ITI 1120
Module 9: Sorting and Searching Algorithms

General Concepts:

1. Sorting and searching algorithms on data structures.

General Objective: Efficient sorting and searching a*lgorithms.*

Result:

1. Problem resolutions that require traveling efficiently through data structures.

Theme 1. *Sorting and searching algorithms on data structures*

Sub-theme: Linear search

```
def find(x, k):
   '''(list, int) -> bool
   Returns True if k is in the list, otherwise False"
   >>>a = [10, 28, -5, 6, 31, 25, 10, -7, 10]
   >>>find(a, 25)
   True
  '''

  find = False
  for val in x:
    if val == k:
      find = True  #the element is found
      break        # thus we stop the loop
  return find
```

Sous-thème: Find the posiiton of an element in a list

```
def whereIsK(x, k):
 '''(list, int) -> int
     Returns the position of k in the list x,
     and -1 if k is not found
 '''
 position = -1
 index = 0
 while index < len(x) and (position == -1):
   if x[index] == k:
     position = index
     break  #we found the first occurrence of k
   index = index + 1
 return position

>>>a = [28, -5, 6, 31, 25, 10, 6, -7, 10]
>>>whereisK(a, 6)
2
```

Sub-theme: Direct access using dictionaries

Works if the searched value is a key.

```
def findD(d, k):
  '''(dict, str) -> bool
  Returns True if k is in the dictionary, and False
otherwise"
  >>>d1 ={'marie':100, 'jean': 86, 'nadia': 98}
  >>>findD(d1, 'nadia')
  True
  '''

  find = k in d  # check if the key k exists
  return find
```

Sub-theme: Binary search in a sorted list

Overall if the value v we look for is small, we just need to look in the left side of the list. If the value is big, we look in the right side. If v is equal to the value in the middle, we stop. Otherwise we continue to search in half of the side just searched…

```
>>> binary _search([1, 3, 4, 5, 7, 9, 10], 9)
5
```

If v = 9, thus v is not equal to the value in the middle, 5. We look the half right side [7,9,10]. the new middle has the value 9. We have a hit!.

If v = 10, we continue to search in the right half [10] that we will hit in the middle.

If v = 8 we continue to search in the left half [7], we do not find it in the middle, we stop (when the list is empty) and returns -1 to specify that we did not find it.

Binary search in a sorted list

```
def binary_search(L, v):
  """ (list, int) -> int
  Returns the position of v in the sorted list L,
  or -1 if v is not in L. """
  position = -1
  # i and j delimitate the section searched
  i = 0
  j = len(L) - 1
  while i != j + 1:
    m = (i + j) // 2  # find the middle
    if L[m] == v:   # if found return the position
      position = m
      break
    elif L[m] < v:  # look on the right
      i = m + 1
    else:           # look on the left
      j = m - 1
  return position
```

Merge two sorted lists

```
def merge(l1, l2):
    ''' (list, list) -> list
    takes 2 sorted lists and returns a new list with
    the elements of the 2 listes in increasing order
    >>>merge([1,3,8], [2,3,10,12])
    [1, 2, 3, 3, 8, 10, 12]
    '''
    i,j = 0,0
    l3 = []
    while (i < len(l1)) and (j < len(l2)):
      if l1[i] <= l2[j]:
        l3.append(l1[i])
        i = i + 1
      else:
        l3.append(l2[j])
        j = j + 1
```

Merge two sorted lists (suite)

```
# one of the 2 lists is done (or both)
# test if l1 is done
while (i < len(l1)):
  l3.append(l1[i])
  i = i + 1
# test if l2 is done
while (j < len(l2)):
  l3.append(l2[j])
  j = j + 1
return l3
```

Sub-theme: Search algorithms Complexity

If the list has N elements, what is the maximum number of steps in the worst ase?

