



UNIT-V

SEARCHING AND SORTING

PREPARED BY:
DHAVAL R. GANDHI
LECTURER IN I.T.

LEARNING OUTCOMES

- ❑ SEARCHING AN ELEMENT INTO LIST:

- ❑ LINEAR SEARCH

- ❑ BINARY SEARCH

- ❑ SORTING METHODS:

- ❑ BUBBLE SORT

- ❑ SELECTION SORT

- ❑ QUICK SORT

- ❑ INSERTION SORT

- ❑ MERGE SORT

SEARCHING

- ❑ Searching is a technique to find the particular element is present or not in the given list.
- ❑ There are two types of searching -
 - ❑ Linear Search
 - ❑ Binary Search
- ❑ Both techniques are widely used to search an element in the given list.

LINEAR SEARCHING

- ❑ Linear search is a method of finding elements within a list.
- ❑ It is also called a sequential search.
- ❑ It is the simplest searching algorithm because it searches the desired element in a sequential manner.
- ❑ Linear search work on both sorted as well as unsorted list.
- ❑ This method of searching works as follows:
- ❑ It compares each and every element with the value that we are searching for. If both are matched, the element is found, and the algorithm returns the key's index position.

LINEAR SEARCH EXAMPLE

- ❑ Let's understand the following steps to find the element key = 7 in the given list.
- ❑ Step - 1: Start the search from the first element and Check key = 7 with each element of list x.

1	3	5	4	7	9
---	---	---	---	---	---

List to be Searched for

- ❑ Step - 2: If element is found, return the index position of the key.
- ❑ Step - 3: If element is not found, return element is not present.

LINEAR SEARCH EXAMPLE



↑
 $k \neq 7$



↑
 $k \neq 7$



↑
Key=7



↑
 $k \neq 7$



Key=7

LINEAR SEARCH ALGORITHM

```
❑ LINEARSEARCH(LIST, KEY)
  FOREACH ITEM IN THE LIST
    IF ITEM == VALUE
      RETURN ITS INDEX POSITION
  RETURN -1
```

LINEAR SEARCH PROGRAM

linear_search.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/searching_

File Edit Format Run Options Window Help

```
def linear_Search(list1, n, key):  
    # Searching list1 sequentially  
    for i in range(0, n):  
        if (list1[i] == key):  
            return i  
    return -1
```

Element found at index: 4

```
list1 = [1 ,3, 5, 4, 7, 9]  
key = 7
```

```
n = len(list1)  
res = linear_Search(list1, n, key)  
if(res == -1):  
    print("Element not found")  
else:  
    print("Element found at index: ", res)
```


LINEAR SEARCH

- ❑ Time complexity of linear search algorithm -
- ❑ **Base Case - $O(1)$**
- ❑ **Average Case - $O(n)$**
- ❑ **Worst Case - $O(n)$**
- ❑ Linear search algorithm is suitable for smaller list (<100) because it check every element to get the desired number.
- ❑ Suppose there are 10,000 element list and desired element is available at the last position, this will consume much time by comparing with each element of the list.
- ❑ To get the fast result, we can use the binary search algorithm.

BINARY SEARCH

- ❑ A binary search is an algorithm to find a particular element in the list. Suppose we have a list of thousand elements, and we need to get an index position of a particular element.
- ❑ We can find the element's index position very fast using the binary search algorithm.
- ❑ The elements in the list must be sorted to apply the binary search algorithm. If elements are not sorted then sort them first.
- ❑ In the binary search algorithm, we can find the element position using the following methods.
- ❑ Recursive Method
- ❑ Iterative Method

BINARY SEARCH

- ❑ The divide and conquer approach technique is followed by the recursive method.
- ❑ A set of statements is repeated multiple times to find an element's index position in the iterative method.
- ❑ Binary search is more effective than the linear search because we don't need to search each list index.

BINARY SEARCH

Steps for find X from list of N elements:-

- ❑ Determine the Lower and Upper limit of the list by assigning Lower index of the list to LOW and Upper Index of the list to HIGH.
- ❑ **LOW = 0 and HIGH = len(list1) - 1**
- ❑ Now calculate the MIDDLE position By:
MIDDLE = (LOW + HIGH)/2.
- ❑ Then we compare the value of the MIDDLE element with X.
- ❑ If **list1[Middle] > X** so it will exist in the lower interval of the list, so **HIGH=M-1.**
- ❑ If **list1[Middle] < X** so it will exist in the upper interval of the list, so **Low=M+1.**

BINARY SEARCH EXAMPLE


LOW				MIDDLE					HIGH
↓				↓					↓
0	1	2	3	4	5	6	7	8	9
23	45	67	89	123	189	210	235	345	545

- ❑ $LOW = 0, HIGH = 9$
- ❑ $MIDDLE = (LOW + HIGH)/2 = (0+9)/2 = 4.5 = 4$
- ❑ $1[4]=123 < 189$ so, $LOW = MIDDLE + 1 = 4 + 1 = 5$.

LOW		MIDDLE		HIGH
↓		↓		↓
5	6	7	8	9
189	210	235	345	545

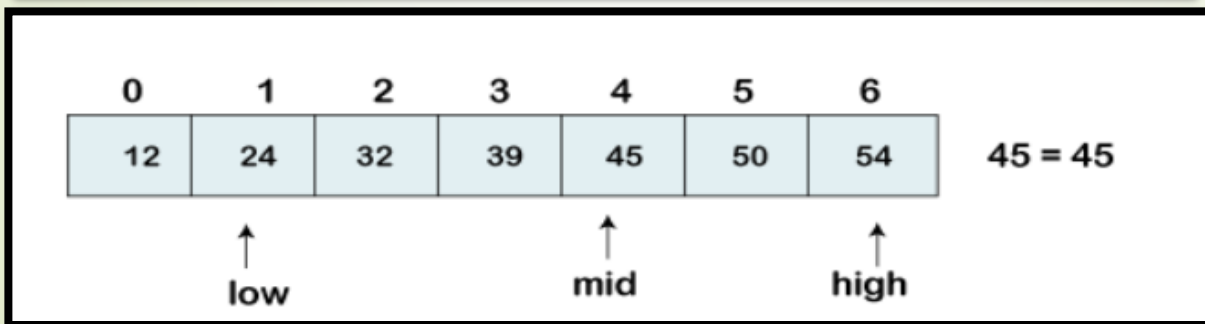
BINARY SEARCH EXAMPLE

- ❑ $LOW = 5, HIGH = 9$
- ❑ $Middle = (LOW + HIGH) / 2 = (5 + 9) / 2 = 14 / 2 = 7$
- ❑ So, $l[7] = 235 > 189$ so, $HIGH = MIDDLE - 1 = 7 - 1 = 6$.

LOW	HIGH
	
5	6
189	210

- ❑ $LOW = 5, HIGH = 6$
- ❑ $Middle = (LOW + HIGH) / 2 = (5 + 6) / 2 = 11 / 2 = 5.5 = 5$
- ❑ Here the value of MIDDLE element is 189 which is equal to 189. So the element is found in the list.

Search 45



BINARY SEARCH ALGORITHM

`binary_search(list1, n):`

Step 1: `low = 0 ,high = len(list1) - 1 ,mid = 0`

Step-2: Repeat upto step-4 while `low <= high`:

`# for get integer result`

Step-3: `mid = (high + low) // 2`

`# Check if n is present at mid`

Step-4: `if list1[mid] < n:`

`low = mid + 1`

`# If n is greater, compare to the right of mid`

`elif list1[mid] > n:`

`high = mid - 1`

`# If n is smaller, compared to the left of mid`

`else:`

`return mid`

Step-5 : Write "Search is not successful"

`return -1`

BINARY SEARCH PROGRAM

binary_search.py - D:/dhavalPC_backup_9-10-18/IT_15/Term-221/4331601/Python_prog/searching_sorting/binary_search.py (3.10.1)

File Edit Format Run Options Window Help

```
def binary_search(list1, n):
    low = 0
    high = len(list1) - 1
    mid = 0
    while low <= high:
        # for get integer result
        mid = (high + low) // 2
        # Check if n is present at mid
        if list1[mid] < n:
            low = mid + 1
        # If n is greater, compare to the right of mid
        elif list1[mid] > n:
            high = mid - 1
        # If n is smaller, compared to the left of mid
        else:
            return mid |
            # element was not present in the list, return -1
    return -1

# Initial list1
list1 = [12, 24, 32, 39, 45, 50, 54]
n = 45

# Function call
result = binary_search(list1, n)

if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in list1")
```

Element is present at index 4

BINARY SEARCH

- ❑ The complexity of the binary search algorithm is $O(1)$ for the best case. This happens if the element that we are looking for is found in the first comparison.
- ❑ The $O(\log n)$ is the worst and the average case complexity of the binary search.
- ❑ A binary search algorithm is the most efficient and fast way to search an element in the list.
- ❑ It skips the unnecessary comparison. As the name suggests, the search is divided into two parts.
- ❑ It focuses on the side of the list, which is close to the number that we are searching.

