ICS Spring 2020

Midterm Exam

- The exam contains **3** parts with a total of **12 pages**, feel free to use any white space to draft your work.
- You have **120 minutes** to finish the whole exam.
- You need to **download** the following items from **Midterm** entry under **Assignments** on **NYU Classes**:
    - **Answers.txt**
    - **insertionSort_student.py**
    - **icsList_student.py**
    - **maxCoins_student.py**

- You can use the Gaddis textbook, the MIT book, lecture slides and lab materials, and anything you need.
- ***Read problems carefully and follow specifications.***
- When you finish your exam, please upload the files to **Midterm** entry under **Assignments** on **NYU Classes**.

Good luck!

Read the questions, think ***before*** coding

Name:_________________

NetID:_________________

Part 1: True or False (20 points, 4 for each)

(put into **answer.txt**)

1. An algorithm can be represented by a flow-chart.
2. It will make a computer faster if we equip it with a 2TB hard disk than 1TB.
3. OOP is an idea designed to trade space for time.
4. By using the Hash table, we always find an item with time complexity O(n).
5. The size of the powerset of a set of n elements is 2^n.

Part 2: Complexity analysis (25 points, 5 for each)

(put into **answer.txt**)

Find the complexity using **big-O** notation, with N as the input size. For example, the following code has complexity O(N):

```
def one_iter(in_list):
""" in_list contains a list of integers """
     sum = 0
     for x in in_list:
          sum += x
     return sum

L = list(range(N))
Output = one_iter(L)
```

Questions: Find the complexity for the following code snippets:

1.

```
def one_iter_nop(in_list):
     """ in_list contains a list of integers """
     sum = 0
     return sum
     while in_list:
         in_list.pop()

L = list(range(N))
Output = one_iter_nop(L)
```

2.

```
def int_to_str(x):
      """Assumes x is a nonnegative int
      Returns an hexadecimal string representation of x."""

      digits = '0123456789ABCDEF'
      if x == 0:
          return '0'
      result = ''
      while x > 0:
          result = digits[x%16] + result
          x = x//16
      return result

Output = int_to_str(N)
```

3.

```
def add_digits(list_n):
      """Assumes list_n is a list of non-negative int
       Convert each element into an hexadecimal string
       Returns the sum of all the digits"""

      digits = {‘0’: 0, ‘1’: 1, ‘2’: 2, ‘3’:3, ‘4’:4, ‘5’: 5, ‘6’:6
                ‘7’: 7, ‘8’:8, ‘9’:9, ‘A’:10, ‘B’:11, ‘C’:12, ‘D’:13
                ‘E’:14, ‘F’:15}
      for n in list_n:
           string_rep = int_to_str(n) #int_to_str(n)is defined in (2)
           val = 0
           for c in string_rep:
                val += digits[c]
      return val

L = [N for i in range(N)]
Output = add_digits(L)
```

```
4.
def bubble_sort(my_list):
    N = len(my_list)
    for i in range(1,N):
        for j in range(0,N-1):
           if my_list[j] > my_list[j + 1]:
              my_list[j], my_list[j + 1] = my_list[j + 1], my_list[j]
    return my_list

L = list(range(N))
Output = bubble_sort(L)

5.
def bubble_sort_optimized(my_list):
    N = len(my_list)
    for i in range(1,N):
        swapped = False
        for j in range(0,N-i):
            if my_list[j] > my_list[j + 1]:
               my_list[j], my_list[j + 1] = my_list[j + 1],
my_list[j]
               swapped = True
        if swapped == False:
           break
   return my_list

Output = bubble_sort_optimized(list(range(N))
```

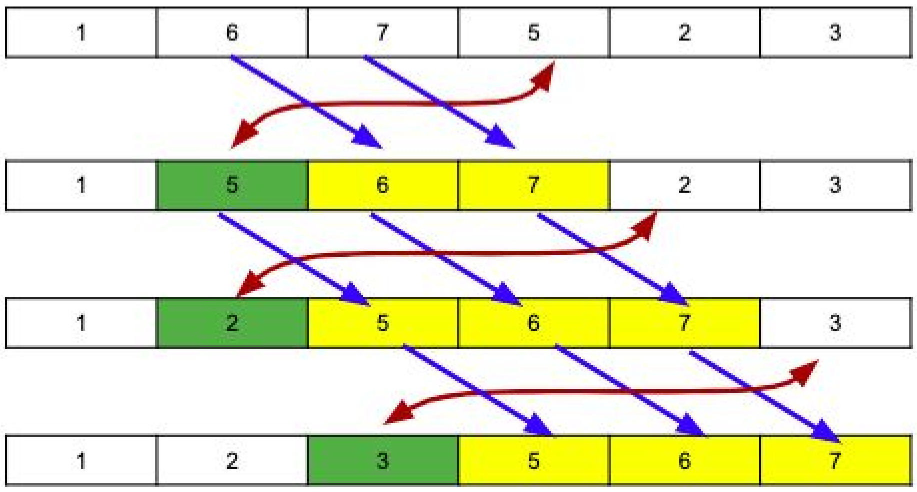
6.**Bonus**

What kind of input will make `bubble_sort_optimized` performs 1) the best, and 2) the worst?

Part 3: Comprehensive Programming (55 points)

Problem 1: Insertion sort and complexity analysis.