- Simple search in a list

  N

  O(N)   (O denotes the complexity order)

- Binary search in a sorted list

  N/2 + N/4 + … + 1

  $O(\log_2 N)$

- Search of a key in a dictionary

  O(1)

Sub-theme: Algorithms to sort out a list

Sorting = Ordering a list elements in an ascending order

- Simple sort (bubble sort)
- Sort by Insertion
- Sort by Selection
- Sort by merging (merge sort)

Sub-theme: Simple sort (bubble sort)

```
def bubbleSort(alist):
    ''' (list)-> None
    sort the list by switching consecutive elements that
    are not in order. Repete if exchanges already occured
    '''
    exchanges = True
    while exchanges:
        exchanges = False
        for i in range(len(alist)-1):
            if alist[i] > alist[i+1]:
                exchanges = True
                alist[i], alist[i+1] = alist[i+1], alist[i]

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Sub-theme: Sorting by Insertion

- General idea: Insert a value in the good position in the listed already sorted on the left. The fist element is in the right position for the moment.

3	6	4	2	7	9

  – Insert the second element in the good position in the list with an element [0:1] (insert 6 in the list [3] to get [3,6])

3	6	4	2	7	9

  – Insert the third element in the list of two elements [0:2] (Insert 4 in the list [3.6] to get [3,4,6]

3	4	6	2	7	9

  – Continue until the end of the list, for the last element. (Insert 2 in the list [3,4,6] to get [2,3,4,6]. Then 7 and after 9)

2	3	4	6	7	9

2	3	4	6	7	9

Sorting by Insertion.

Without changing the initial list. We return a new list.

```
def sort_by_insertion(L):
  """ (list) -> list
  Produce a new list L2 with the elements of L
  in ascending order
  >>> L = [3, 4, 7, -1, 2, 5]
  >>> L2 = sort_by_insertion (L)
  >>> L2
  [-1, 2, 3, 4, 5, 7]
  >>> L
  [3, 4, 7, -1, 2, 5]  ""«

  i = 0
  L2 = []
  while i != len(L):
    insert(L2, L[i])
    i = i + 1
  return L2
```

Sorting by Insertion(suite)
Without changing the initial list.

```
def insert(L, val):
  """ (list, int) -> None
  Precondition: L is already sorted
  Insert val in the correct position in L.
  >>> L = [1,3,6]
  >>> insert(L, 2)
  >>> L
  [1,2,3,6]
  """
  # find where to insert val. Start at the end of L and
  # search an element of smaller value
  i = len(L)
  while i != 0 and L[i - 1] >= val:
     i = i - 1
  L.insert(i, val)
```

Sorting by Insertion
The initial list is modified.

```
def sort_by_insertion(L):
  """ (list) -> None
  Sort the list L in ascending order
  >>> L = [3, 4, 7, -1, 2, 5]
  >>> sort_by_insertion (L)
  >>> L
  [-1, 2, 3, 4, 5, 7]
  """
  i = 0
  while i != len(L):
    insert(L, i)
    i = i + 1
```

Sorting by Insertion (suite)

The initial list is modified.

```
def insert(L, b):
    """ (list, int) -> None
    Precondition: L[0:b] is already sorted
    Insert L[b] in the correct position in L[0:b + 1].
    >>> L = [3, 4, -1, 7, 2, 5]
    >>> insert(L, 2)
    >>> L
    [-1, 3, 4, 7, 2, 5]  """
    # find or insert L[b]
    # start at the end of L[b] and search an element
    i = b  # of smaller value
    while i != 0 and L[i - 1] >= L[b]:
        i = i - 1
    # move L[b] to index i
    value = L[b] # move the values after i on the right.
    del L[b]
    L.insert(i, value)
```

Sub-theme: Sorting by Selection

- Genera; ideas: select the smallest element (minimum) and put at the proper position, on the left. The left side is sorted out and continue to increase until the list is sorted out.

3	6	4	2	7	9

  - First of all, find the position ★ that contains the minimum

			★		
3	6	4	2	7	9

  - Secondly, switch that position with the first element

2	6	4	3	7	9

  - Continue with the list starting with the second element, 3-rd, ….

			★		
2	6	4	3	7	9
2	3	4	6	7	9

Sorting by Selection.

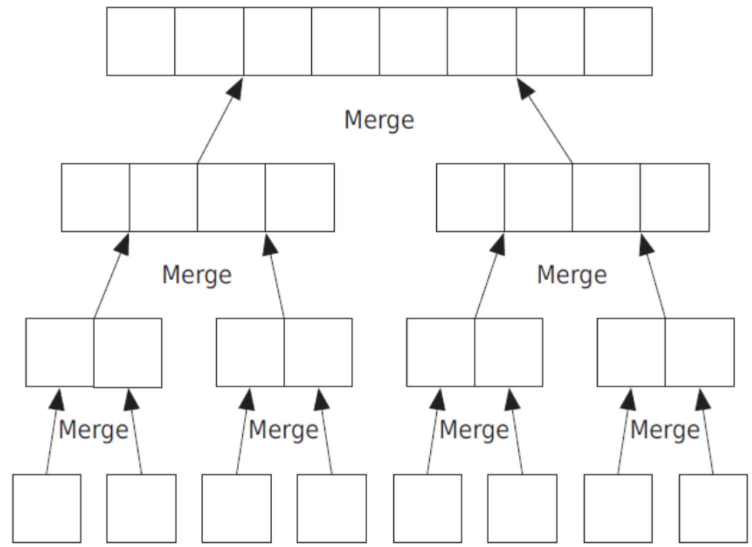
The initial list is modified.

