8 Searching and Sorting Algorithms

Learning Outcome

Fundamental Algorithms
- Implement sort algorithms: insertion sort, bubble sort, quicksort, merge sort
- Use examples to explain sort algorithms
- Implement search algorithms: linear search, binary search, hash table search
- Use examples to explain search algorithms
- Compare and describe the efficiencies of the sort and search algorithms using Big-O notation for time complexity (worst case). Exclude: space complexity

Implementing Algorithms
- Implement sort programs: insertion sort, bubble sort, quicksort, merge sort
- Implement search programs: linear search, binary search, hash table search

In the last chapter, we learnt to search and sort a list using the methods provided in Python. Here, we will look closer into the various algorithms, explaining how the search and sort are done behind the scene and how efficient the algorithms are.

8.1 Searching Algorithms

A searching algorithm intends to find a particular element in a list. This targeted element may not exist in the list, may appear once or multiple times. We call this targeted element the key.

8.1.1 Linear Search

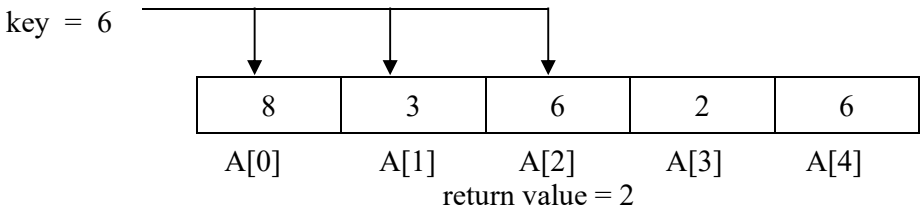
As the name implies, linear search basically searches the items in the list one-by-one.

The function **LinearSearch** requires two parameters: the **list** and the **key**. The algorithms begins at index 0, scans every element in the list until the key is found or the list is exhausted. If the key is found, the function returns the index of the matched item in the list; otherwise, the value −1 is returned. Here are two examples for illustration.

A: 8 3 6 2 6

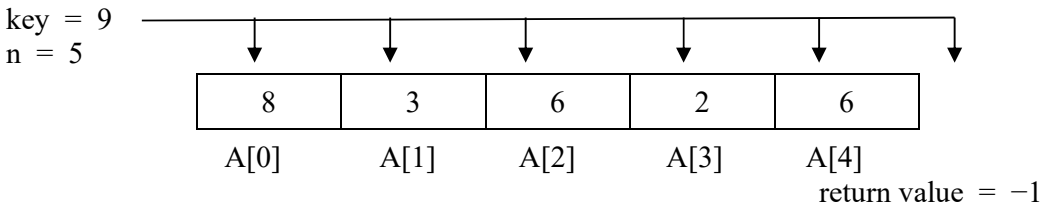
1. **key** = 6

Search the list from the beginning, returning the index of the first occurrence of element 6.



2. **key** = 9 , start = 0, n = 5.

Start at the first element and search the list for the number 9. Since it is not found, return the value −1.



Linear Search Implementation

```
def  LinearSearch (A, key):
#Search the list A for a match with key
#Return the position of the key if found, or -1 otherwise.

     pos = 0                    # start position to search
     found = False

     while  ( not found and pos < len(A) ):
          if A[pos] == key:
               found = True
          else:
               pos = pos + 1

     if  found:                 # return index of matching item
          return  pos
     else:                      # search failed, return -1
          return -1
```

8.1.2 Binary Search

The linear search applies to any list. If the list is **ordered**, an algorithm, called the **binary search**, provides an improved search technique.

Your experience in looking up a number in a large phone directory is a model for the algorithm. Given the name, you move to an earlier or later page in the phone book depending on the relative location of the person's name in the alphabet. For instance, if the person's name begin with "R", and you are in the "T's" you move to an earlier page. The process continues until you get a match or discover that the name is not in the book.

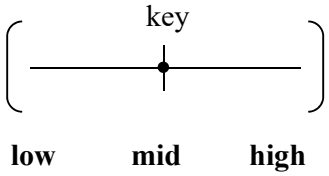
A related idea applies to searching an ordered list. We go to the middle of the list and look for a quick match of our key with the midpoint value. If we fail to find a match, we look at the relative size of the key and the midpoint value and then move to the lower or upper half of the list. In general, if we know how the data are ordered, we can use that information to shorten the search time.

The following steps describe the algorithm. The indices at the ends of the list are **low** = 0 and **high** = n − 1 , where n is the number of elements in the list.

1. Compute the index of the array's midpoint.
   mid = (**low** + **high**) / 2

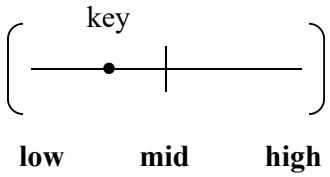
2. Compare the value at this midpoint with the key.

   - If a match occurs, return the index mid to locate the key.



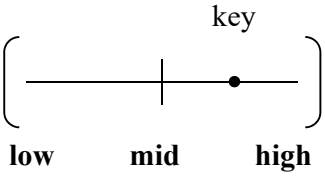
   If (A[**mid**] = = key)
   return **mid**

   - If key < A[**mid**], the key must lie within the lower range (left half) since the list is ordered. The new boundaries are **low** and **high = mid − 1.**



Search Lower Range, index from **low** to **mid − 1**

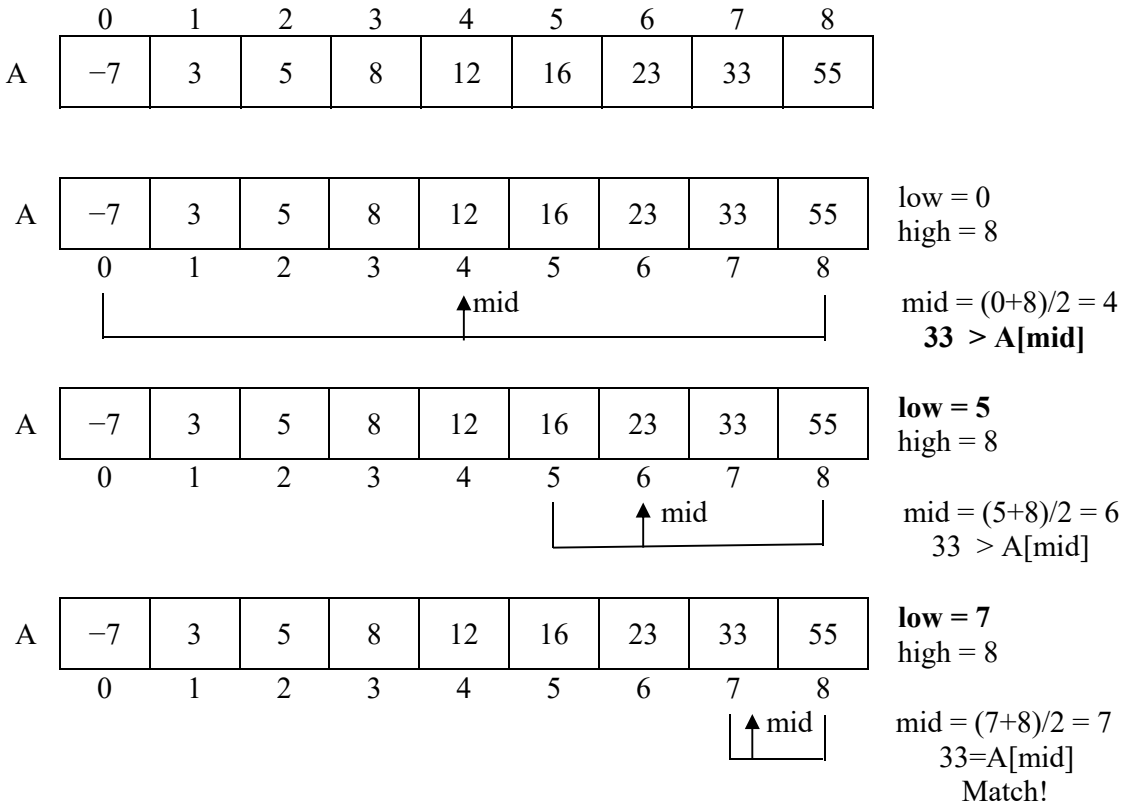
   - If key > A[**mid**], the key must lie within the upper range (right half) since the list is ordered. The new boundaries are **low = mid + 1** and **high**.