PRACTICE

- ❑ Consider the list $l1 [] = \{64, 105, 211, 245, 295, 370, 404, 489, 512, 525\}$.
- ❑ Search 245 from this array with the help of binary search and state required number of iterations.

LINEAR VS BINARY SEARCH

Basis of comparison	Linear search	Binary search
Definition	The linear search starts searching from the first element and compares each element with a searched element till the element is not found.	It finds the position of the searched element by finding the middle element of the array.
Sorted data	In a linear search, the elements don't need to be arranged in sorted order.	The pre-condition for the binary search is that the elements must be arranged in a sorted order.
Approach	It is based on the sequential approach.	It is based on the divide and conquer approach.
Size	It is preferable for the small-sized data sets.	It is preferable for the large-size data sets.
Efficiency	It is less efficient in the case of large-size data sets.	It is more efficient in the case of large-size data sets.
Worst-case scenario	In a linear search, the worst-case scenario for finding the element is $O(n)$.	In a binary search, the worst-case scenario for finding the element is $O(\log^2 n)$.
Best-case scenario	In a linear search, the best-case scenario for finding the first element	In a binary search, the best-case scenario for finding the first element

SORTING

- ❑ **Sorting** is a process that organizes a collection of data into either ascending or descending order.
- ❑ Two Types of Sorting
 - ❑ **Internal Sort**
 - ❑ **External Sort**
- ❑ An **internal sort** requires that the collection of data fit entirely in the computer's main memory.
- ❑ It is useful for sorting fewer amounts of elements.
- ❑ We can use an **external sort** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.
- ❑ It is useful when we have to sort large amount of elements.

SORTING

Features of Sorting:-

- ❑ Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
- ❑ Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- ❑ Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.
- ❑ A comparison-based sorting algorithm makes ordering decisions only on the basis of comparisons.

SORTING ALGORITHMS

Internal Sort:-

- ☐ Bubble Sort
- ☐ Selection Sort
- ☐ Insertion Sort
- ☐ Quick Sort

External Sort:-

- ☐ Merge Sort
- ☐ Radix Sort

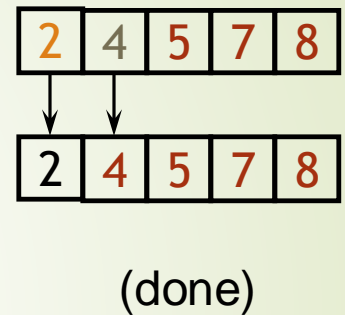
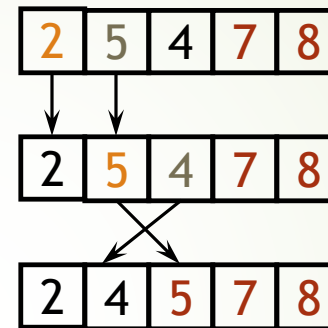
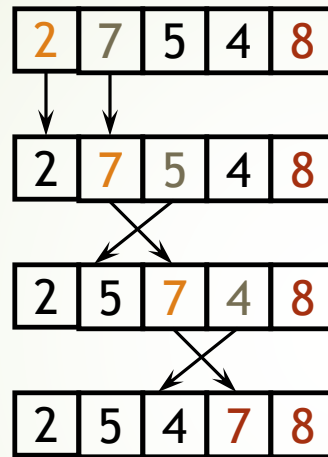
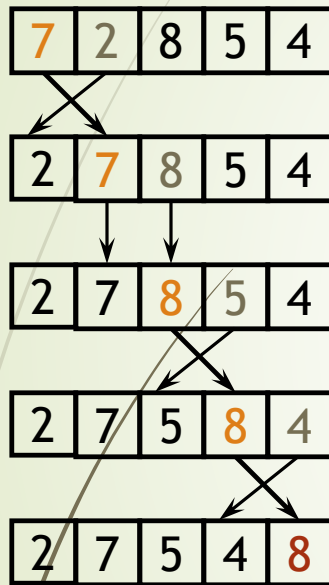
BUBBLE SORT

- ☐ It is simple sorting method.
- ☐ It works fine for smaller number of elements.
- ☐ Given a list of n elements, bubble sort requires up to $n-1$ passes to sort the data.
- ☐ During first pass 1st and 2nd element are compared
- ☐ If 1st element $>$ 2nd element then swapped.
- ☐ now 2nd element and third element compared.
- ☐ If 2nd element $>$ 3rd element then swapped.
- ☐ So after 1st pass largest value is placed at its proper position.

BUBBLE SORT

- ❑ Compare each element (except the last one) with its neighbor to the right
 - If they are out of order, swap them
 - This puts the largest element at the very end
 - The last element is now in the correct and final place
- ❑ Compare each element (except the last *two*) with its neighbor to the right
 - If they are out of order, swap them
 - This puts the second largest element next to last
 - The last two elements are now in their correct and final places
- ❑ Compare each element (except the last *three*) with its neighbor to the right
 - Continue as above until you have no unsorted elements on the left.

BUBBLE SORT



BUBBLE SORT ALGORITHM

```
Step-1 : length <- len(list1)
Step-2 : Repeat upto step 4 for I in range(length)
Step-3 : Repeat Step 4 for J in range(length-I-1)
Step-4 : if list1[J] > list1[J+1] then
         temp <- list1[J]
         list1[J] <- list1[J+1]
         list1[J+1] <- temp
Step-5 : Exit
```

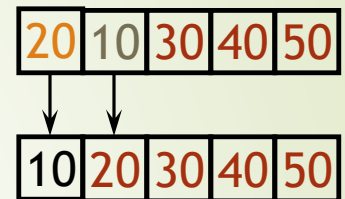
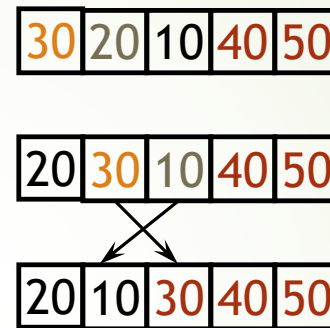
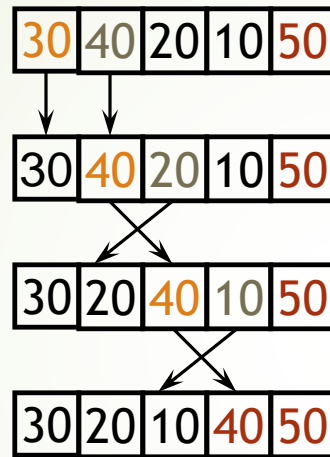
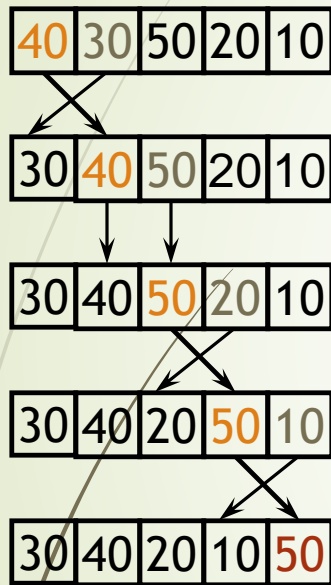
BUBBLE SORT PROGRAM

```
# Creating a bubble sort function
def bubble_sort(list1):
    length=len(list1)
    # Outer loop for traverse the entire list
    for i in range(length):
        for j in range(length-i-1):
            if(list1[j]>list1[j+1]):
                temp = list1[j]
                list1[j] = list1[j+1]
                list1[j+1] = temp
    return list1

list1 = [5, 3, 8, 6, 7, 2]
print("The unsorted list is: ", list1)
# Calling the bubble sort function
print("The sorted list is: ", bubble_sort(list1))
```

The unsorted list is: [5, 3, 8, 6, 7, 2]
The sorted list is: [2, 3, 5, 6, 7, 8]

BUBBLE SORT



(done)

BUBBLE SORT – ANALYSIS

❑ **Best-case:** → $O(n)$

- Array is already sorted in ascending order.
- The number of moves: 0 → $O(1)$
- The number of key comparisons: $(n-1)$ → $O(n)$

❑ **Worst-case:** → $O(n^2)$

- Array is in reverse order:
- Outer loop is executed $n-1$ times,
- The number of moves: $3*(1+2+\dots+n-1) = 3 * n*(n-1)/2$ → $O(n^2)$
- The number of key comparisons: $(1+2+\dots+n-1) = n*(n-1)/2$ → $O(n^2)$

❑ **Average-case:** → $O(n^2)$

- We have to look at all possible initial data organizations.

