Introduction to Algorithm

**Definition**

* Algorithm is any well-defined computational procedure.
* It takes some value, or set of values as input and produces some values or set of values as output.
* It is a tool for solving computation problem.
* **Example:** Sorting algorithm for sorting problem.

**Insertion Sort**

* Insertion sort is a sorting algorithm
* **Input**: a sequence of n numbers- 
* **Output:** Reordering of input sequences  such that .

|  |
| --- |
| **INSERTION-SORT(A)** |

1. for j = 2 to A.length
2. key = A[j]
3. //Inset A[j] into the sorted sequence A[1 … j-1]
4. i = j - 1
5. while i > 0 and A[i] > key
6. A[i+1] = A[i]
7. i = i – 1
8. A[i+1] = key

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 5 | 2 | 4 | 6 | 1 | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 5 | 4 | 6 | 1 | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 6 | 1 | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 4 | 5 | 6 | 1 | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 4 | 5 | 6 | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |

**Running Time:**

|  |
| --- |
| **INSERTION-SORT(A) Cost Time** |

1. for j = 2 to A.length C1 n
2. key = A[j] C2 n-1
3. //Inset A[j] into the sorted sequence A[1 … j-1] 0 n-1
4. i = j – 1 C4 n-1
5. while i > 0 and A[i] > key C5 
6. A[i+1] = A[i] C6 
7. i = i – 1 C7 
8. A[i+1] = key C8 n-1

Running Time & Space Complexity

* Computer is fast but it is not infinitely fast.
* If computer were infinitely fast any correct algorithm would do.
* As computer is not infinitely fast, so algorithm should be designed to finish within expected time.
* Memory of computer is not unlimited and not free rather computer memory is fast, limited and costly.
* Algorithm should be designed that should use expected extra memory.

Kinds of Runtime Analysis

* **Worst-case:** (usually)
  + *T*(*n*) = maximum time of algorithm on any input of size *n*.
* **Average-case:** (sometimes)
  + *T*(*n*) = expected time of algorithm over all inputs of size *n*.
  + Need assumption of statistical distribution of inputs.
* **Best-case:** 
  + Cheat with a slow algorithm that works fast on *some* input**.**



Asymptotic Notation - O-notation

|  |
| --- |
|  |

ORDER:

**DEF:** O(*g*(*n*)) = {*f* (*n*): there exist positive constants *c* and *n*0 such that 0 ≤ *f* (*n*) ≤ *c* *g*(*n*) for all *n*  ≥ *n*0 }

**Basic manipulations:**



* Drop low-order terms; ignore leading constants.
* Example: 3*n*3 + 10*n2* – 5*n* + 1 = o(*n*3)

Asymptotic Notation - Ω-notation

|  |
| --- |
|  |

ORDER:

**DEF:** Ω (*g*(*n*)) = { *f* (*n*): there exist positive constants *c* and *n*0 such that 0 ≤ *c* *g*(*n*) ≤ *f* (*n*) for all *n*  ≥ *n*0 }

**Basic manipulations:**

* Drop low-order terms; ignore leading constants.
* Example: 3*n*3 + 10*n2* – 5*n* + 1 = (*n*3)

Asymptotic Notation - -notation

|  |
| --- |
|  |

ORDER:

**DEF:** (*g*(*n*)) = { *f* (*n*): there exist positive constants *c* and *n*0 such that 0 ≤ c1 g(n) ≤ f (n) ≤ c2g(n) for all n ≥ n0 }

**Basic manipulations:**

* Drop low-order terms; ignore leading constants.
* Example: 3*n*3 + 10*n2* – 5*n* + 1 = (*n*3)

Asymptotic Notation(Example):

|  |  |
| --- | --- |
| |  | | --- | | **Analysis of Insertion Sort:**  **INSERTION-SORT(A) Cost Time** |   for j = 2 to A.length C1 n  key = A[j] C2 n-1  //Inset A[j] into the sorted sequence A[1 … j-1] 0 n-1  i = j – 1 C4 n-1  while i > 0 and A[i] > key C5  A[i+1] = A[i] C6  i = i – 1 C7  A[i+1] = key C8 n-1 |