Search Upper Range, index from **mid + 1** to **high**

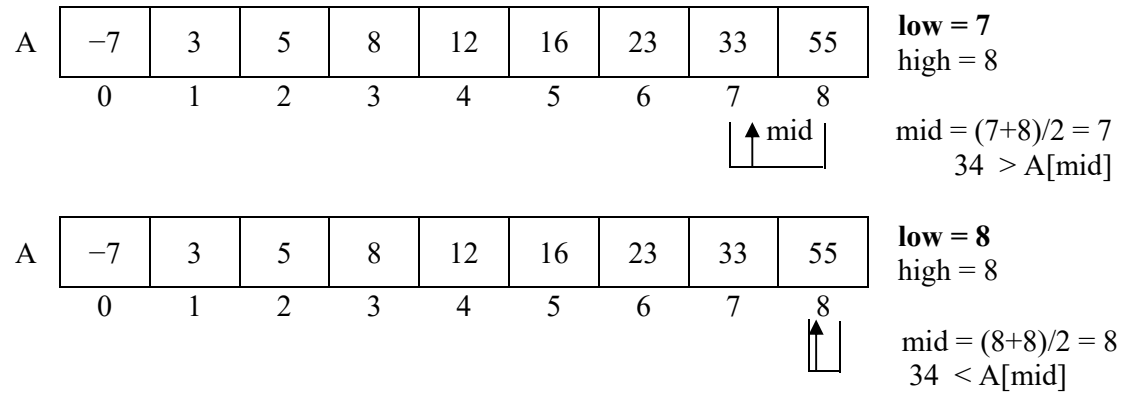
3. The algorithm refines the location of a match by halving the length of the interval in which key can exist and then executing the same search algorithm on the smaller sublist.
4. Eventually, if the key is not in the list, **low** will exceed **high** and the algorithm returns the failure indicator of −1 (match not found).

Let's illustrate with an array A and search for the key = 33.



Note that binary search requires 3 comparisons while linear search would take 8 comparisons.

Another example for the same array but search for key = 34. The first two comparison will be exactly the same with the above example. Let's look at the last comparison



Now we have key < A[mid] and this leads to high = mid − 1 = 7. Here low exceeds high and the algorithm will return the failure indicator of −1 (match not found).

Note that binary search requires 4 comparisons while linear search would take 9 comparisons.

Binary Search Implementation

```
def  BinarySearch (A, key):
#Search the ordered list A for a match with key
#Return the position of the key if found, or -1 otherwise.

     found = False       # initialization
     low = 0
     high = len(A) - 1

     while (not found) and (low <=high):
          mid = (low + high) // 2   # mid index of the sublist

          if  key == A[mid]:        # have a match
               found = True
          if  key < A[mid]:         # go to lower sublist
               high = mid-1
          if  key > A[mid]:         # go to upper sublist
               low = mid+1

     if found:           # return index of matching item
          return mid
     else:               # i.e.  low > high
          return - 1     # search failed, return -1
```

8.1.3 Hash Table Search

In an array or list, items are inserted subsequently into the data structure, starting from the first index. Linear search and binary search compares the targeted item by a sequence of trial-and-error comparisons.

Here we introduce another data structure, hash table, where the location of each item is determined by a hash function of the item itself. This makes hash table search at the designated location of the item and thus less comparisons.

Hash Function

As an illustration, suppose that up to 25 integers in the range 0 through 999 are to be stored in a hash table, This hash table can be implemented as an integer array *table* in which each array element is initialized with some dummy value, such as −1. If we use each integer i in the set as an index, that is, we store i in *table*[i].

The function h defined by h(i) = i that determines the location of an item i in the hash table is called a **hash function**.

To determine whether a particular integer *number* has been stored, we need only check if *table*[*number*] is equal to *number*. only one location needs to be examined. This method is thus very time efficient, but it is surely not space-efficient. Only 25 of the 1000 available locations are used to store items, leaving 975 unused locations; only 2.5 percent of the available space is used, and so 97.5 percent is wasted!

Because it is possible to store 25 values in 25 locations, we might try improving space utilization by using an array *table* with capacity 25. Obviously, the original hash function h(i) = i can no longer be used. Instead, we might use

h(i) = i modulo 25

or in Python,

```
def h(i):
    return (i % 25)      # division by 25 method
```

because this function always produces an integer in the range 0 through 24. The integer 52 thus is stored in *table*[2], since h(52) = 52 % 25 = 2. Similarly, 129, 500, 273, and 49 are stored in locations 4, 0, 23, and 24 respectively.

	Hash Table
table[0]	500
table[1]	−1
table[2]	52
table[3]	−1
table[4]	129
table[5]	−1
.	.
.	.
.	.
.	.
.	.
.	.
table[23]	273
table[24]	49

Collision Strategies

There is an obvious problem with the preceding hash table, namely, that **collisions** may occur. For example, if 77 is to be stored, it should be placed at location h(77) = 77 % 25 = 2, but this location is already occupied by 52. In the same way, many other values may collide at a given position, for example, 2, 27 and 102; and, in fact, all integers of the form 25k + 2 hash to location 2. Obviously, some strategy is needed to resolve such collisions.

Linear Probing (or Linear Probe Open Addressing)
One simple strategy for handling collisions is known as **linear probing.** In this method, a linear search of the table begins at the location where a collision occurs and continues until an empty slot is found in which the item can be stored.

Thus, in the preceding example, when 77 collides with the value 52 at location 2, we simply put 77 in position 3; to insert 102, we follow the **probe sequence** consisting of locations 2, 3, 4, and 5 to find the first available location and thus store 102 in *table*[5].

If the search reaches the bottom of the table, we continue at the first location. For example, 123 is stored in location 1, since it collides with 273 at location 23, and the probe sequence 23, 24, 0, 1 locates the first empty slot at position 1.

	Hash Table
table[0]	500
table[1]	123
table[2]	52
table[3]	77
table[4]	129
table[5]	102
.	.
.	.
.	.
.	.
.	.
.	.
table[23]	273
table[24]	49

Hash Table Search with Linear Probing

To determine if a specified value is in this hash table, we first apply the hash function to compute the position at which this value should be found. There are three cases to consider.

First, if this location is empty, we can conclude immediately that the value is not in the table.

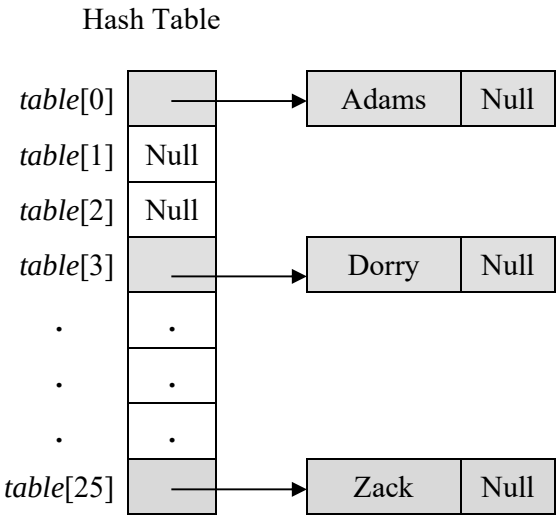
Second, if this location contains the specified value, the search is immediately successful.