❑ **So, Bubble Sort Time Complexity is $O(n^2)$**

5 2 35 9 3 -2 52 0

Example of Bubble Sort

	Pass1	Pass2	Pass3	Pass4	Pass5	Pass6	Pass7
5	2	2	2	2	-2	-2	-2
2	5	5	3	-2	2	0	0
35	9	3	-2	3	0	2	2
9	3	-2	5	0	3	3	3
3	-2	9	0	5	5	5	5
-2	35	0	9	9	9	9	9
52	0	35	35	35	35	35	35
0	52	52	52	52	52	52	52

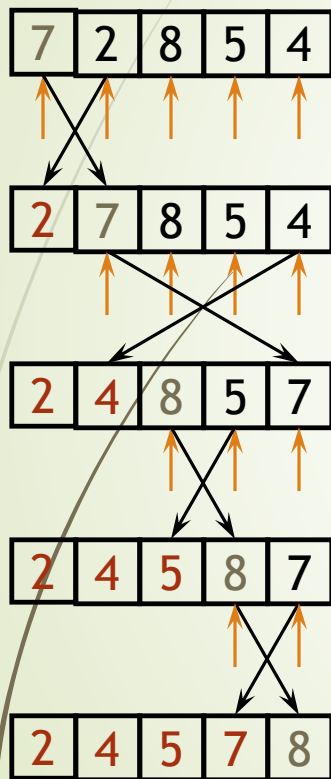
SELECTION SORT

- ❑ It is simple sorting method.
- ❑ It works fine for smaller number of elements.
- ❑ Given a list of n elements, selection sort requires up to $n-1$ passes to sort the data.
- ❑ We find the smallest element from the list and swap it with the first element at the beginning of list. so element with smallest value is placed at first position in array.
- ❑ During second pass array is searched from second record to find second smallest element. When this element is found it is swapped with second element in array.
- ❑ During each successive pass smallest element is placed at its proper position.

SELECTION SORT

- ❑ Given an array of length n ,
 - Search elements 0 through $n-1$ and select the smallest
 - Swap it with the element in location 0
 - Search elements 1 through $n-1$ and select the smallest
 - Swap it with the element in location 1
 - Search elements 2 through $n-1$ and select the smallest
 - Swap it with the element in location 2
 - Search elements 3 through $n-1$ and select the smallest
 - Swap it with the element in location 3
 - Continue in this fashion until there's nothing left to search

SELECTION SORT



- The selection sort might swap an array element with itself--this is harmless, and not worth checking for
- Analysis:
 - The outer loop executes $n-1$ times
 - The inner loop executes about $n/2$ times on average (from n to 2 times)
 - Work done in the inner loop is constant (swap two array elements)
 - Time required is roughly $(n-1)*(n/2)$
 - You should recognize this as $O(n^2)$

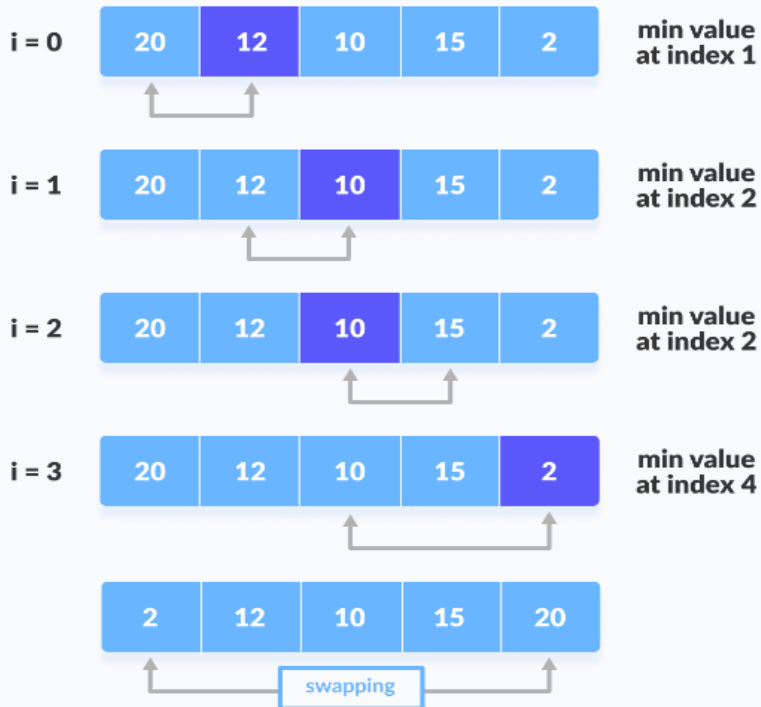
SELECTION SORT ALGORITHM

```
Step-1 : length <- len(list1)
Step-2 : Repeat upto step 6 for I in range(length)
Step-3 : min <- list1[I]
        pos <- I
Step-4 : Repeat Step 4 for J in range(I+1,length)
Step-5 : if list1[J] < min then
        min <- list1[J]
        pos <- J
Step-6 : temp <- list1[I]
        list1[I] <- list1[POS]
        list1[POS] <- temp
Step-7 : Exit
```

SELECTION SORT EXAMPLE

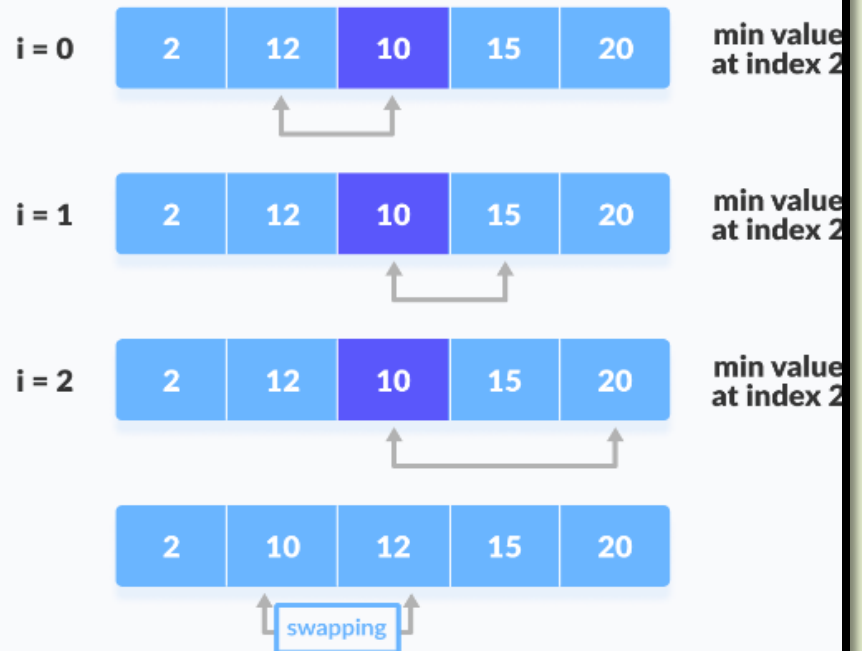
20 12 10 15 2

step = 0



The first iteration

step = 1



The second iteration

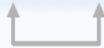
SELECTION SORT ALGORITHM

step = 2

i = 0



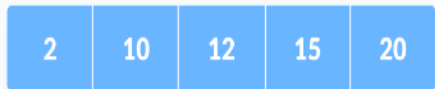
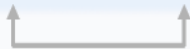
min value
at index 2



i = 2



min value
at index 2

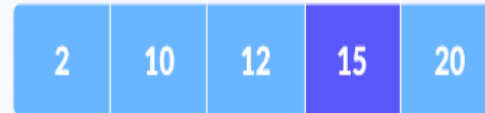


already in place

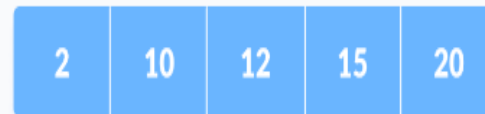
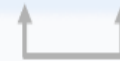
The third iteration

step = 3

i = 0



min value
at index 3



already in place

The fourth iteration

SELECTION SORT ALGORITHM


```
# Creating a selection sort function
def selection_sort(list1):
    length=len(list1)
    k=1
    # Outer loop for traverse the entire list
    for i in range(length):
        MIN=list1[i]
        pos=i
        for j in range(i+1,length):
            if(list1[j]< MIN):
                MIN = list1[j]
                pos=j
        temp=list1[i]
        list1[i]=list1[pos]
        list1[pos]=temp
        print("pass number", k)
        k=k+1
    print(list1)
    return list1

list1 = [5, 3, 8, 6, 7, 2]
print("The unsorted list is: ", list1)
# Calling the bubble sort function
print("The sorted list is: ", selection_sort(list1))
```

The unsorted list is: [5, 3, 8, 6, 7, 2]
pass number 1[2, 3, 8, 6, 7, 5]
pass number 2[2, 3, 8, 6, 7, 5]
pass number 3[2, 3, 5, 6, 7, 8]
pass number 4[2, 3, 5, 6, 7, 8]
pass number 5[2, 3, 5, 6, 7, 8]
pass number 6[2, 3, 5, 6, 7, 8]
The sorted list is: [2, 3, 5, 6, 7, 8]

Sorted

Unsorted



23	78	45	8	32	56
----	----	----	---	----	----

Original List

8	78	45	23	32	56
---	----	----	----	----	----

After pass 1

8	23	45	78	32	56
---	----	----	----	----	----

After pass 2

8	23	32	78	45	56
---	----	----	----	----	----

After pass 3

8	23	32	45	78	56
---	----	----	----	----	----

After pass 4

8	23	32	45	56	78
---	----	----	----	----	----

After pass 5

SELECTION SORT – ANALYSIS

- ❑ In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).
- ❑ → **So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.**
 - ❑ Ignoring other operations does not affect our final result.
- ❑ In selection Sort function, the outer for loop executes $n-1$ times.
- ❑ We invoke swap function once at each iteration.
- ❑ → Total Swaps: $n-1$
- ❑ → Total Moves: $3*(n-1)$ (Each swap has three moves)

SELECTION SORT – ANALYSIS (CONT.)