```
def sort_by_selection(L):
    """ (list) -> None
    Sort the elements of L in ascending order
    >>> L = [3, 4, 7, -1, 2, 5]
    >>> sort_by_selection (L)
    >>> L
    [-1, 2, 3, 4, 5, 7]
    """
    i = 0
    while i != len(L):
        # move the minimum to the left
        min = find_min(L, i)
        L[i], L[min] = L[min], L[i]
        i = i + 1
```

Sorting by Selection (find_min)

```
def find_min(L, b):
  """ (list, int) -> int
  Precondition: L[b:] is not empty.
  Returns the index of the minimal value in L[b:].
  >>> find_min([3, -1, 7, 5], 0)
  1
  >>> find_min([3, -1, 7, 5], 2)
  3
  """
  min = b # the index of the temporary minimum
  i = b + 1
  while i != len(L):
    if L[i] < L[min]:
    # we found a smaller element than the temporary
    # minimum
      min = i
    i = i + 1
  return min
```

Su-theme: Sorting by merging (merge sort)




```
def mergesort(L):
  """ (list) -> None
  Sort elements of L.
  >>> L = [3, 4, 7, -1, 2, 5]
  >>> mergesort(L)
  >>> L
  [-1, 2, 3, 4, 5, 7]
  """
  # Produce a temporary list. Each element of the list
  # produces a one element List. We can start the
  # process of merging
  temp = []
  for i in range(len(L)):
       temp.append([L[i]])
```

Sorting by merging (merge sort) (suite)

```
# The two lists to be merged are temp[i] and temp[i+1].
# There is at least 2 lists to merge, we merge
# them and we repeat the process.
i = 0
while i < len(temp) - 1:
  L1 = temp[i]
  L2 = temp[i + 1]
  newL = merge(L1, L2)
  temp.append(newL)
  i = i + 2
# Copy the result in L.
if len(temp) != 0:
  L[:] = temp[-1][:]
```

Sub-theme: Sorting algorithms complexity

If the list has N elements, what is the maximum number of steps in the worst cases? (O = magnitude order)

- Simple sort (bubble sort)

  $(N - 1) + (N - 1) + \ldots + (N-1) = (N - 1)\ (N - 1) = N*N - 2*N + 1$

  $O(N^2)$

- Sorting by Selection and sorting by Insertion

  $N + (N-1) + (N-2) + \ldots + 1 = N\ (N+1)/2 = (N*N + N)\ /2$

  $O(N^2)$

- Sort by merging

  $O(N \log_2 N)$

Sub-theme: Execution time Measurement

```
# generate a list of 10000 random alues elements
L = random.sample(range(10000), 10000)

print(« SORTING ALGORITHM:")
     copy = L[:]
     t1 = time.perf_counter()
     sorting_algo(copy)
     t2 = time.perf_counter()
     print(t2-t1) # Execution time

# We replace sorting_algo with our algorithms
# and with the predefined function sort()

BUBBLE SORT:
29.11057368348544
SORTING BY SELECTION:
18.46829047618859 6
SORTING BY INSERTION:
7.289117465590792
MERGE SORT:
0.09519784972807344
PYTHON SORT:
0.003839320149005232
```

Question

If the list has N = 10,000 elements, how many operations (multiplications) are processed in the following function?

```
def my_fonction(a):
    s = 0
    for x in a:
      for y in a:
         s += x*y
    return s
```

Pssibles responses (choose one):

a) 10,000
b) 100,000
c) 100,000,000
d) 1000

Question – *Solution*:

Correcte response: c)
Explaination: The outsie loop is being processed 10,000 anf for each of its execution, the inside loop is processed 10,000. Donc 10,000 * 10,000 = 100,000,000

Conclusion

- Algorithms to visit data structures de données.
  - Searching algorithms.
  - Sorting algorithms.
  - Complexity and execution time.