In the third case, this location contains a value other than the one for which we are searching, because of the way that collisions were resolved in constructing the table. In this case, we begin a “circular” linear search at this location and continue until either the item is found or we reach an empty location or the starting location, indicating that the item is not in the table.

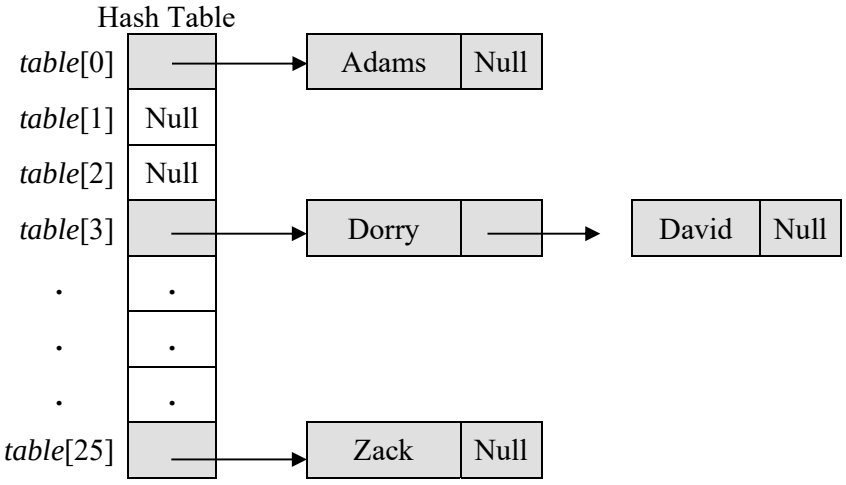
In the linear probe scheme, whenever collisions occur, the colliding values are stored in locations that should be reserved for items that hash directly to these locations. This approach makes subsequent collisions more likely, thus compounding the problem.

Chaining (or Chaining with Separate Lists)
A better approach, known as **chaining,** uses a hash table that is an array of linked lists that store the items. To illustrate, suppose we wish to store a collection of names, we might use an array *table* of 26 linked lists, initially empty, and the simple hash function h(name) = ord(name[0]) – ord('A') ; that is, h(name) is 0 if name[0] is 'A', 1 if name[0] is 'B', ... , 25 if name[0] is 'Z'.

Thus, for example, "Adams" and "Dorry" are stored in nodes pointed to by *table*[0] and *table*[3], respectively.



When a collision occurs, we simply insert the new item into the appropriate linked list. For example, since h("David") = h("Dorry") = ord('D') – ord('A') = 3, a collision occurs when we attempt to store the name "David", and thus we add a new node containing this name to the linked list pointed to by *table*[3] :



Hash Table Search with Chaining
Searching such a hash table is straightforward. We simply apply the hash function to the item being sought and then use one of the search algorithms for linked list.

Chaining method is generally faster than Linear Probing method because only items that hash to the same table location are searched. Furthermore, linear probe addressing assumes a fixed-length table, whereas in chaining with separate lists, entries in the hash table are dynamically allocated. The list size is limited only by the amount of memory. The primary disadvantage of chaining is the space required to allocate the additional node pointer field. In general, the dynamic structure of separate chaining makes it the preferred choice for hashing.

A factor on the design of a hash table is the selection of the hash function. The behavior of the hash function obviously affects the frequency of collisions. For example, the preceding hash function in the example is not a good choice because some letters occur much more frequently than others as first letters of names. Thus the linked list of names beginning with 'T' tends to be much longer than containing names that begin with 'Z'. This clustering effect results in longer search times for T-names than for Z-names.

A better hash function that distributes the names more uniformly throughout the hash table might be the "average" of the first and last letters in the name:

$$h(name) = (ord(first\ letter) + ord(last\ letter)) / 2$$

or one might use the "average" of all the letters. The hash function must not, however, be so complex that the time required to evaluate it makes the search time unacceptable.

Tutorial 8A

1. (a) N95P2Q10
The following list of twenty-one integers is stored in ascending order in an array:

8, 12, 17, 18, 24, 27, 28, 35, 38, 39, 49, 63, 64, 68, 70, 71, 77, 84, 88, 89, 91

If the list is searched by means of a binary search, state which elements would be accessed, and in what order,

(i) When searching for the number 88 (which is present), and
(ii) when searching for 65 (which is not present) ? [2]

(b) N02P1Q6
Describe the difference between a binary search and a linear search. [2]

2. SP02P1Q2
The following pseudo-code algorithm describes one method of finding an arbitrary name in an alphabetically ordered array of N unique names.

set ***first*** to 1
set ***last*** to N

repeat
set ***mid*** to the integer part of ***(first + last)/2***
if the ***midth*** name precedes the wanted name **then**
set ***first*** to ***mid + 1***
else
set ***last*** to ***mid - 1***
endif
until ***first*** > ***last*** or ***midth*** name is the wanted name

(a) If 142 names are stored in the array, and HAMMOND is the 44th name, state the elements of the array that are examined when searching for HAMMON. [4]

(b) If a search is made for a name that is not in the array, what is the largest number of elements that might need to be examined before one could say that the name is not present? Explain how you arrive at your answer. [3]

3. N09P1Q3
Below is a recursive algorithm for finding a value, SearchItem, in an ordered array, X.

Search(Low, High)
Mid =(Low+High) div 2
If X(Mid) = SearchItem then output "Found" : exit
If X(Mid) > SearchItem then **Search**(Low, Mid-1)
Else **Search**(Mid+1, High)
End Search

Note: the div operation returns an integer value after division e.g. 7 div 2 = 3

Using the above algorithm:
(a) Explain what is meant by a recursive algorithm. [1]
(b) Describe what might occur during execution with an incorrectly written recursive routine. [3]

Array X has 15 elements and the subscript start at 1.

(c) If the algorithm was used to search the array X for the value stored at X(3), state the calls to Search as the recursion executes. [4]
(d) The algorithm does not handle the case where SearchItem is not present in X. Indicate what changes need to be made to Search to rectify this problem. [3]
(e) For this method of searching, state the maximum number of comparisons for array X, justify your answers. [5]

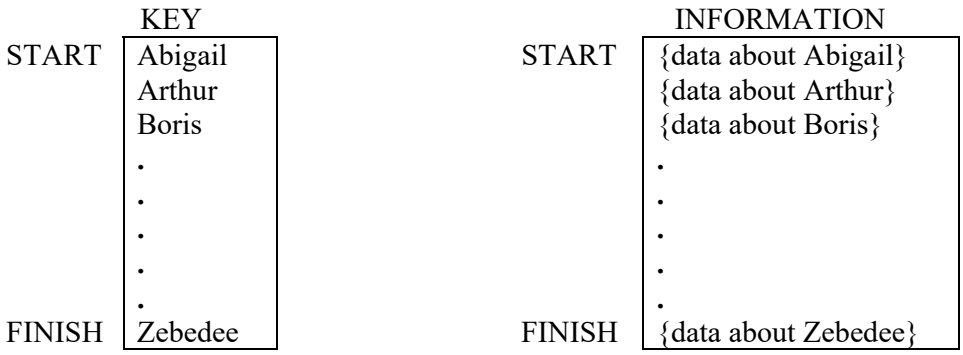
4. Using a hash table with eleven locations and the hashing function h(i) = i % 11, show the hash table that results when the following integers are inserted in the order given: 26, 42, 5, 44, 92, 59, 40, 36, 12, 60, 80 .Assume that collisions are resolved using

(a) linear probing.