- ❑ The inner for loop executes the size of the unsorted part minus 1 (from 1 to $n-1$), and in each iteration we make one key comparison.
- ❑ → # of key comparisons = $1+2+\dots+n-1 = n*(n-1)/2$
- ❑ → **So, Selection sort is $O(n^2)$**
- ❑ The best case, the worst case, and the average case of the selection sort algorithm are same. → all of them are **$O(n^2)$**
 - ❑ This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
 - ❑ Since $O(n^2)$ grows so rapidly, the selection sort algorithm is appropriate only for small n .
 - ❑ Although the selection sort algorithm requires $O(n^2)$ key comparisons, it only requires $O(n)$ moves.
 - ❑ A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

5 2 35 9 3 -2 52 0

Example of Selection Sort

	Pass1	Pass2	Pass3	Pass4	Pass5	Pass6	Pass7
5	-2	-2	-2	-2	-2	-2	-2
2	2	0	0	0	0	0	0
35	35	35	2	2	2	2	2
9	9	9	9	3	3	3	3
3	3	3	3	9	5	5	5
-2	5	5	5	5	9	9	9
52	52	52	52	52	52	52	35
0	0	2	35	35	35	35	52

Indicates elements to be compare during each PASS.

Indicates elements which are already sorted.

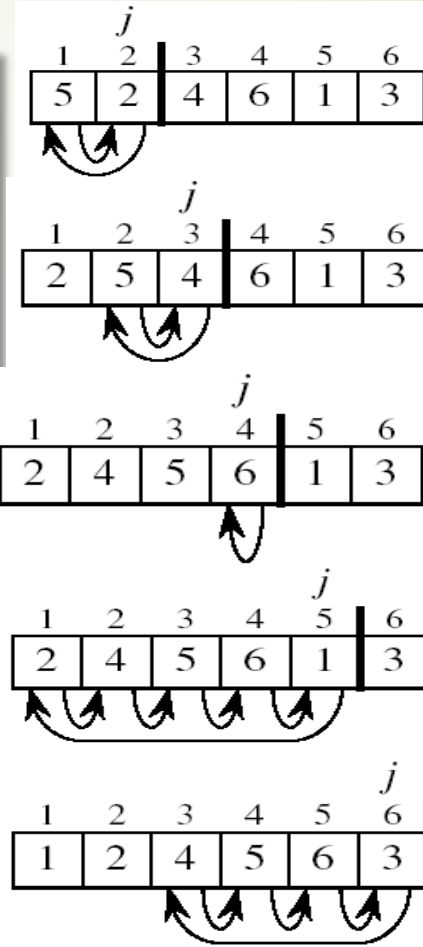
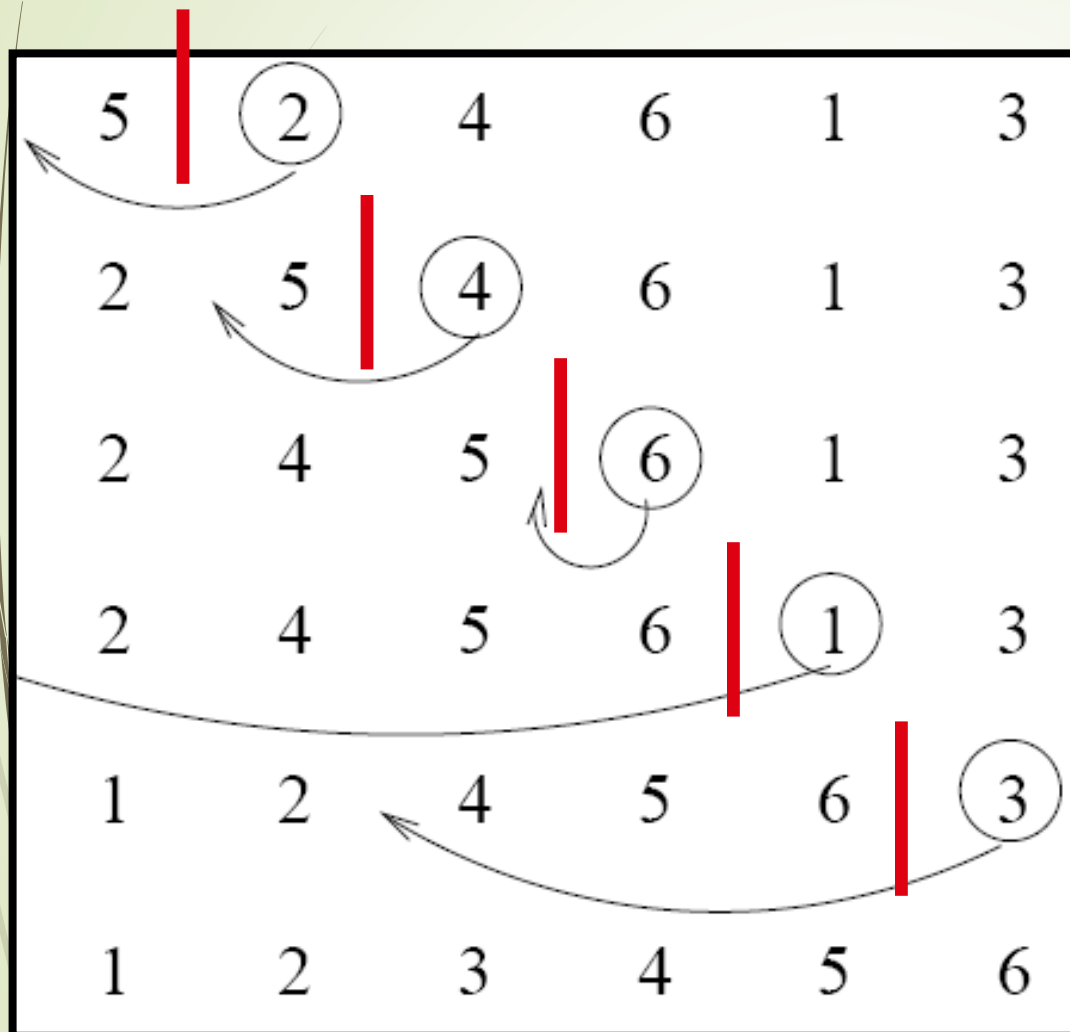
INSERTION SORT

- ❑ Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
 - ❑ Most common sorting technique used by card players.
- ❑ The list is divided into two parts: sorted and unsorted.
- ❑ In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- ❑ A list of n elements will take at most $n-1$ passes to sort the data.

INSERTION SORT

- ❑ We require N Pass to sort N elements
- ❑ During Pass-1, 1st element of the list is scanned which is trivially sorted.
- ❑ During Pass-2, 2nd element of the list is scanned and compared with 1st element in the list. If 2nd element is smaller than 1st element then they are interchanged.
- ❑ During pass-3 ,3rd element of the list is scanned and it is compared with 2nd and 1st element respectively.
- ❑ Same process is repeated upto N elements.

