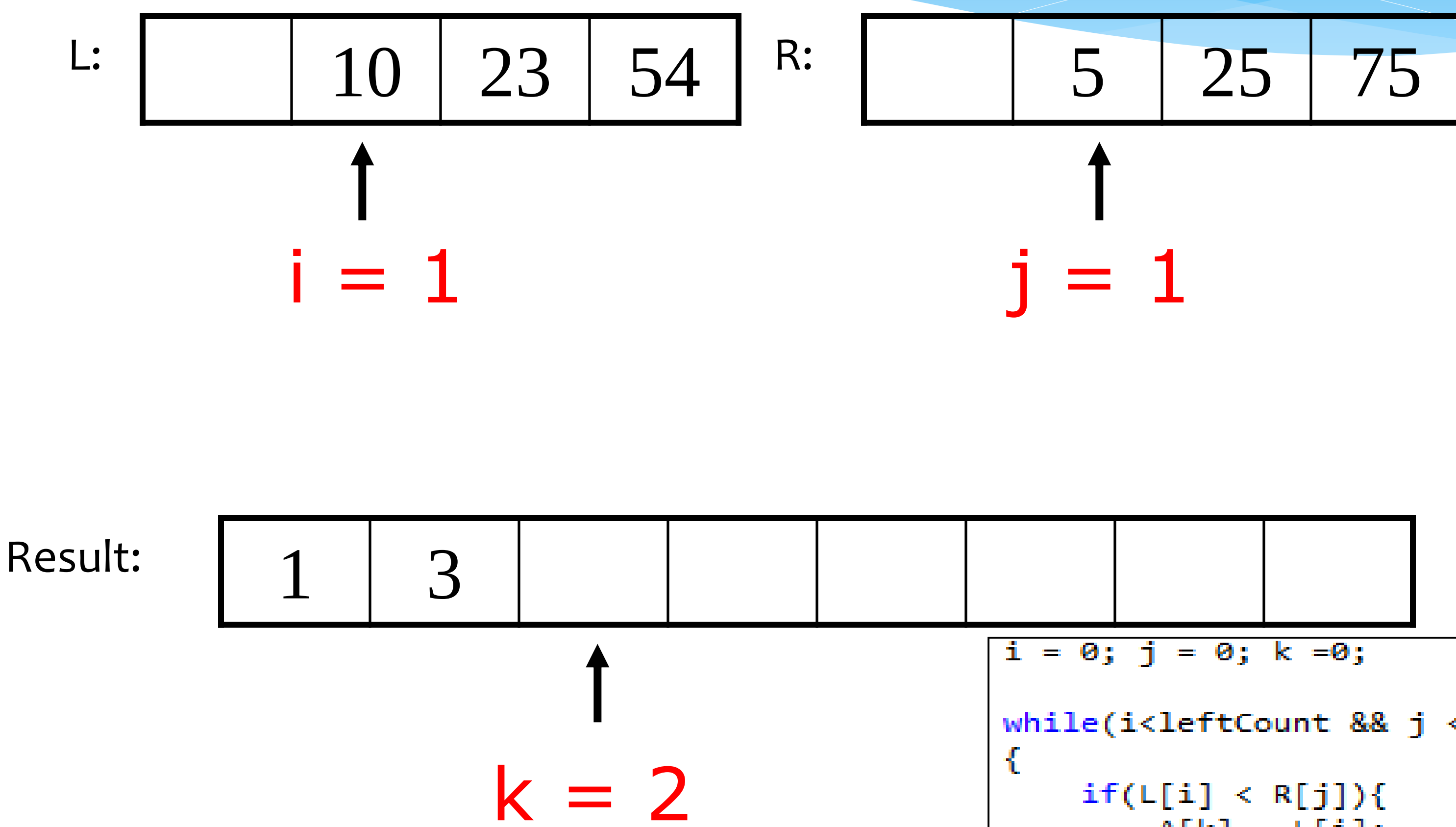

```
i = 0; j = 0; k =0;

while(i<leftCount && j < rightCount)
{
    if(L[i] < R[j]){
        A[k] = L[i];
        k++; i++;
    }

    else{
        A[k] = R[j];
        k++; j++;
    }
}
```