(b) chaining.

Assignment 8A

1. J94P1Q9
The diagram shows how a collection of records is stored in two arrays. The keys are in ascending order.



The following two sections of pseudocode represent alternative procedures for a search to find the position of a particular record.

```
procedure SearchOne (First, Last)
        if  First > Last  then {record is not present}
        else if WantedKey = Key[First] then {record found at position First}
        else SearchOne (First+1, Last)

procedure SearchTwo (First, Last)
        if  First > Last  then {record is not present}
        else Middle = (First + Last) div 2
            if WantedKey = Key[Middle] then {record found at position Middle}
            else if WantedKey > Key[Middle] then SearchTwo(Middle+1, Last)
            else SearchTwo (First, Middle-1)
```

(a) The eleven keys held in the array on one occasion are Anne, Bryn, Cleo, Dora, Eric, Fran, Grag, Hugh, Iisa, John and Kate. Illustrate the operation of the two procedures by showing the path followed when WantedKey is Hugh and each procedure is called initially with the parameter values 1, 11. (You should show each procedure call made, with its parameter values, and each key value tested.) [8]

(b) Each procedure works irrespective of the number of keys. Discuss how you would decide which procedure to use for a particular application. [2]

2. We want to use linear search to count the number of occurrence of a key in a list. When the key does not exist in the list, it is 0 occurrence. Write a recursive function `RecSearch(A, key, start)` which requires three parameters, the list A, the key to be found, and the start index to search in the list. This function returns the number of occurrence of this key in the list.

3. Write a recursive function `binSearch ( A, low, high, key)` which searches a sorted list **A, A[low] . . . . . A[high]** , for a match with **key** using binary search**.** Return the index of the matching item or −1 if the key is not found.

8.2 Sorting Algorithms

The ordering of items in a list is important for many applications. For instance, an inventory list may sort records by their part numbers, a dictionary maintains words in alphabetical order to allow quick access to a word. In this section we consider the problem of sorting a list ,

$$X_1,\ X_2,\ X_3,\ \ldots\ldots,\ X_n$$

that is, arranging the list elements so that they are in ascending order

$$X_1 <= X_2 <= X_3 <= \ldots\ldots <= X_n$$

or in descending order

$$X_1 >= X_2 >= X_3 >= \ldots\ldots >= X_n$$

The sort functions that we develop here operate on a list of integers and uses a **swap** function to exchange the positions of two items in the list.

```
def  swap(A, i, j):
# exchange the items at position i and j
     temp = A[i]
     A[i] = A[j]
     A[j] = temp
```

We now introduce classical sorting algorithms that cover the main techniques of in-place sorting in ascending order. Although the algorithms are not efficient for practical use for a large number of data items, they illustrate the main approaches for sorting of an array.

8.2.1 Bubble Sort

For an array A with n elements, the bubble sort requires **up** to (n−1) passes. For each pass, we compare adjacent elements and exchange their values when the first element is greater than the second element. At the end of the each pass, the largest element has "bubbled up" to the end of the current sublist. For instance, after pass 0 (1st pass) is complete, the tail of the list (A[n−1]) is sorted and the front of the list remains unordered.

Let's look at the details of the passes. In the process, we maintain a record of the last index that is involved in an exchange. The variable lastExchagneIndex is used for this purpose and is set to 0 at the start of each pass.

Pass 0 compares adjacent elements (A[0], A[1]), (A[1], A[2]), ... , (A[n−2], A[n−1]). For each pair (A[j], A[j+1]), exchange the values if A[j] > A[j+1] and update lastExchangeIndex to j. At the end of the pass, the largest element is in A[n−1] and the value lastExchangeIndex indicates that all elements in the tail of the list from A[lastExchangeIndex+1] to A[n−1] are in sorted order.

For subsequent passes, we compare adjacent elements in the sublist A[0] to A[lastExchagneIndex]. The process terminates when lastExchangeIndex = 0. The algorithm makes a **maximum** of (n−1) passes.

We illustrate the bubble sort algorithm with the five-element array A = 50, 20, 40, 75, 35.

	A[0]	A[1]	A[2]	A[3]	A[4]	
Pass 0 (the 1st pass):	50,	20,	40,	75,	35	Exchange 50 and 20
	20,	50,	40,	75,	35	Exchange 50 and 40
	20,	40,	50,	75,	35	50 and 75 are ordered
	20,	40,	50,	75,	35	Exchange 75 and 35
	20,	40,	50,	35,	**75**	75 is the largest element lastExchagneIndex = 3

Since lastExchangeIndex is not 0, the process continues.
In pass 1, we scan the sublist of elements A[0] to A[lastExchangeIndex] = A[3].

	A[0]	A[1]	A[2]	A[3]	A[4]	
Pass 1 (the 2nd pass):	20,	40,	50,	35,	**75**	20 and 40 are ordered
	20,	40,	50,	35,	**75**	40 and 50 are ordered
	20,	40,	50,	35,	**75**	Exchange 50 and 35
	20,	40,	35,	**50**,	**75**	50 is the largest element lastExchangeIndex = 2

The new value of lastExchagneIndex becomes 2, and the process continues.
In pass 2, we scan the sublist A[0] to A[lastExchangeIndex] = A[2].

	A[0]	A[1]	A[2]	A[3]	A[4]	
Pass 2 (the 3rd pass):	20,	40,	35,	**50**,	**75**	20 and 40 are ordered
	20,	40,	35,	**50**,	**75**	Exchange 40 and 35
	20,	35,	**40**,	**50**,	**75**	40 is the largest element lastExchangeIndex = 1

The resulting value of lastExchangeIndex is 1.
In pass 3, we scan the sublist A[0] to A[lastExchangeIndex] = A[1].

The single comparison of 20 and 35 leads to no exchanges. lastExchangeIndex is 0, and the process terminates.

	A[0]	A[1]	A[2]	A[3]	A[4]	
Pass 3 (the 4th pass):	20,	35,	**40**,	**50**,	**75**	20 and 35 are ordered
	20,	**35**,	**40**,	**50**,	**75**	Ordered List lastExchangeIndex = 0

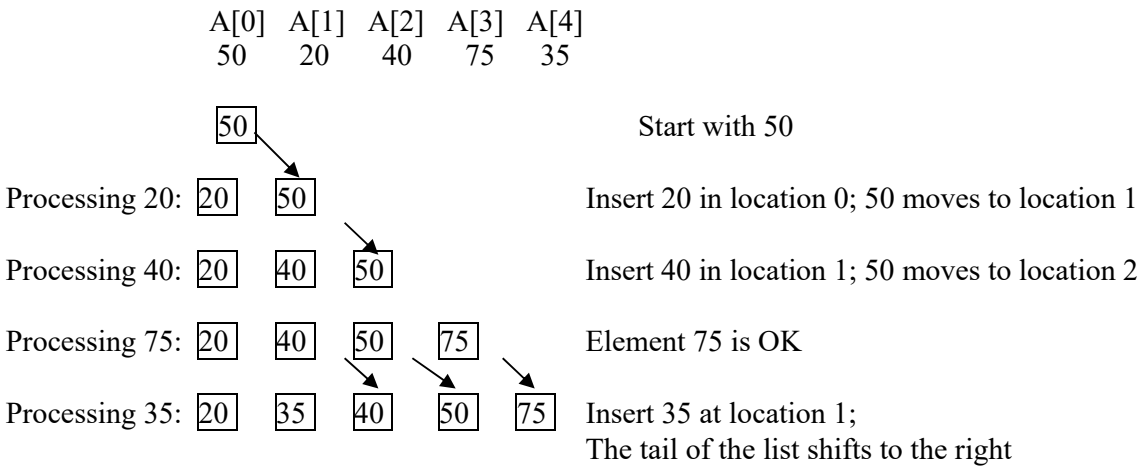
Bubble Sort Implementation:

```
def  bubbleSort (A):
# sort a list, A, using bubble sort algorithm

     i = len(A) - 1 # index of last element in the sublist

     while  i > 0: # continue until no exchanges are made

          lastExchangeIndex = 0

          # scan the sublist A[0] to A[i]
          for  j  in  range(i + 1):

               # exchange a pair and update lastExchangeIndex
                    if  A[j] > A[j+1]:
                    swap (A, j, j+1)
                    lastExchangeIndex = j

          # set i to index of the last exchange
          # continue sorting the sublist A[0] to A[i]
          i = lastExchangeIndex
```