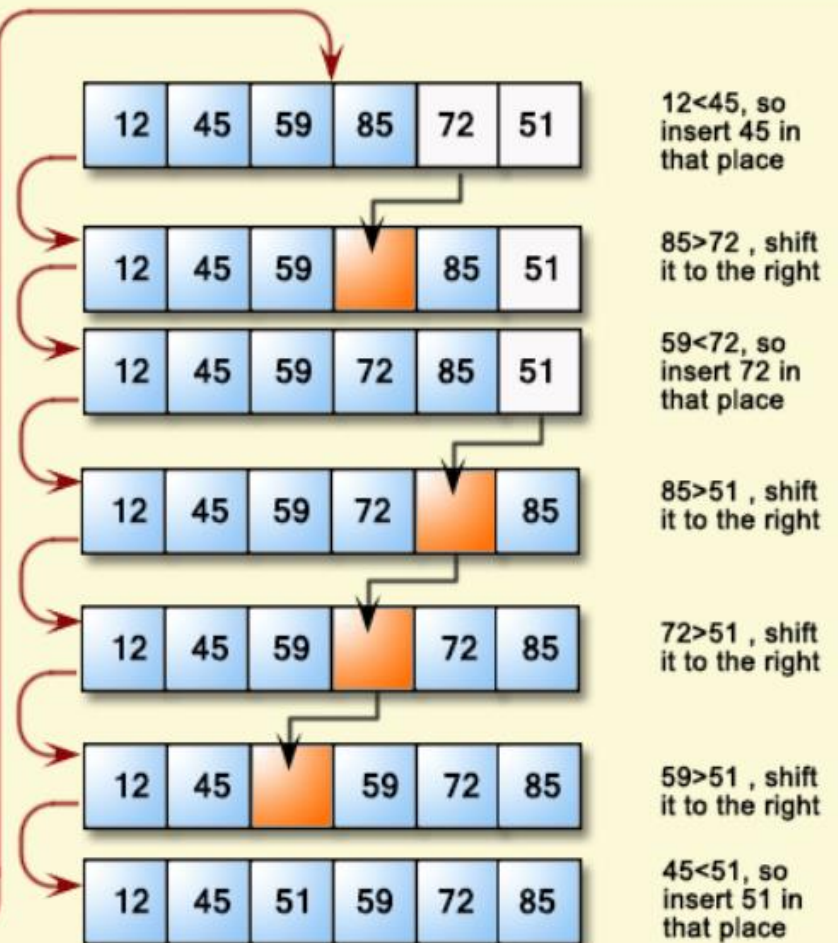
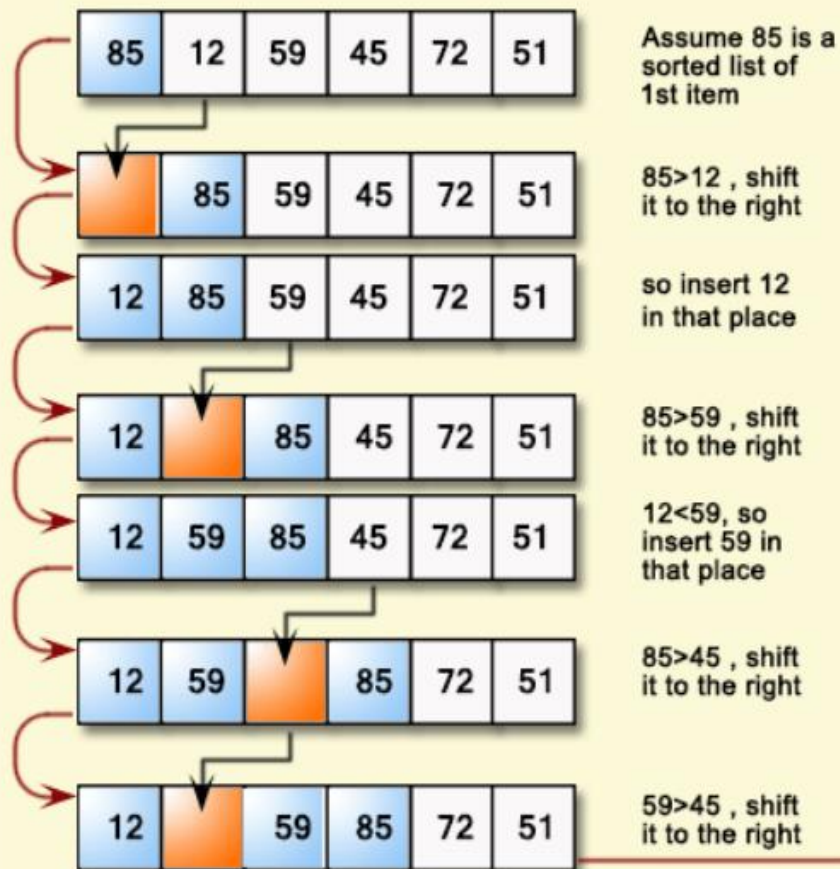
INSERTION SORT EXAMPLE



INSERTION SORT ALGORITHM

```
Step-1 : length <- len(list1)
Step-2 : Repeat upto step 5 for I in range(length)
Step-3 : key <- list1[I]
Step-4 : Repeat Step 5 for J in range(I,0,-1)
Step-5 : if key < list1[J-1] then
         temp <- list1[J]
         list1[J] <- list1[J-1]
         list1[J-1] <- temp
Step-6 : Exit
```


INSERTION SORT EXAMPLE



INSERTION SORT PROGRAM

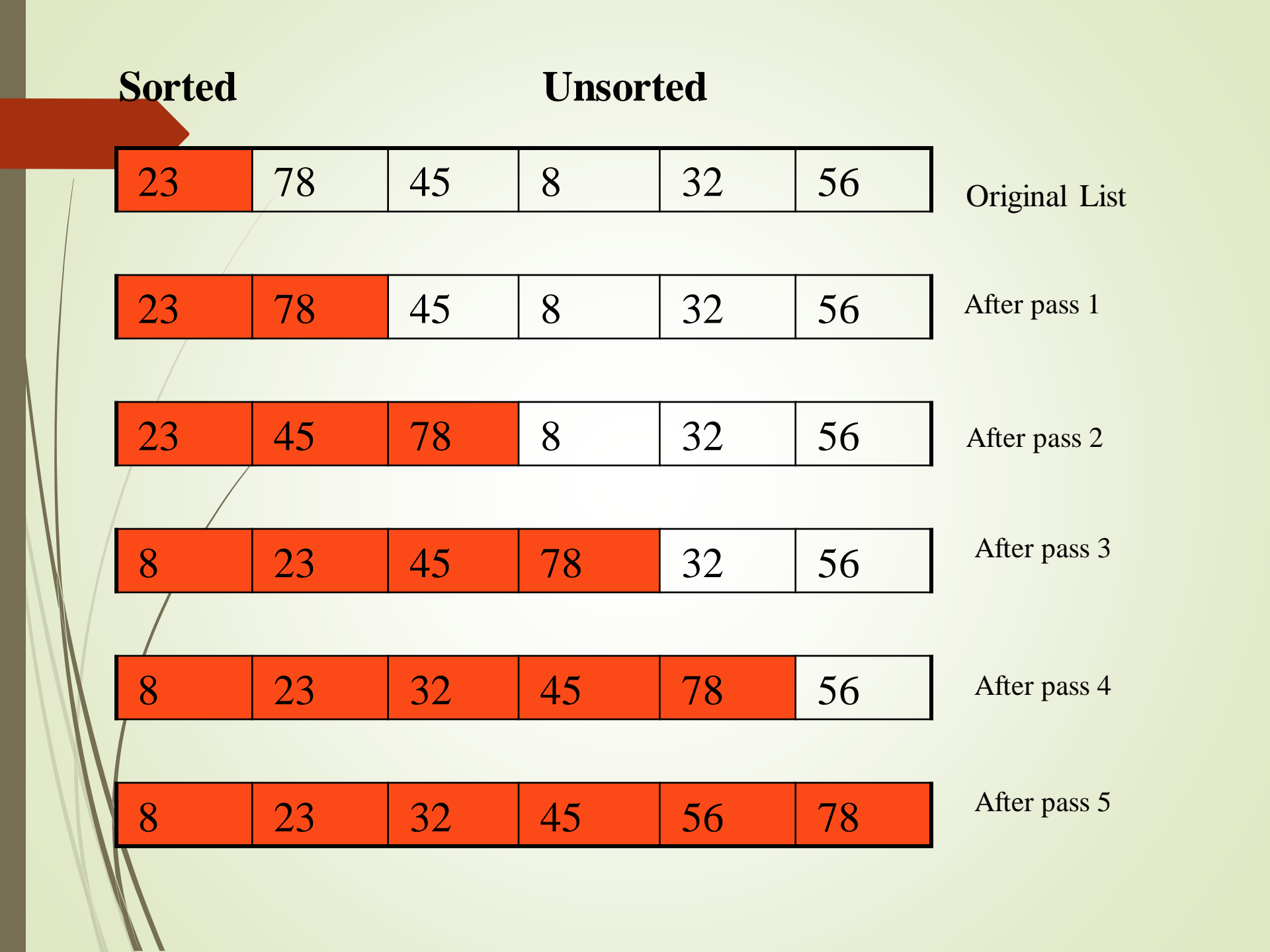
```
# Creating a insertion sort function
def insertion_sort(list1):
    length=len(list1)
    k=1
    # Outer loop for traverse the entire list
    for i in range(length):
        key=list1[i]

        # Compare key with each element on the left of it until an element
        for j in range(i,0,-1):
            if(key < list1[j-1]):
                temp=list1[j]
                list1[j]=list1[j-1]
                list1[j-1]=temp
        print("pass number",k,end='')
        k=k+1
        print(list1)
    return list1

list1 = [5, 3, 8, 6, 7, 2]
print("The unsorted list is: ", list1)
# Calling the bubble sort function
print("The sorted list is: ", insertion_sort(list1))
```