|  |
| --- |
|  |

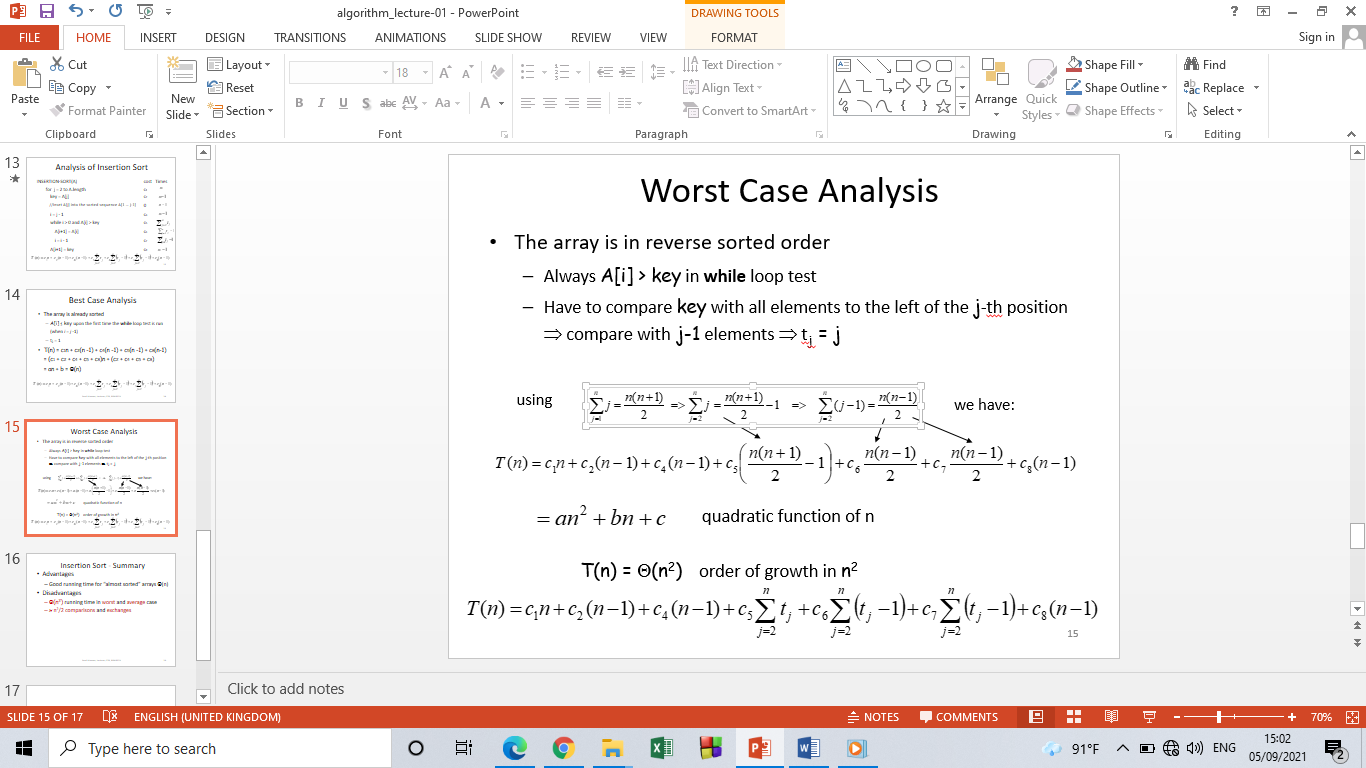
**Best Case Analysis:**

* The array is already sorted
  + A[i] ≤ key upon the first time the **while** loop test is run (when *i* = *j* -1)
  + tj= 1
* T(n) = c1n + c2(n -1) + c4(n -1) + c5(n -1) + c8(n-1) = (c1 + c2 + c4 + c5 + c8)n + (c2 + c4 + c5 + c8)

= an + b = Θ(n)

* The array is in reverse sorted order
  + Always A[i] > key in **while** loop test
  + Have to compare keywith all elements to the left of the j*-*th position

⇒ compare with j-1 elements ⇒ tj = j



T(n) = Θ(n2) order of growth in n2



Insertion Sort – Summary

* **Advantages**
  + Good running time for “almost sorted” arrays Θ(n)
* **Disadvantages**
  + Θ(n2) running time in worst and average case
  + ≈ n2/2 comparisons and exchanges