8.2.1 Insertion Sort

The insertion sort is similar to a familiar paper-shuffling process that orders a list of names. The registrar puts each person's name on a card and then rearranges the cards in alphabetical order by sliding a card forward in the stack until it finds the correct location. As the process goes on, the cards at the front of the stack are sorted and those at the rear of the stack are waiting to be processed.

We describe the process with the list of five integer values: A = 50, 20, 40, 75, 35.



Insertion Sort Implementation:

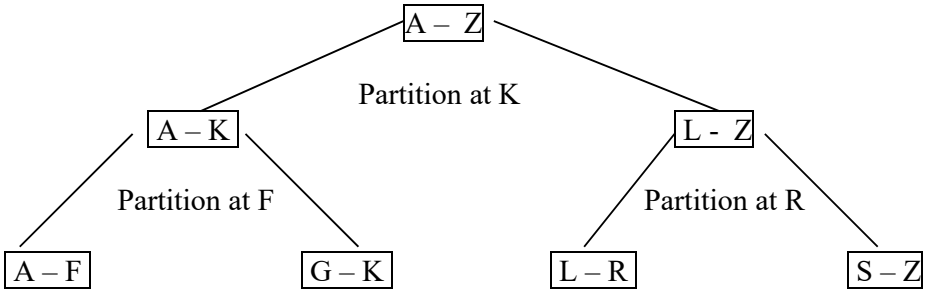
The function insertionSort is passed a list A and the size of the list n.

Let's look at pass i (1 <= i <= n−1) . The sublist A[0] to A[i−1] is already sorted in ascending order. The pass assigns A[i] to the list. Let A[i] be the target and move down the list, comparing the target with items A[i−1], A[i−2], and so forth. Stop the scan at the first element A[j] that is less than or equal to target or at the beginning of the list (j = 0). As we move down the list, slide each element to the right (A[j] = A[j−1]). When we have found the correct location for A[i], insert it at location j.

```
def  insertionSort(A, n):
# sort an list, A, of n integer elements using insertion sort

     for  i  in range(1, n):
          # i  identifies the sublist A[0] to A[i]

          # index j scans down list from A[i-1]
          # looking for correct position to locate target
          target = A[i]
          j = i

          # locate insertion point by scanning downward
          # as long as target < A[j-1] and
          # we have not encountered the beginning of the list
          while  j > 0  and  target < A[j-1]:

               # shift elements up to make room for insertion
               A[j] = A[j-1]
               j = j - 1

          # the location is found; insert the target
          A[j] = target
```

8.2.3 Quick sort

It is the fastest known sorting algorithm. Like most sorting algorithm, the Quick sort technique is derived from familiar experiences. To sort a large stack of papers by name, we can split the papers into two piles with some pivot character such as K separating the list. All names less than or equal to K go in one pile and the rest go into the second pile. We then take each pile and split it into two parts. For instance, in the following figure, the partition points are 'F' and 'R'. We continue to subdivide the piles into smaller and smaller stacks.



The Quick sort algorithm uses the partition approach to sort a list. The algorithm determines a pivot value to split the list into two parts. It separates the elements into parts within the list. We introduce the algorithm with an example and then add the details.

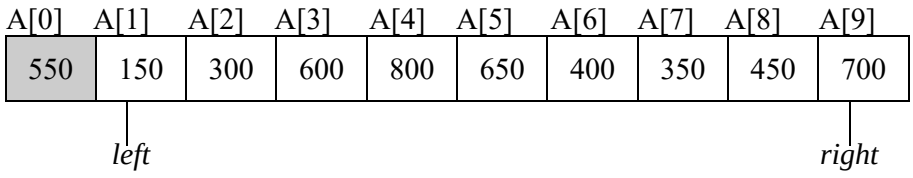
Assume that list A contains 10 integer values:

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
A =	800,	150,	300,	600,	550,	650,	400,	350,	450,	700

Scanning Phase
We scan the entire range of elements in the list A[0] to A[9]. The range extends from *low* = 0 to *high* = 9 with a middle index at *mid* = 4. The first pivot value is A[*mid*] = 550 and the algorithm separates the elements of A into two sublists S_{low} and S_{high}.

Sublist S_{low} is the lower sublist and will contain the elements that are less than or equal to the pivot. The higher sublist S_{high} will contain the elements that are greater than the pivot.

Since we know that the pivot will ultimately end up in S_{low}, we temporarily move it to the low end of the range and exchange its value with A[0] (A[*low*]). This allows us to scan the sublist A[1] to A[9] using two indices *left* and *right*. The variable *left* is initially set at index 1 (*low* + 1) and is responsible for locating elements for sublist S_{low}. Variable *right* is set at index 9 (*high*) and locates elements for sublist S_{high}. The goal of the pass is to identify the elements in each of the sublists.

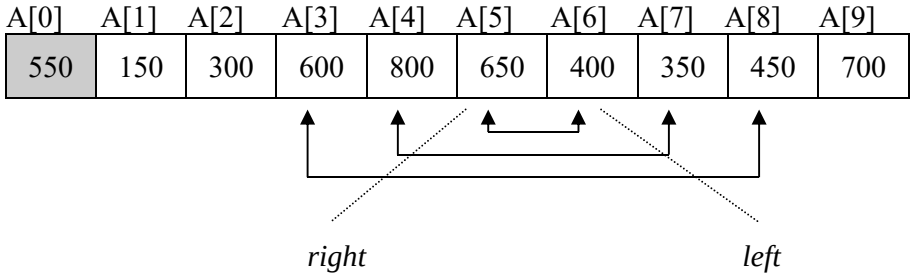


The creativity of Quick sort derives from the interaction between the two indices as they scan the list. *left* moves up the list, whereas the index *right* moves down the list. We move *left* forward looking for an element A[*left*] that is greater than the pivot. At that point the scan stops and we prepare to relocate the element to the upper sublist.

Before the relocation can occur, we move the index *right* downward in the list and wait for it to identify an element that is less than or equal to the pivot. We then have two elements that are in the wrong sublists and can exchange them.

swap (A, *left*, *right*) # swap misplaced partners

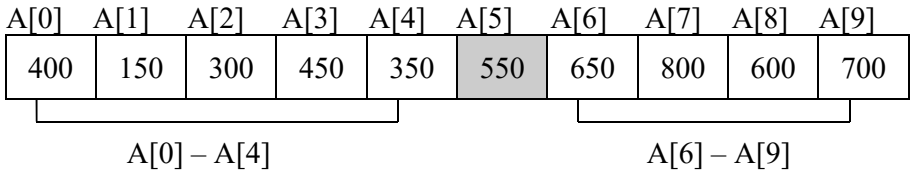
The process continues until *left* and *right* pass each other with *right* = 5, *left* = 6. At this point, *right* has first entered into the lower list, which contains the elements less than or equal to pivot. We hit the separation point between the two lists and have identified the final location for pivot. In the example, swap 600 and 450, 800 and 350, 650 and 400.