Merging



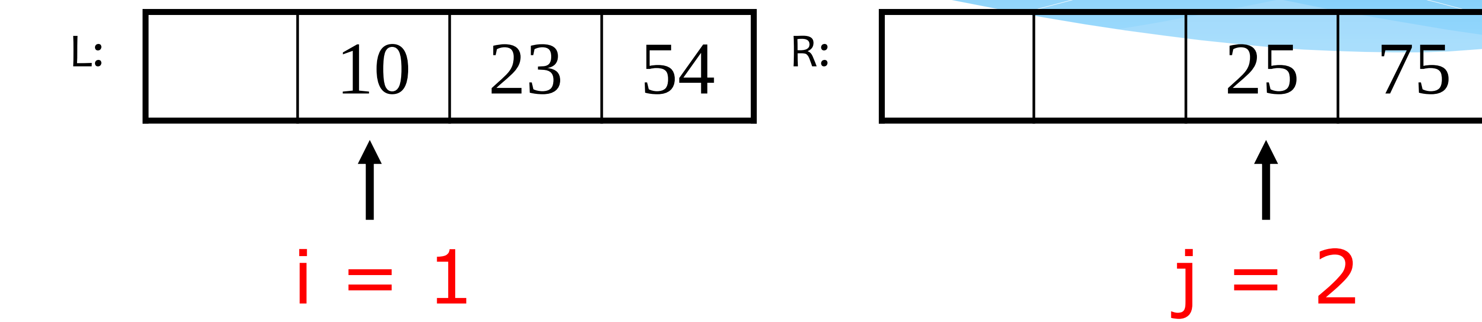

```
i = 0; j = 0; k =0;

while(i<leftCount && j < rightCount)
{
    if(L[i] < R[j]){
        A[k] = L[i];
        k++; i++;
    }

    else{
        A[k] = R[j];
        k++; j++;
    }
}
```

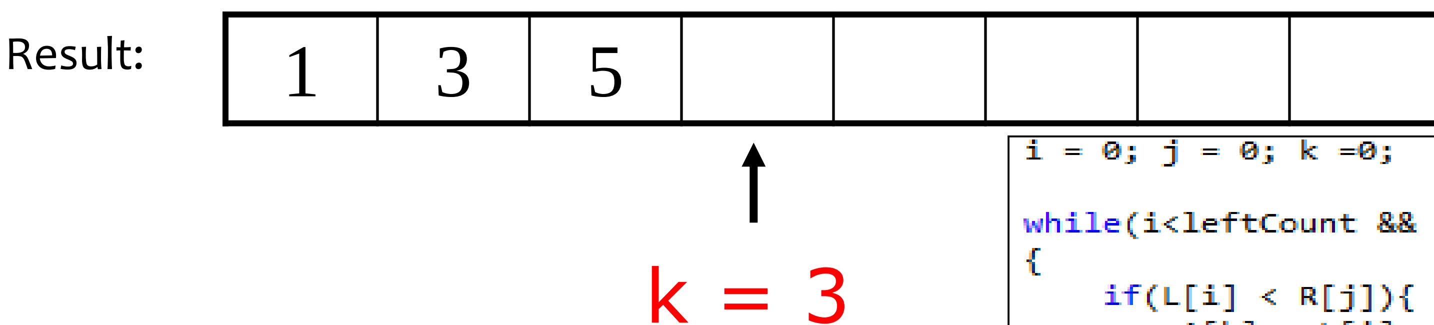
Merging




```
i = 0; j = 0; k =0;

while(i<leftCount && j < rightCount)
{
    if(L[i] < R[j]){
        A[k] = L[i];
        k++; i++;
    }

    else{
        A[k] = R[j];
        k++; j++;
    }
}
```

Merging






```
i = 0; j = 0; k =0;

while(i<leftCount && j < rightCount)
{
    if(L[i] < R[j]){
        A[k] = L[i];
        k++; i++;
    }

    else{
        A[k] = R[j];
        k++; j++;
    }
}
```

Merging

L:

		23	54

R:

		25	75

i = 2

j = 2

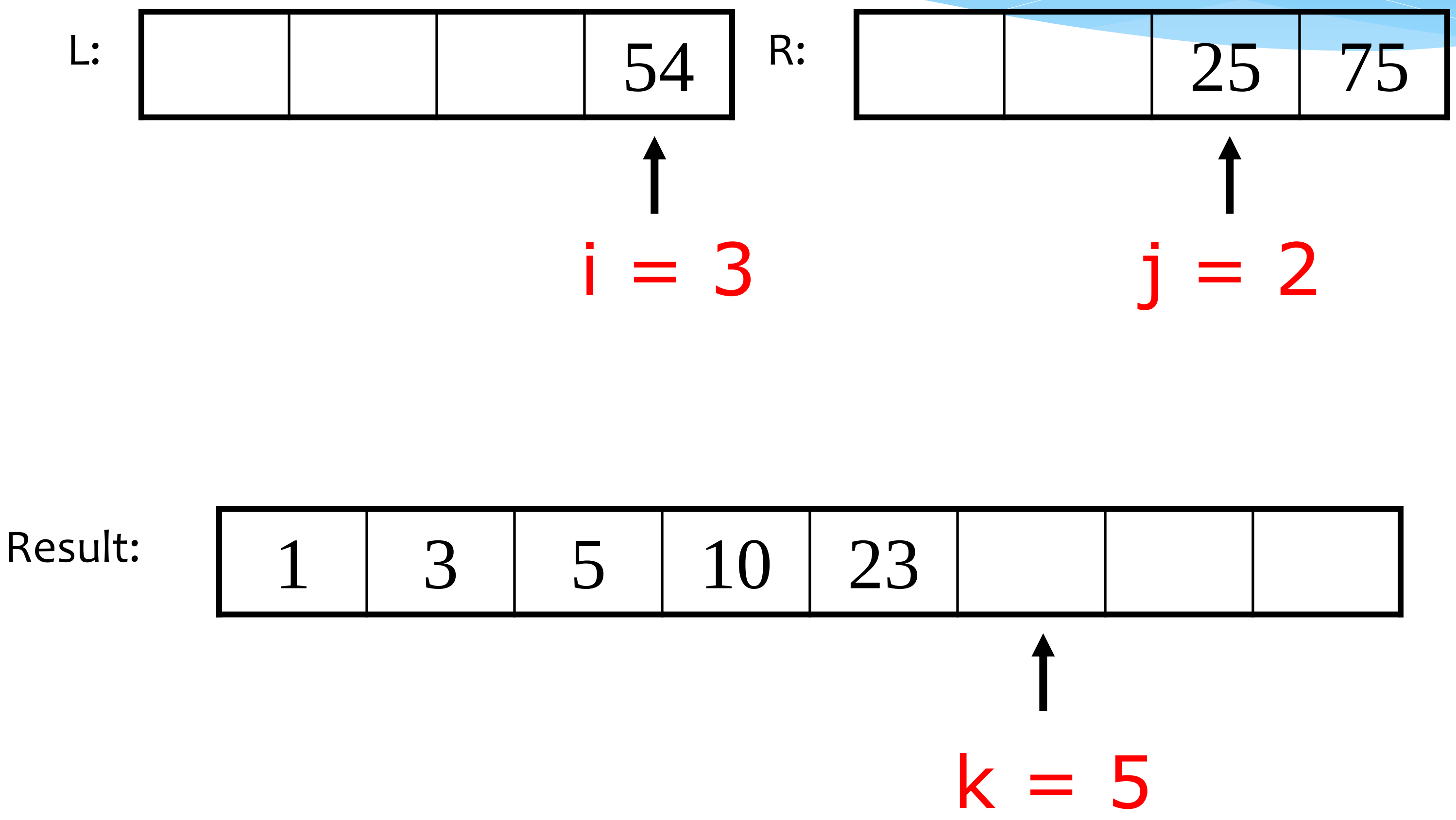
Result:

1	3	5	10				

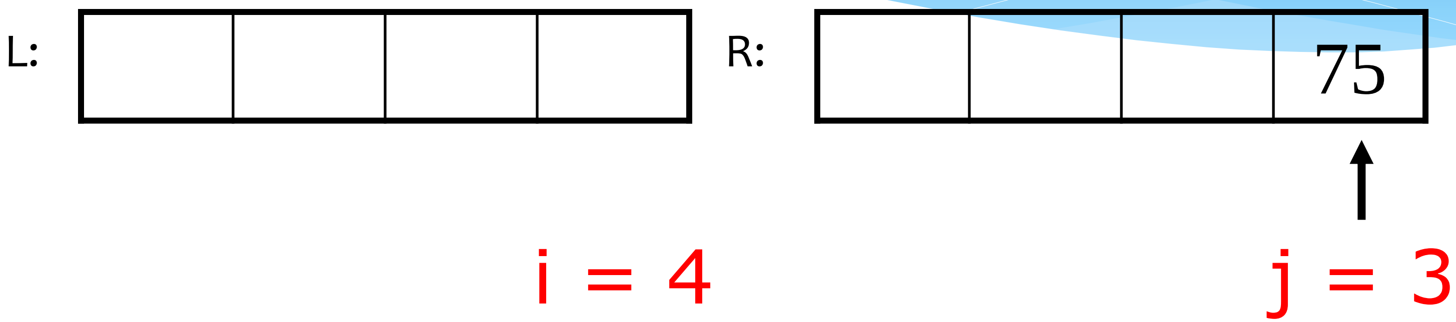
k = 4

```
i = 0; j = 0; k =0;

while(i<leftCount && j < rightCount)
{
    if(L[i] < R[j]){
        A[k] = L[i];
        k++; i++;
    }

    else{
        A[k] = R[j];
        k++; j++;
    }
}
```

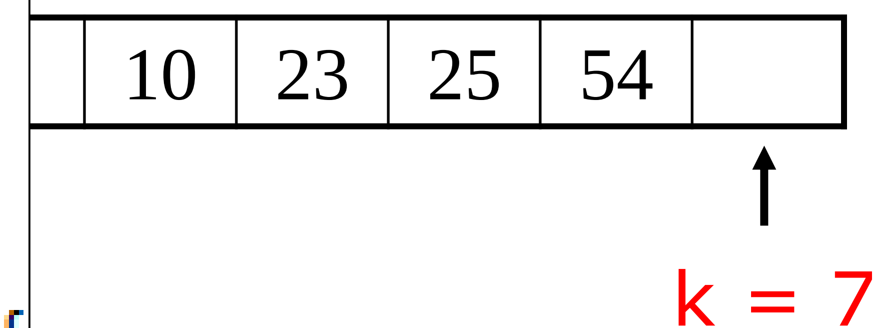
Merging



Merging




```
while(i < leftCount){
    A[k] = L[i];
    k++; i++;
}

while(j < rightCount){
    A[k] = R[j];
    k++; j++;
}
```



Merging

L:

R:

i = 4

j = 4

Result:

1	3	5	10	23	25	54	75

k = 7

MERGE SORT

```
void MergeSort(int *A,int n) {
    int mid, i, *L, *R;

    if(n < 2)
        return; // base condition. If the array has less than two element, do nothing.

    mid = n/2; // find the mid index.

    // create left and right subarrays
    // mid elements (from index 0 till mid-1) should be part of left sub-array
    // and (n-mid) elements (from mid to n-1) will be part of right sub-array

    L = (int*)malloc(mid*sizeof(int));
    R = (int*)malloc((n - mid)*sizeof(int));

    for(i = 0;i<mid;i++)
        L[i] = A[i]; // creating left subarray

    for(i = mid;i<n;i++)
        R[i-mid] = A[i]; // creating right subarray

    MergeSort(L,mid); // sorting the left subarray
    MergeSort(R,n-mid); // sorting the right subarray

    Merge(A,L,mid,R,n-mid); // Merging L and R into A as sorted list.
}
```

MERGE SORT

```
void Merge(int *A,int *L,int leftCount,int *R,int rightCount) {
	int i,j,k;

	i = 0; j = 0; k =0;

	while(i<leftCount && j < rightCount)
	{
		if(L[i] < R[j]){
			A[k] = L[i];
			k++; i++;
		}

		else{
			A[k] = R[j];
			k++; j++;
		}
	}

	while(i < leftCount){
		A[k] = L[i];
		k++; i++;
	}

	while(j < rightCount){
		A[k] = R[j];
		k++; j++;
	}
}
```

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	◈ slow ◈ in-place ◈ for small data sets (< 1K)
insertion-sort	$O(n^2)$	◈ slow ◈ in-place ◈ for small data sets (< 1K)
merge-sort	$O(n \log n)$	◈ fast ◈ sequential data access ◈ for huge data sets (> 1M)