Divide and Conquer

* Divide-and conquer is a general algorithm design paradigm:
  + Divide: divide the input data ***S*** in two or more disjoint subsets ***S***1***, S***2, …
  + Recur: solve the sub problems recursively
  + Conquer: combine the solutions for ***S***1***,*** ***S***2, …, into a solution for ***S***
* The base case for the recursion are sub problems of constant size
* Analysis can be done using **recurrence equations**

**Quick Sort:**

QUICKSORT(A,p,r) : 1. if(p<r)

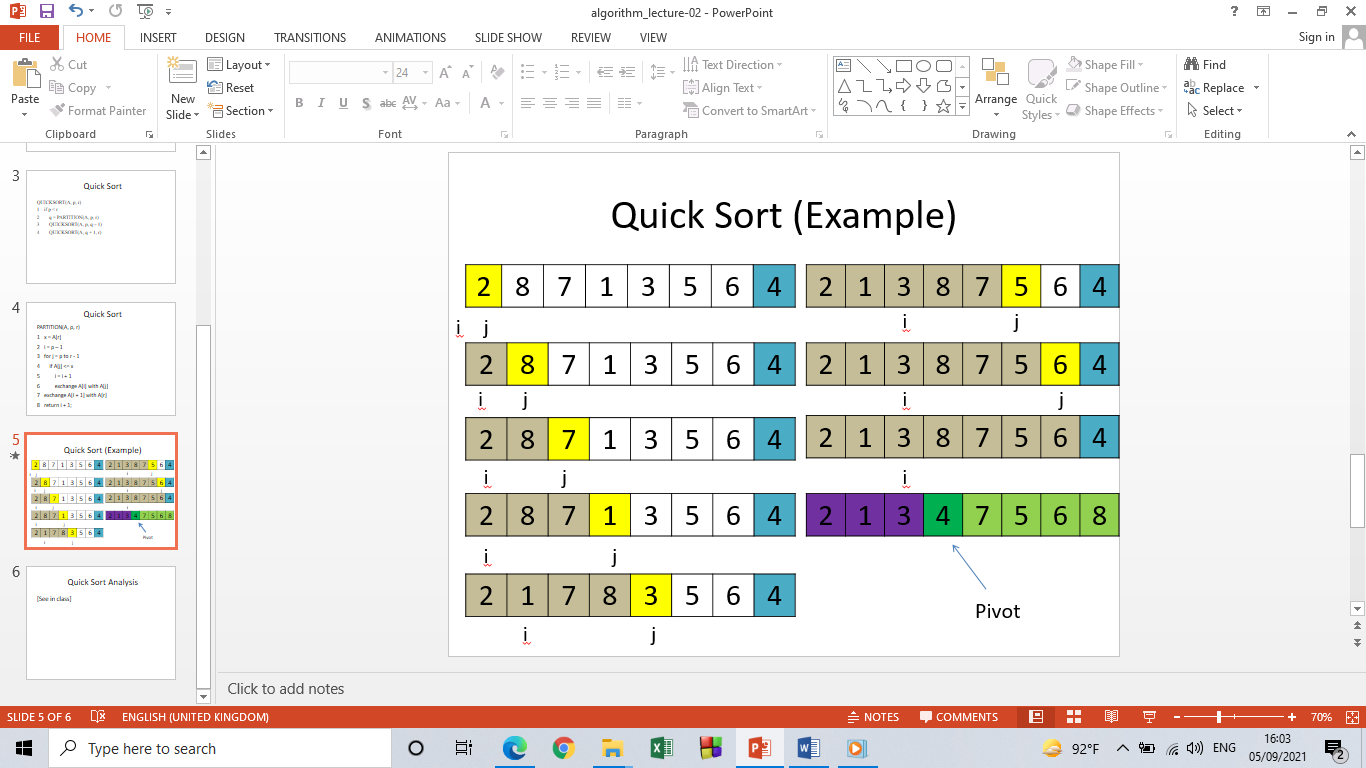
2. q=PARTITION(A,p,r)

3. QUICKSORT(A,p,q-1)

4. QUICKSORT(A,q+1,r)

PARTITION(A,p,r):

1. x= A[r]
2. i = p-1
3. for j=p to r-1
4. if A[j] ≤ x
5. i = i+1
6. exchange A[i] with A[j]
7. exchange A[l+1] with A[r]
8. Return i+1