Sorted

Unsorted



23	78	45	8	32	56
----	----	----	---	----	----

Original List

23	78	45	8	32	56
----	----	----	---	----	----

After pass 1

23	45	78	8	32	56
----	----	----	---	----	----

After pass 2

8	23	45	78	32	56
---	----	----	----	----	----

After pass 3

8	23	32	45	78	56
---	----	----	----	----	----

After pass 4

8	23	32	45	56	78
---	----	----	----	----	----

After pass 5

INSERTION SORT – ANALYSIS

- ❑ Running time depends on not only the size of the array but also the contents of the array.
- ❑ **Best-case:** $\rightarrow O(n)$
 - ❑ Array is already sorted in ascending order.
 - ❑ Inner loop will not be executed.
 - ❑ The number of moves: $2*(n-1)$ $\rightarrow O(n)$
 - ❑ The number of key comparisons: $(n-1)$ $\rightarrow O(n)$
- ❑ **Worst-case:** $\rightarrow O(n^2)$
 - ❑ Array is in reverse order:
 - ❑ Inner loop is executed $i-1$ times, for $i = 2, 3, \dots, n$
 - ❑ The number of moves: $2*(n-1) + (1+2+\dots+n-1) = 2*(n-1) + n*(n-1)/2 \rightarrow O(n^2)$
 - ❑ The number of key comparisons: $(1+2+\dots+n-1) = n*(n-1)/2 \rightarrow O(n^2)$
- ❑ **Average-case:** $\rightarrow O(n^2)$
 - ❑ We have to look at all possible initial data organizations.
- ❑ **So, Insertion Sort is $O(n^2)$**

ANALYSIS OF INSERTION SORT

- ❑ Which running time will be used to characterize this algorithm?
 - ❑ Best, worst or average?
- ❑ Worst:
 - ❑ Longest running time (this is the upper limit for the algorithm)
 - ❑ It is guaranteed that the algorithm will not be worse than this.
- ❑ Sometimes we are interested in average case. But there are some problems with the average case.
 - ❑ It is difficult to figure out the average case. i.e. what is average input?
 - ❑ Are we going to assume all possible inputs are equally likely?
 - ❑ In fact for most algorithms average case is same as the worst case.

5 2 35 9 3 -2 52 0

Example of Insertion Sort Method

	5	2	35	9	3	-2	52	0
Pass1	5	2	35	9	3	-2	52	0
Pass2	2	5	35	9	3	-2	52	0
Pass3	2	5	35	9	3	-2	52	0
Pass4	2	5	9	35	3	-2	52	0
Pass5	2	3	5	9	35	-2	52	0
Pass6	-2	2	3	5	9	35	52	0
Pass7	-2	2	3	5	9	35	52	0
Pass8	-2	0	2	3	5	9	35	52

Indicates element which is scanned during each Pass.

Indicates elements within which comparison is performed.

SIMPLE MERGE SORT

- ❑ Merge sort is used to merge two sorted list in a single sorted list.
- ❑ In a simple merge sort we compare the elements of both lists and the element which has the smallest value is placed in a new list.
- ❑ This process is repeated until all the elements from both lists are placed in a new list.
- ❑ First we compare the elements of both the table and the element which has the smallest value placed in new table.

SIMPLE MERGE SORT ALGORITHM

Step-1: $I \leftarrow 0, J \leftarrow 0$

$N1 \leftarrow \text{len}(\text{list1})$

$N2 \leftarrow \text{len}(\text{list2})$

Step-2: Repeat Step 3 while $I \leftarrow (N1-1)$ and $J \leftarrow (N2-1)$

Step-3: if $\text{list1}[I] < \text{list2}[J]$ then

$\text{list3.append}(\text{list1}[I])$

$I \leftarrow I + 1$

else

$\text{list3.append}(\text{list2}[J])$

$J \leftarrow J + 1$

Step-4: Repeat Step 5 while $I \leftarrow (N1-1)$

Step-5: $\text{list3.append}(\text{list1}[I])$

$I \leftarrow I + 1$

Step-6: Repeat Step 7 while $J \leftarrow (N2-1)$

Step-7: $\text{list3.append}(\text{list2}[J])$

$J \leftarrow J + 1$

SIMPLE MERGE SORT ALGORITHM

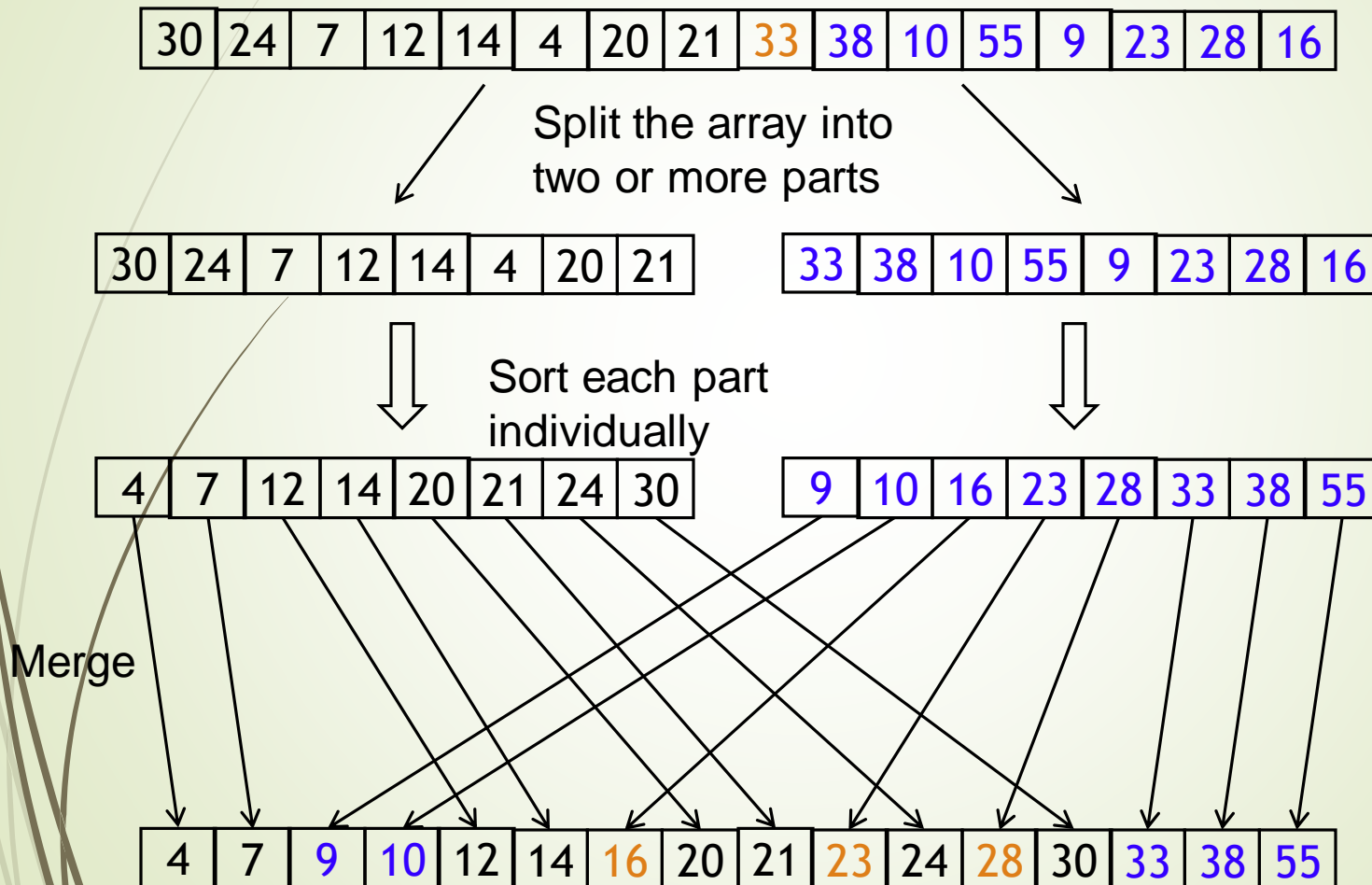
```
def merge_sort(list1, list2, list3):  
    I = 0  
    J = 0  
    N1 = len(list1)  
    N2 = len(list2)  
    while (I <= N1-1 and J <= N2-1):  
        if list1[I] < list2[J]:  
            list3.append(list1[I])  
            I = I + 1  
        else:  
            list3.append(list2[J])  
            J = J + 1  
    while (I <= N1-1):  
        list3.append(list1[I])  
        I = I + 1  
    while (J <= N2-1):  
        list3.append(list2[J])  
        J = J + 1  
    print(list3)  
list1=[28,31,37,39]  
list2=[24,34,43,53,85,89]  
list3=[]  
print("list1 before sorting",list1)  
print("list2 before sorting",list2)  
merge_sort(list1,list2,list3)  
print("after sorting", list3)
```

list1 before sorting [28, 31, 37, 39]
list2 before sorting [24, 34, 43, 53, 85, 89]
[24, 28, 31, 34, 37, 39]
[24, 28, 31, 34, 37, 39, 43, 53, 85, 89]
after sorting [24, 28, 31, 34, 37, 39, 43, 53, 85, 89]