We then exchange the pivot A[0] with A[*right*].

swap (A, 0, *right*)

The result creates sublist A[0] – A[4] whose elements are less than those in sublist A[6] – A[9]. The pivot (550) at A[5] creates two sublists that are approximately one half the size of the original list. These two sublists are processed using the same algorithm in which we call the recursive phase.

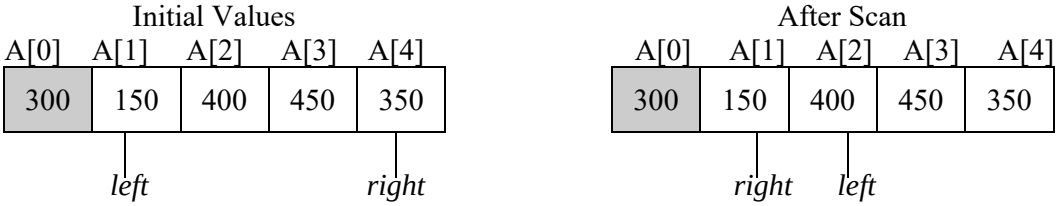


Recursive Phase Process the two sublist S_{low} (A[0] – A[4]) and S_{high} (A[6] – A[9]) using the same methods.

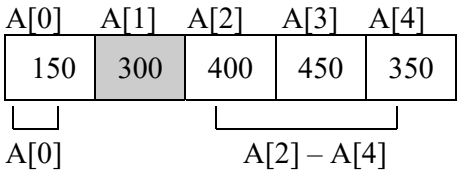
Sublist S_{low} :
The range of the sublist is 0 to 4 with *low* = 0 and *high* = 4, *mid* = 2, and pivot is A[*mid*] = 300. Exchange pivot and A[*low*] and assign initial value to *left* and *right*:

left = 1 = *low*+1
right = 4 = *high*

left stops at index 2 (A[2] > pivot)
right stops at index 1 (A[1] < pivot)



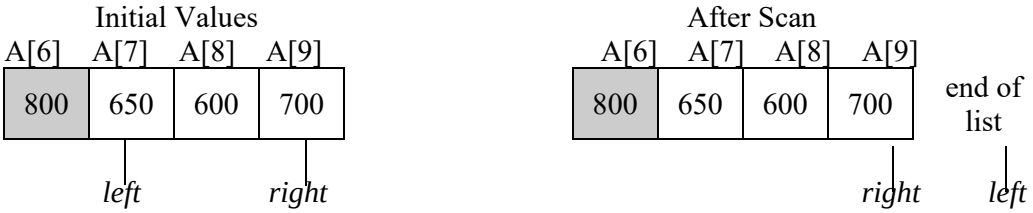
Since *right* < *left*, the process halts and *right* is the separate point between two smaller sublists A[0] and A[2] – A[4]. Complete the process by exchanging A[*right*] = 150 and A[*low*] = 300. Note that the location of pivot leaves us with a one-element sublist and a three-element sublist. The recursive process terminates on an empty or single-element sublist.



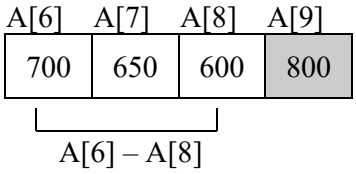
Sublist S_{high} :
The range of the sublist is 6 to 9 with *low* = 6 and *high* = 9, *mid* = 7, and pivot is A[*mid*] = 800. Exchange pivot and A[*low*] and assign initial value to *left* and *right*:

left = 7 = *low*+1
right = 9 = *high*

left stops when it passes the end of the list
right remains at its initial position



Since *right* < *left*, the process halts and *right* locates the insertion point for pivot. Complete the process by exchanging A[*right*] = 700 and A[*low*] = 800. Note that the location of pivot leaves us with a three-element sublist and an empty sublist. The recursive process terminates on an empty or single-element sublist.



Completing the Sort

Process sublist 400, 450, 350 (A[2] – A[4])

Pivot = 450

The scanning process arranges the element in order 350, 400, 450. One more recursive call is needed with the two-element sublist 350, 400.

Process sublist 700, 650, 600 (A[6] – A[8])

Pivot = 650

After scanning, the elements are arranged in the order 600, 650, 700. The values 600 and 700 constitute two one-element sublists.

The Quick sort is complete and the resulting list is sorted.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
150	300	350	400	450	550	600	650	700	800

Quick Sort Implementation

The recursive algorithm partitions a list A[*low*] to A[*high*] about a pivot, which is selected from the middle of the list:

pivot = A[*mid*]; # *mid* = (*low*+*high*)/2

After exchanging the pivot value with A[*low*], set the indices *left* = *low*+1 and *right* = *high* to point at the beginning and end of the list.

The algorithm manages the two indices. *left* first moves up the list as long as it does not exceed *right* and points at elements that are less than or equal to pivot.

```
# index left traverses the elements that are less than or equal to pivot
while ( left <= right and A[left] <= pivot )
    left = left + 1                    # go to the next element
```

After *left* is positioned, *right* moves down the list as long as it refers to elements that are greater than pivot.

```
# scan down upper sublist; stopping when right identifies an element <= pivot.
while ( A[right] > pivot )
        right = right −1
```

On conclusion of this loop, if *left* < *right*, the indices identify two elements that are in the wrong sublists. The values are exchanged.

```
# exchange a large element in the lower sublist with
# a smaller element from the higher sublist
swap ( A, left, right )
```

The swapping of elements terminates when *right* is less than *left*. At this point, *right* identifies the top of the left sublist that contains the elements less than or equal to pivot. The index *right* is the pivot location in the list.
Retrieve the pivot value from A[*low*]: swap (A, *low*, *right*)

```
def  split (A, low, high):
# split the list into two sublists and rearranges the list
# so that the pivot is properly positioned at A[pos]

     # get the mid index and assign its value to pivot
     middle = (low + high) / 2
     pivot = A[middle]

     # exchange the pivot with the first item
     swap (A, middle, low)
     left = low + 1   # index for left search
     right = high     # index for right search

     while left <= right:
          # search from left for element > pivot
          while  left <= right and A[left] <= pivot:
                    left = left + 1

          # search from right for element <= pivot
          while A[right] > pivot:
                   right = right - 1

          # interchange elements if left and right
          # have not passed each other
          if  left < right:
              swap (A, left, right)

     # end of searches; place pivot in correct position
     pos = right
     A[low] = A[right]
     A[pos] = pivot

     return  pos       # pos is the final position of pivot
```

Quicksort uses recursion to process the sublists. After locating the pivot to split the list, we recursively call Quicksort with parameters *low* to *right*−1 (for lower sublist) and *right*+1 to *high* (for upper sublist).