**Merge Sort:**

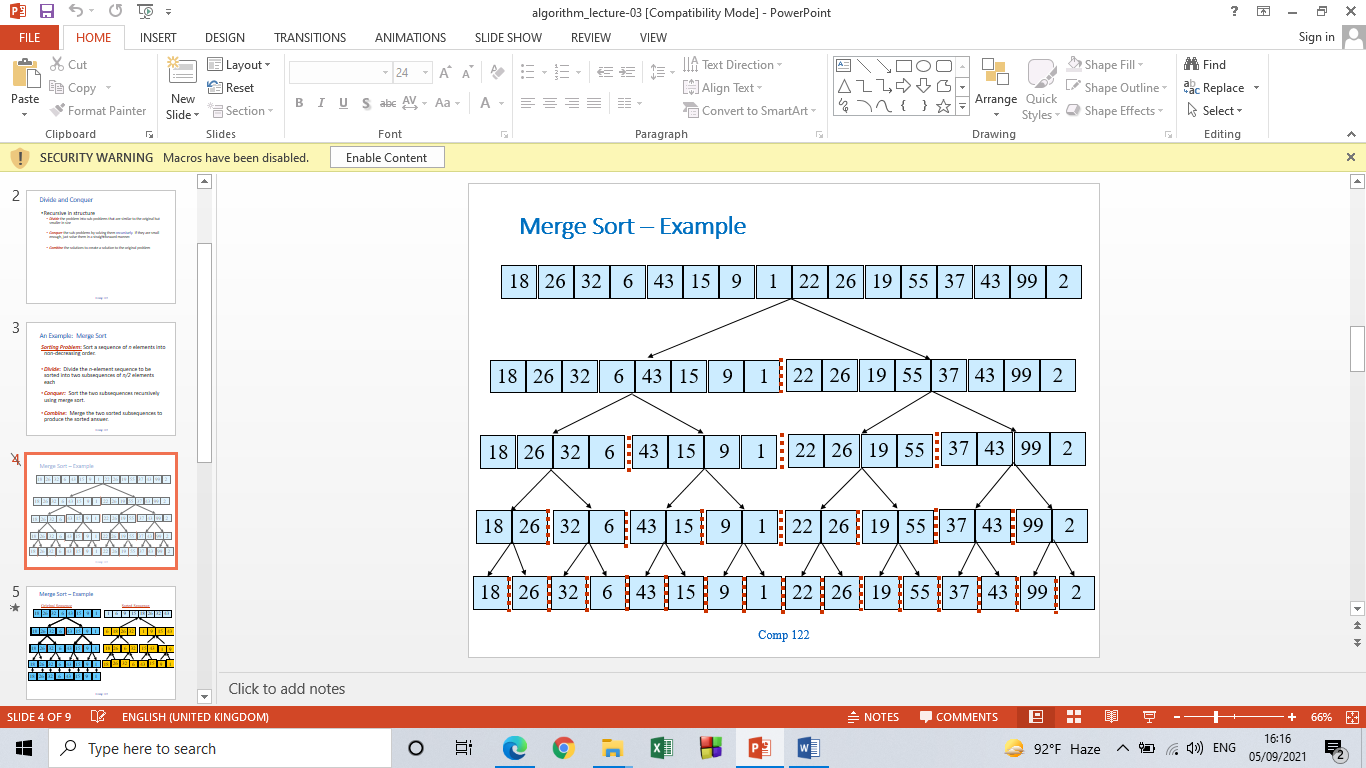
Recursive in structure

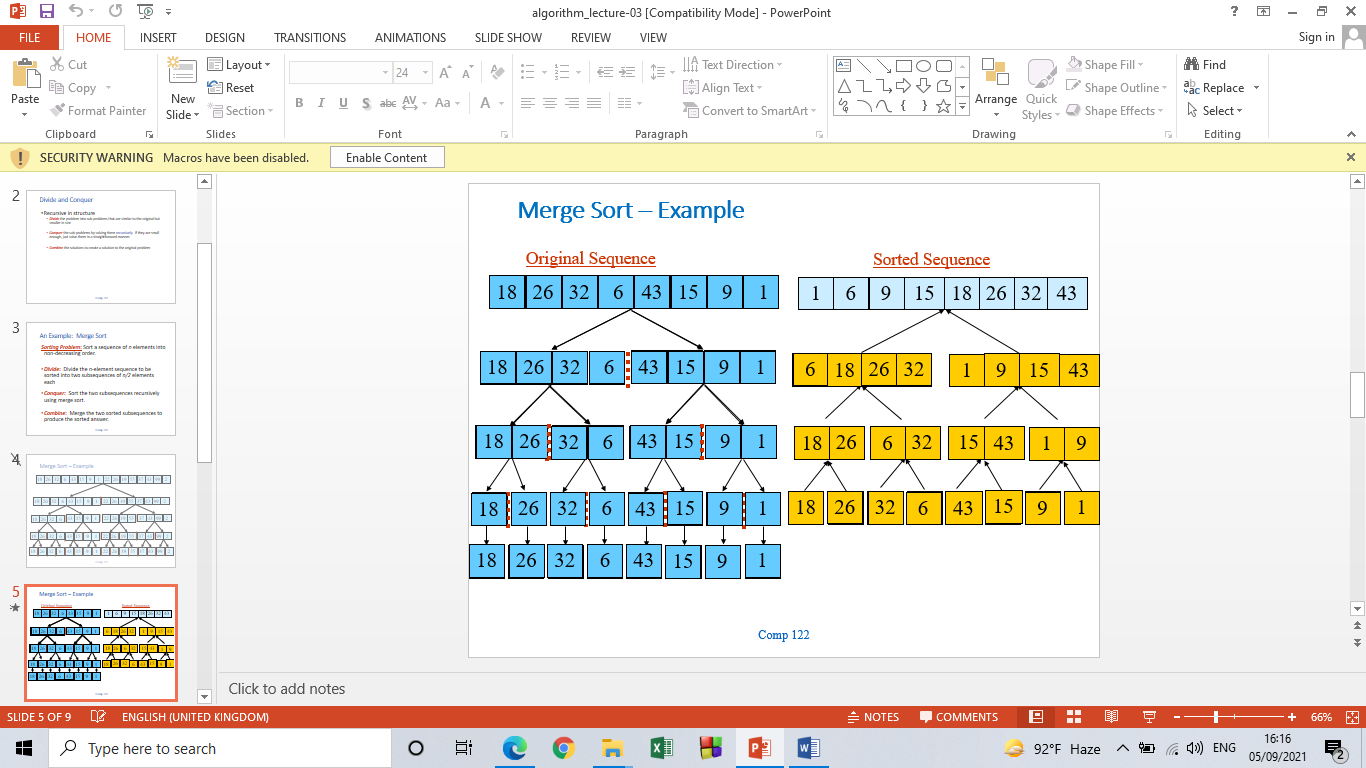
* + ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  + ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  + ***Combine*** the solutions to create a solution to the original problem

**An Example:**

***Sorting Problem*:** Sort a sequence of *n* elements into non-decreasing order.

* **Divide:** Divide the n-element sequence to be sorted into two subsequences of n/2 elements each
* **Conquer:** Sort the two subsequences recursively using merge sort.
* **Combine:** Merge the two sorted subsequences to produce the sorted answer.