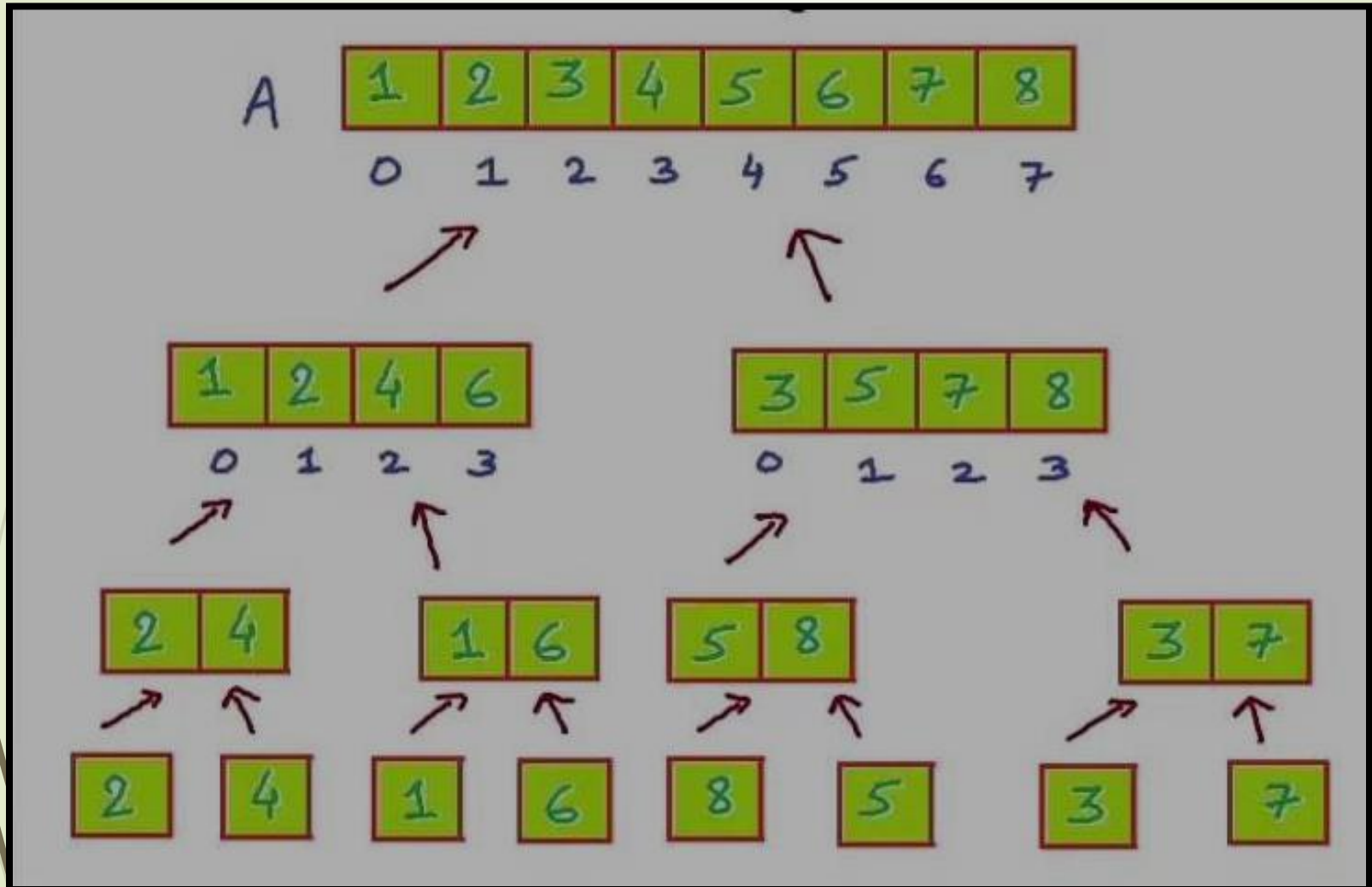
MERGE SORT

- ❑ Merge sort algorithm is one of two important divide-and-conquer sorting algorithms.
- ❑ It is a recursive algorithm.
 - ❑ Divides the list into halves,
 - ❑ Sort each half separately, and
 - ❑ Then merge the sorted halves into one sorted array.

MERGE SORT EXAMPLE



MERGESORT - EXAMPLE



Merge Sort Example

1. Divide the array into two parts

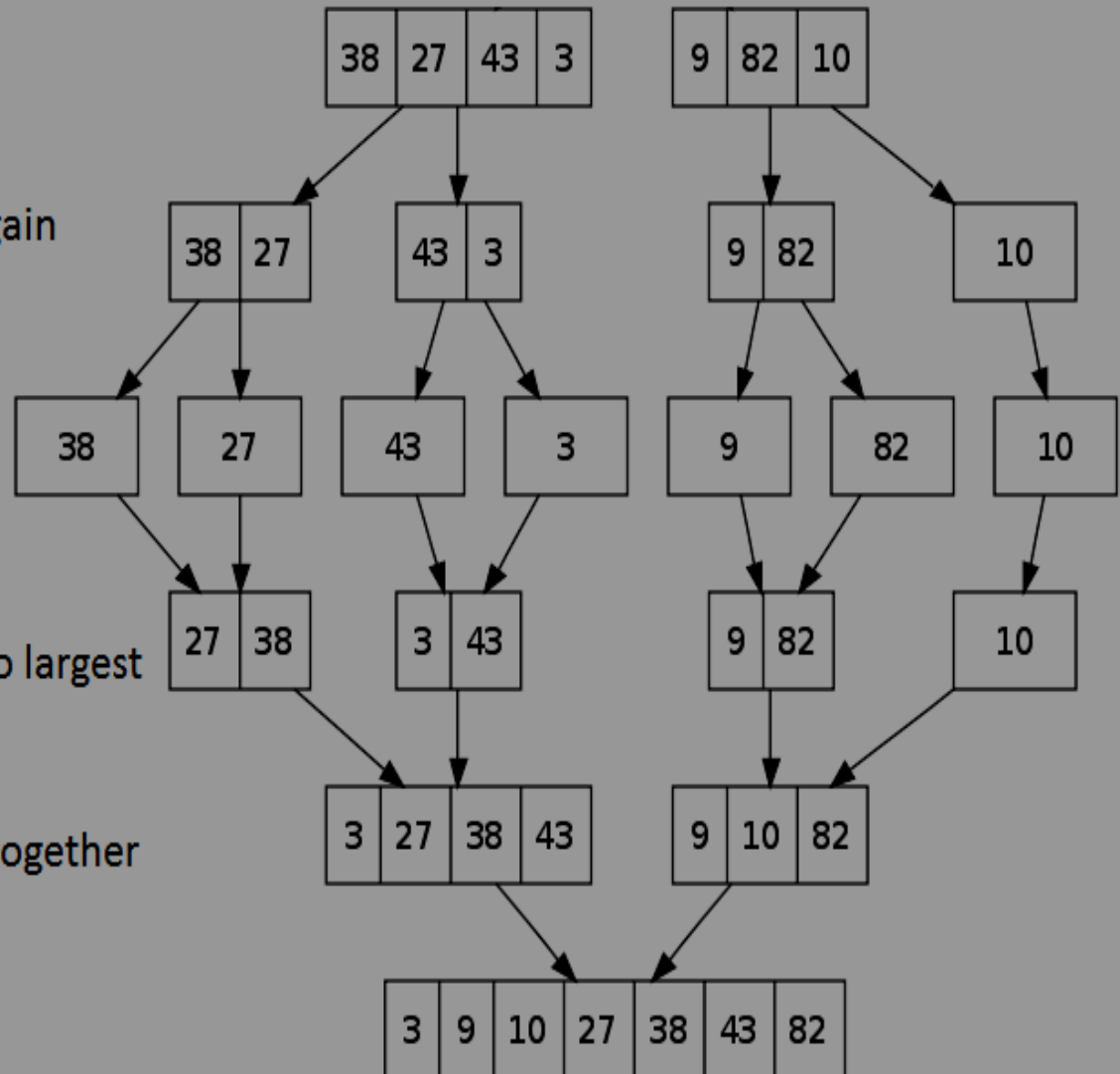
2. Divide the array into two parts again