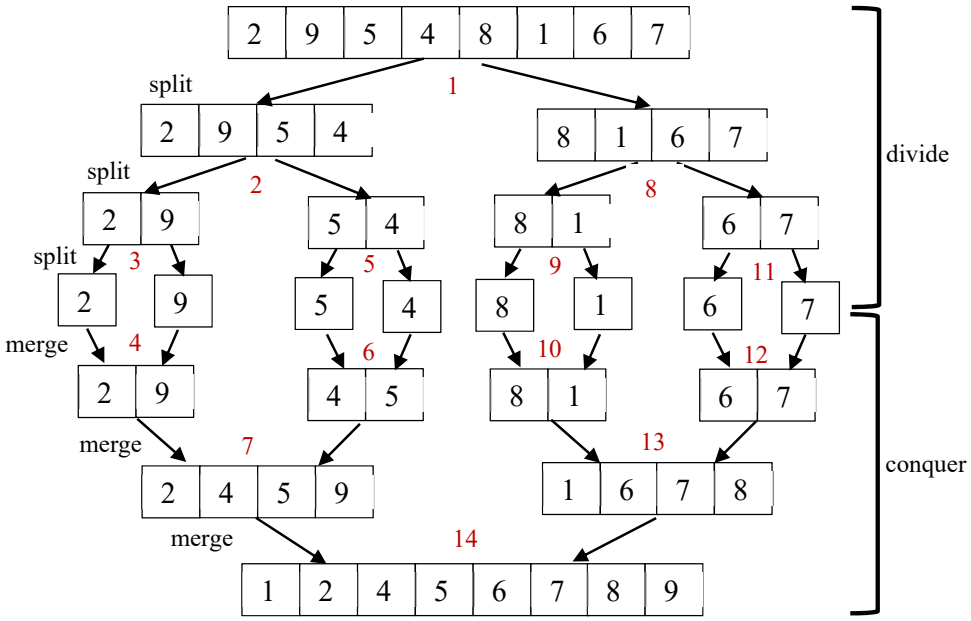
The stopping condition occurs when a sublist has fewer than two elements since a one-element or empty list is ordered.

```
def  quickSort (A, low, high):
# sort array elements A[low], . . . . . , A[high]

     if  low < high:     # list has more than one element
          # split into two sublists;
          # pos is the final position of pivot
          pos  =  split (A, low, high)

          quickSort (A, low, pos-1)  # quick sort left sublist
          quickSort (A, pos+1, high)# quick sort right sublist

     # else list has 0 or 1 element and requires no sorting
```

8.2.4 Merge Sort

Merge sort algorithm can be described recursively as follows: The algorithm divides the list into two halves and applies a merge sort on each half recursively. After the two halves are sorted, the algorithm then merges them.

The figure below illustrates a merge sort of a list of eight elements [2 9 5 4 8 1 6 7]. The list is split into 2 sublist [2 9 5 4] and [8 1 6 7]. Apply a merge sort on these two sub-lists recursively to split [2 9 5 4] into [2 9] and [5 4] and [8 1 6 7] into [8 1] and [6 7]. This process continues until the sub-list contains only one element. For example, list [2 9] is split into the sub-list [2] and [9]. Since [2] contains a single element, it cannot be further split. Now merge [2] with [9] into a new sorted list [2 9] and [5] with [4] into a new sorted list [4 5]. Merge [2 9] with [4 5] into a new sorted list [2 4 5 9] and finally merge [2 4 5 9] with [1 6 7 8] into a new sorted list [1 2 4 5 6 7 8 9].



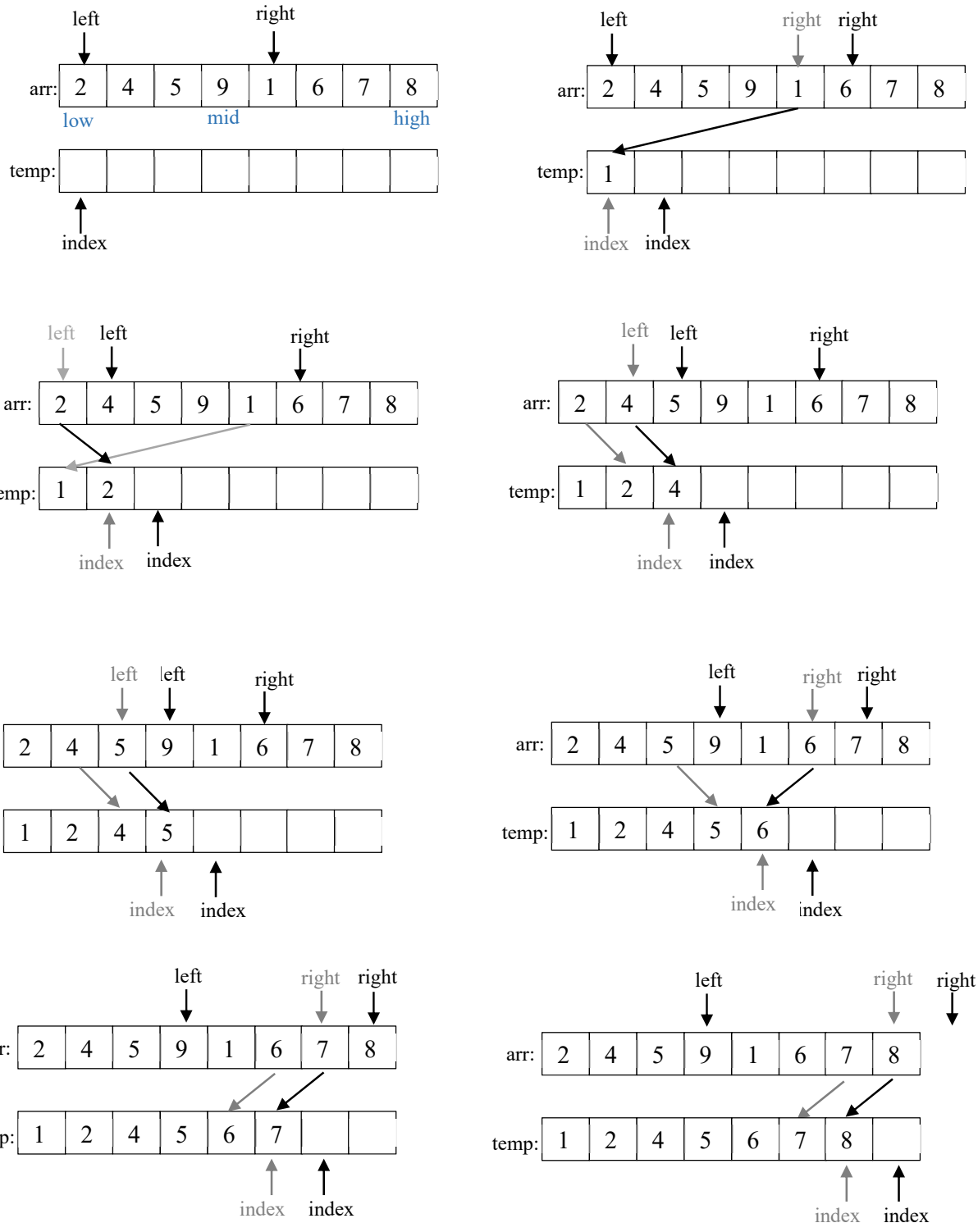
The recursive call continues dividing the list into sub-lists until each sub-list contains only one element. The algorithm then merges these small sub-lists into larger sorted sub-lists until one sorted list results.