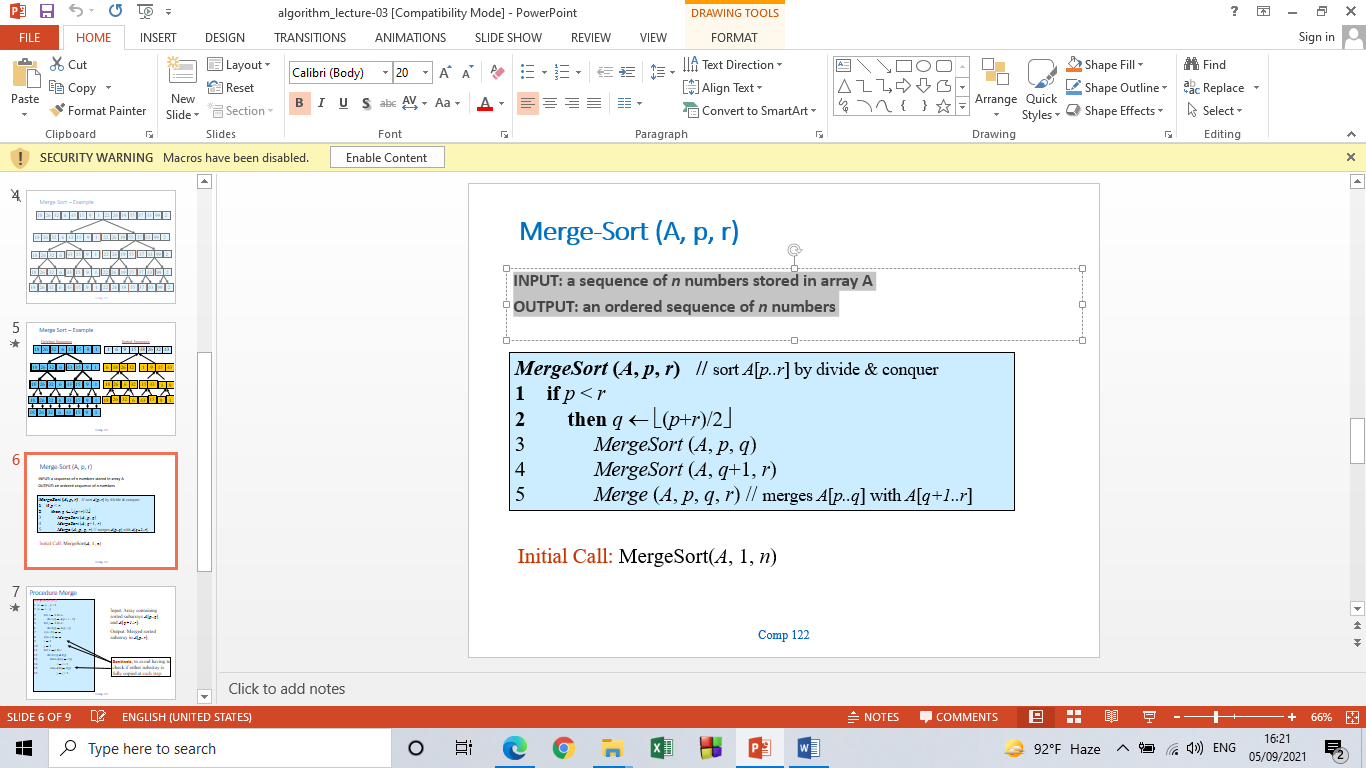




**Merge-Sort (A, p, r)**

**INPUT:** a sequence of *n* numbers stored in array A

**OUTPUT:** an ordered sequence of *n* numbers



***MergeSort* (*A*, *p*, *r*) //** sort *A*[*p..r*] by divide & conquer

1. **If p<r**
2. **Then q**←⎣(*p*+*r*)/2⎦
3. *MergeSort* (*A*, *p*, *q*)
4. *MergeSort* (*A*, *q*+1, *r*)
5. *Merge* (*A*, *p*, *q*, *r*) // merges *A*[*p..q*] with *A*[*q+1..r*]

Initial Call: MergeSort(*A*, 1, *n*)

**Procedure Merge:**

**Merge(*A*, *p*, *q*, *r*)**

|  |
| --- |
| 1. n1 ← q – p + 1 2. n2 ← r – q 3. for i ← 1 to n1 4. do L[i] ← A[p + i – 1] 5. for j ← 1 to n2 6. do R[j] ← A[q + j] 7. L[n1+1] ← ∞ 8. R[n2+1] ← ∞ 9. i ← 1 10. *j* ← 1 11. for *k* ←*p* to *r* 12. do if *L*[*i*] ≤ *R*[*j*] 13. then *A*[*k*] ← *L*[*i*] 14. . *i* ← *i* +1 15. Else A[k] ← R[j] 16. j← j+1 |

Input: Array containing sorted subarrays *A*[*p..q*] and *A*[*q+1..r*].

Output: Merged sorted subarray in *A*[*p..r*].

|  |
| --- |
| **Sentinels**, to avoid having to check if either subarray is fully copied at each step. |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| …… | 1 | 6 | 8 | 9 | 26 | 32 | 42 | 43 | ……. |

A

k

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 8 | 26 | 32 | ∞ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 9 | 42 | 43 | ∞ |

L R

i j

**Analysis of Merge Sort:**