3. Break each element into single parts

4. Sort the elements from smallest to largest

5. Merge the divided sorted arrays together

6. The array has been sorted



MERGE SORT ALGORITHM

Step-1: `if len(nlist)>1:`

`mid = len(nlist)//2`

`lefthalf = nlist[:mid]`

`righthalf = nlist[mid:]`

Step-2: `mergeSort(lefthalf)`

`mergeSort(righthalf)`

Step-3: `i=0,j=0,k=0;`

Step-4: `while i < len(lefthalf) and j < len(righthalf):`

`if lefthalf[i] < righthalf[j]:`

`nlist[k]=lefthalf[i]`

`i=i+1`

`else:`

`nlist[k]=righthalf[j]`

`j=j+1`

`k=k+1`

MERGE SORT ALGORITHM

Step-5 : while $i < \text{len}(\text{lefthalf})$:

$\text{nlist}[k] = \text{lefthalf}[i]$

$i = i + 1$

$k = k + 1$

Step-6: while $j < \text{len}(\text{righthalf})$:

$\text{nlist}[k] = \text{righthalf}[j]$

$j = j + 1$

$k = k + 1$

Step-7: Exit


```

def mergeSort(nlist):
    print("Splitting ",nlist)
    if len(nlist)>1:
        mid = len(nlist)//2
        lefthalf = nlist[:mid]
        righthalf = nlist[mid:]

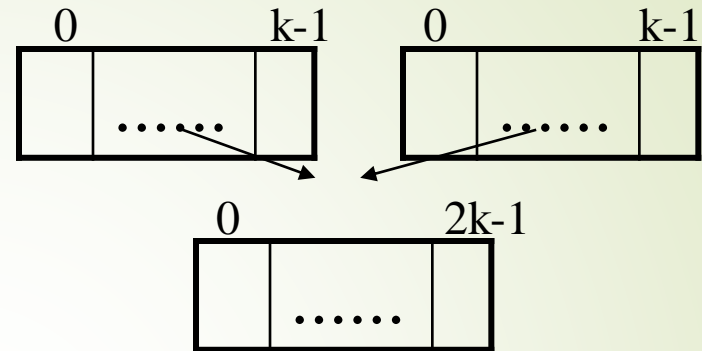
        mergeSort(lefthalf)
        mergeSort(righthalf)
        i=j=k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                nlist[k]=lefthalf[i]
                i=i+1
            else:
                nlist[k]=righthalf[j]
                j=j+1
            k=k+1
        while i < len(lefthalf):
            nlist[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j < len(righthalf):
            nlist[k]=righthalf[j]
            j=j+1
            k=k+1

    print("Merging ",nlist)
nlist = [14,46,43,27,57,41,45,21,70]
mergeSort(nlist)
print(nlist)

```

MERGESORT – ANALYSIS OF MERGE (CONT.)

- ❑ Merging two sorted arrays of size k



- ❑ **Best-case:**

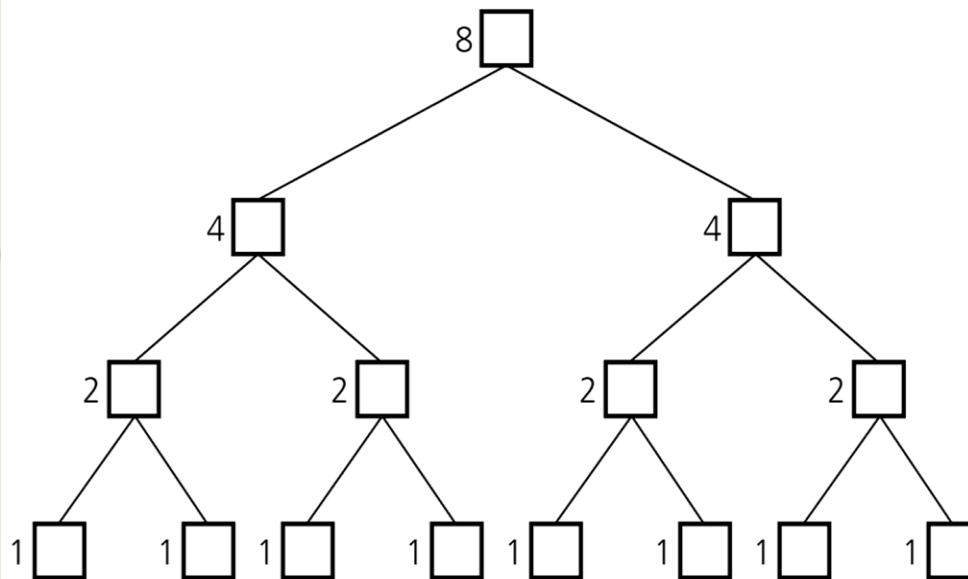
- ❑ All the elements in the first array are smaller (or larger) than all the elements in the second array.
- ❑ The number of moves: $2k + 2k$
- ❑ The number of key comparisons: k

- ❑ **Worst-case:**

- ❑ The number of moves: $2k + 2k$
- ❑ The number of key comparisons: $2k-1$

MERGESORT - ANALYSIS

Levels of recursive calls to *mergesort*, given an array of eight items



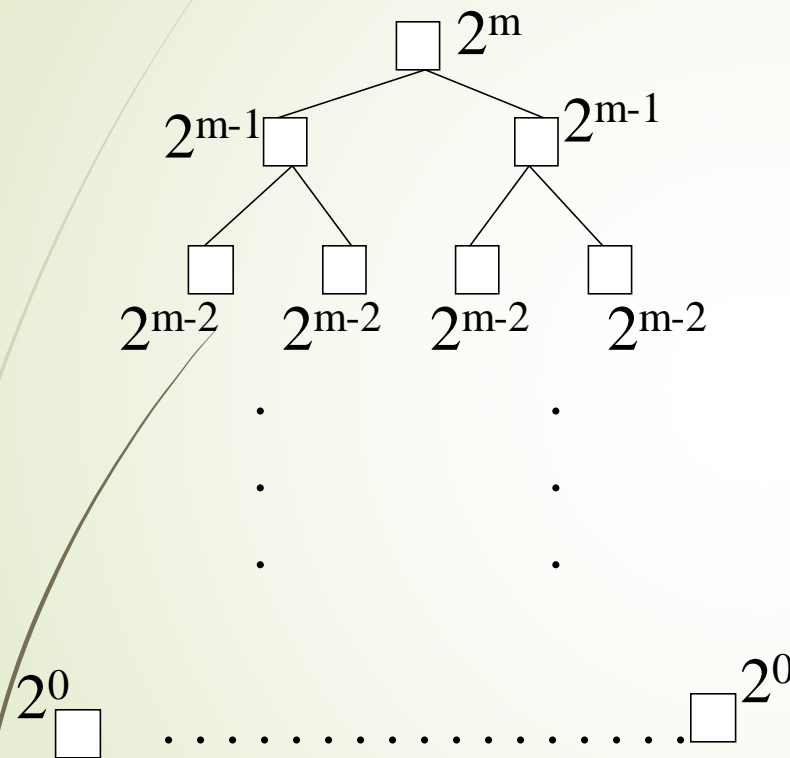
Level 0: mergesort 8 items

Level 1: 2 calls to mergesort with 4 items each

Level 2: 4 calls to mergesort with 2 items each

Level 3: 8 calls to mergesort with 1 item each

MERGESORT - ANALYSIS



level 0 : 1 merge (size 2^{m-1})

level 1 : 2 merges (size 2^{m-2})

level 2 : 4 merges (size 2^{m-3})

level $m-1$: 2^{m-1} merges (size 2^0)

level m

MERGESORT - ANALYSIS

Worst-case –

The number of key comparisons:

$$\begin{aligned} &= 2^0 * (2 * 2^{m-1} - 1) + 2^1 * (2 * 2^{m-2} - 1) + \dots + 2^{m-1} * (2 * 2^0 - 1) \\ &= (2^m - 1) + (2^m - 2) + \dots + (2^m - 2^{m-1}) \quad (m \text{ terms}) \end{aligned}$$

$$= m * 2^m - \sum_{i=0}^{m-1} 2^i$$

$$= m * 2^m - 2^m + 1$$

Using $m = \log n$

$$= n * \log_2 n - n + 1$$

$$\rightarrow O(n * \log_2 n)$$

MERGE SORT – ANALYSIS

- ❑ Mergesort is extremely efficient algorithm with respect to time.
 - ❑ Both worst case and average cases are $O(n * \log_2 n)$
- ❑ But, mergesort requires an extra array whose size equals to the size of the original array.
- ❑ If we use a linked list, we do not need an extra array
 - ❑ But, we need space for the links
 - ❑ And, it will be difficult to divide the list into half ($O(n)$)

COMPARISON OF SORTING ALGORITHMS

	<u>Worst case</u>	<u>Average case</u>
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$
Radix sort	n	n
Treesort	n^2	$n * \log n$
Heapsort	$n * \log n$	$n * \log n$