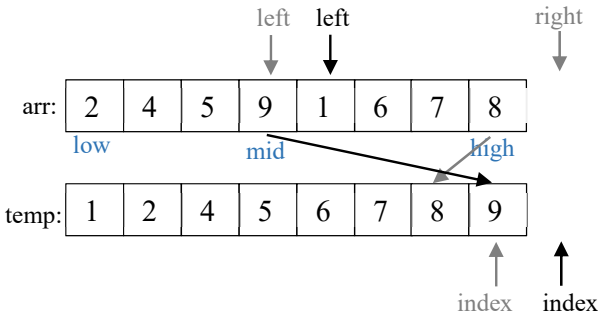
The merge sort algorithm is implemented as follows:

```
def mergeSort(arr, low, high):
    if low < high:
        mid = (low + high)//2
        # Sort first and second halves
        mergeSort(arr, low, mid)
        mergeSort(arr, mid+1, high)
        merge(arr, low, mid, high)
```

```
def merge(arr, low, mid, high):

    num = high - low + 1    #get number of elements
    temp = [0] * num #create temp array to store merged result

    # initialize
    left = low       # initial index of first subarray
    right = mid + 1  # initial index of second subarray
    index = 0        # initial index of merged array

    # merge both halves
    while left <= mid and right <= high:
        if arr[left] <= arr[right]: #left element is smaller
            temp[index] = arr[left]
            left += 1
        else:                       #right element is smaller
            temp[index] = arr[right]
            right += 1
        index += 1                  #increment index by 1

    #copy remaining elements of first subarray to temp
    while left <= mid:
        temp[index] = arr[left]
        left += 1
        index += 1

    #copy the remaining elements of second subarray to temp
    while right <= high:
        temp[index] = arr[right]
        right += 1
        index += 1

    #copied back into original array
    for i in range(0, num):
        arr[low+i] = temp[i]
```

The figures below illustrate how to merge the two sublists [2 4 5 9] and [1 6 7 8]





Tutorial 8B

1. For the following array A, show A after each pass using **bubble sort** and **insertion sort** to arrange the elements in ascending order

i	0	1	2	3	4	5
A[i]	30	50	70	10	40	60

2. For the following array A, use the **quick sort** algorithm to sort the elements in ascending order. Select the pivot from the midpoint in the list. During each pass, list all exchanges that will move a corresponding pair of elements in the lower and upper sublist. List the ordering of elements after each pass.

i	0	1	2	3	4	5	6	7	8	9
A[i]	45	20	50	30	80	10	60	70	40	90

3. N04P2Q6(b,c)

   (a) Explain how a quicksort can be used to sort the employee numbers in a transaction file into order, smallest first, using the following numbers as an example:

   6111 3217 2684 9163 7412 [5]

   (b) Two methods were considered to sort the transaction file, a bubble sort or a quick sort. With reference to the nature of the file to be sorted explain how a decision can sensibly be made. [2]

4. Use diagrams to show the various stages of **MergeSort** for the following lists of numbers:

   13, 22, 57, 99, 39, 64, 57, 48, 70

Assignment 8B

1. J01P1Q10(b)

In the following algorithm, NUMBER(i) is the value of the i^{th} data item in the array NUMBER.

```
LAST = NUMBER OF VALUES IN ARRAY NUMBER
COUNT = 0
START = 0
REPEAT
    FLAG = 0
    COUNT = COUNT + START
    REPEAT
        INCREMENT COUNT
        IF NUMBER(COUNT) < NUMBER(COUNT+1) THEN
            FLAG = 1
            TEMP = NUMBER(COUNT)
            NUMBER(COUNT) = NUMBER(COUNT+1)
            NUMBER(COUNT+1) = TEMP
        ENDIF
    UNTIL  COUNT = LAST – 1
    COUNT = 0
    INCREMENT START
UNTIL FLAG = 0
END
```

The array NUMBER contains the value 5, 6, 1, 3.

(i) State the values stored in each of the variables START, COUNT and FLAG after the algorithm is executed.

(ii) Give the state of the array NUMBER after the algorithm is executed.

(iii) Explain the significance of the variable FLAG.

[8]

2. N03P1Q4

An array, X, of integers has the following values stored in it:

X[1]	X[2]	X[3]	X[4]	X[5]
56	34	24	50	43

(a) Using a trace table with columns labelled i, j, X[1], X[2], X[3], X[4] and X[5] show how the contents of array X are sorted into order when the following algorithm for a Bubble sort is followed:

```
set i = 5
repeat
```

```
            for  j = 1 to i-1  do
                    if  X[j] > X[j+1]  then
                            swap X[j] and X[j+1]
                    endif
            endfor
            set  i = i -1
    until  i = 1
```

[8]

(b) Rewrite the code so that the sort process terminates as soon as possible when the array is fully sorted. [3]

(c) Give **two** different sets of test data that could be used to test the modified routine. Explain the purpose of each set of data. [4]

(d) When deciding which sort method to use in a program give three factors that need to be considered. [3]

3. N94P2Q11

(a) Give a detailed algorithm for a function **min(a, b, c)** which returns a value of 1, 2 or 3 to indicate which of the three values a, b or c is the **least**. [5]

(b) For values of N from 1 to 100, each of the arrays A[N], B[N] and C[N] contains a list of positive numbers, the values within each array increasing as N increases. Describe in detail an efficient algorithm making use of the function **min(a, b, c)** to merge the three arrays and print all 300 numbers in increasing order. [6]

4. J86P1Q3

Two sequential files A and B contain records of a fixed length with key field values in ascending order. The two files are to be merged to form a single sequential file C containing the same records with the key field values in ascending order. Each of the files A and B is terminated by a dummy record with a huge key field value, represented by ***hugekey***. File C is to be terminated similarly. Apart from the dummy records, all the key field values are supposed to be different from each other; it is therefore an error if any record in file A has the same key field value as any record in file B. Describe in detail an algorithm to carry out the merge. [8]

8.3 Big-O Notation

After learning all the searching and sorting algorithms, we need to analyze their performance so that we know which one to choose for a particular problem. The performance can be quantified in terms of time and space complexity, i.e. how much runtime and how large memory space each algorithm takes.

However, the running time for a searching algorithm definitely increases with the size of the list to be searched and also depending on some condition of the list (e.g. whether it has been sorted). So the common practice is that we study how the time cost changes with respect to its input size n in the worst case performance, and this is called Big-O Notation. Before moving onto searching and sorting algorithms, let's look at the Big-O Notation for some simple programs.

Constant Complexity: O(1)
Complexity of the program remains constant regardless of the input size.

```
def get_last(List):
     return List[-1]
```

Linear Complexity: O(n)
The time cost grows linearly and proportionally with the input size.

```
def get_sum(List):
     total = 0
     for item in List:
          total = total + item
     return total
```

Quadratic Complexity: $O(n^2)$

```
def multi_table(n):
#generate the multiplication table
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            print( i * j, end = ' ')
        print()
```

Clearly, linear search requires O(n) comparisons of the items in the list. Binary search halves the list in each iteration, so it requires $O(\log_2 n)$ comparisons. But don't fortget that binary search requires input to be an ordered list. For hash table, in ideal circumstances without collision, we found the item in one step, i.e. O(1). When collision occurs, it requires O(n). Hence a good hash function is very important. On the other hand, hash table may require more spaces as well. Each searching algorithm has its own pros and cons.

Sorting algorithms are more complicated and the nested loops makes both bubble sort and insertion sort quadratic complexity with $O(n^2)$. This is not a problem with small data sets, but with hundreds or thousands of elements, this becomes very significant.

Bubble sort does perform better for partially sorted lists because it is able to detect when a list is sorted and does not continue making unnecessary passes through the list. As a general sorting scheme, however, it is very inefficient because of the large number of interchanges that it requires. In fact, it is the least efficient of the sorting schemes.

Insertion sort also is too inefficient to be used as a general-purpose sorting scheme. However, the low overhead that it requires makes it better than bubble sort.

Quick sort and merge sort take $O(n \log_2 n)$ comparisons and are very efficient general-purpose sorting schemes and especially for large lists.