Insertion sort is one simple sort like bubble sort. However in real practice, it is about twice as fast as the bubble sort. To understand how insertion sort works, let’s think about sorting a half sorted list [1, 6, 7, 5, 2, 3]. Since the left to 5 are partially sorted, we can sort 5 by inserting it in the appropriate place in the left sorted group. We have to shift 6 and 7 to the right so that provides a room for 5. Similarly, we can sort 2 and 3 by inserting them to the left and shifting the numbers that are greater than them to the right. The following figure shows the process.



Briefly, the insertion sort compares the elements in the sorted sublist with the unsorted element one by one from **the end** of the sorted sublist. It inserts the unsorted element right behind the first element that is smaller than it. The pseudocode of the insertion sort is as follows,

```
func insertionSort(a): #a is an 1-d array that has N elements.
    for i form 2 to N: #starting from the 2nd element to the last one
        m = i
        while m >= 2 and a[m]< a[m-1]:
            temp = a[m]
            a[m] = a[m-1]
            a[m-1] = temp
            m = m - 1
end func
```

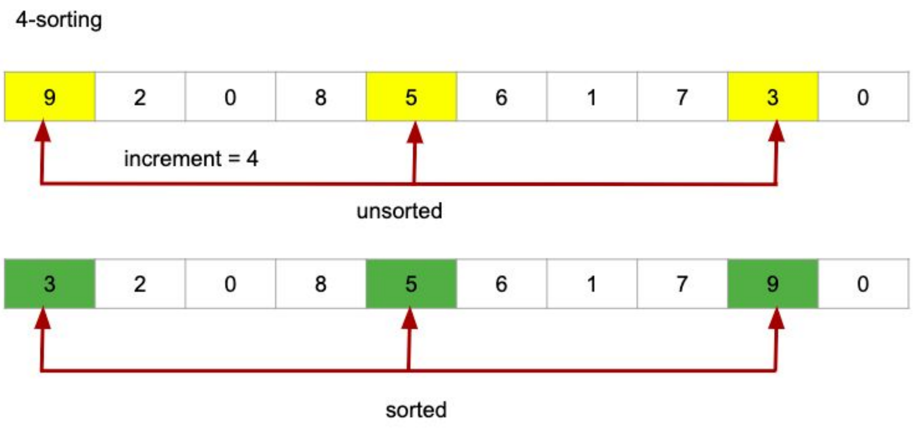
(a) Write a function `insertionSort()` that sorts the input list in ascending order. Your function should sort the input list, like the following example.

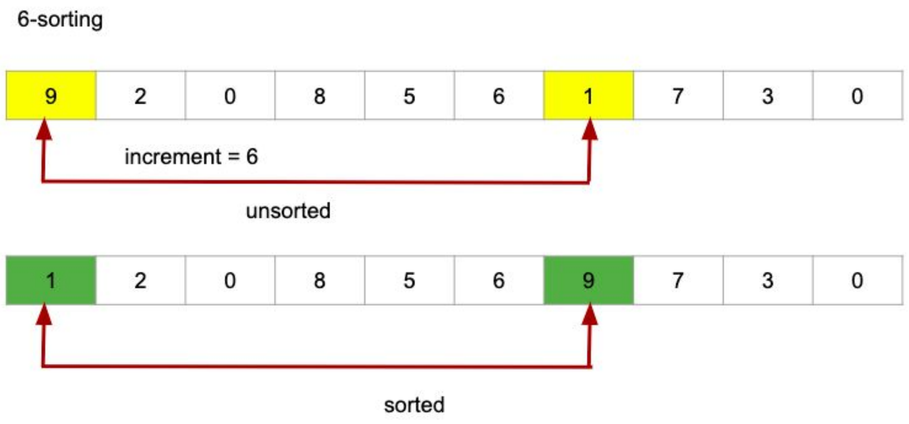
```
In [12]: lst = [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]

In [13]: insertionSort(lst)

In [14]: lst
Out[14]: [0, 0, 1, 2, 3, 5, 6, 7, 8, 9]
```

(b) What is the time complexity of your code in Big-O notation? Give a brief explanation. Please write down your answer in the **answer.txt**

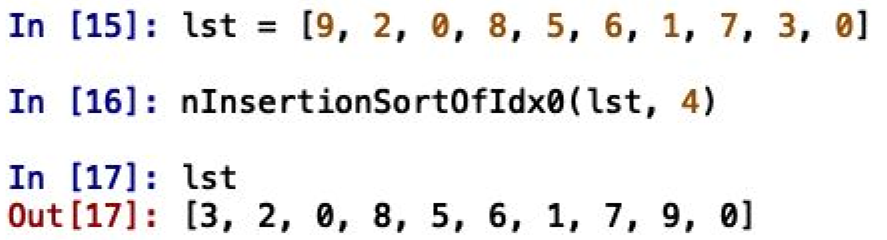
(c) Bonus: what is the spatial complexity of your code in Big-O notation? Give a brief explanation. (Please write down your answer in the **answer.txt**)

(d) Implementing a simplified h-insertion sort. h-insertion sort is a variation of insertion sort. It performs sorting in **the same way** as insertion sort. But instead of sorting the whole input list, it only sorts the h-spaced elements in the list, where h is the increment of spacing between elements. In this case, we only sort the h-spaced list whose elements are counted based on the **first** element in the list. For example, in a 10-element list, if the gap is 4, the sublist we need to sort is [9, 5, 3](i.e., corresponding to list[0], list[4], and list[8]). If the gap is 6, then, the 6-spaced sublist is [9, 1] (i.e., corresponding to list[0] and list[6]).

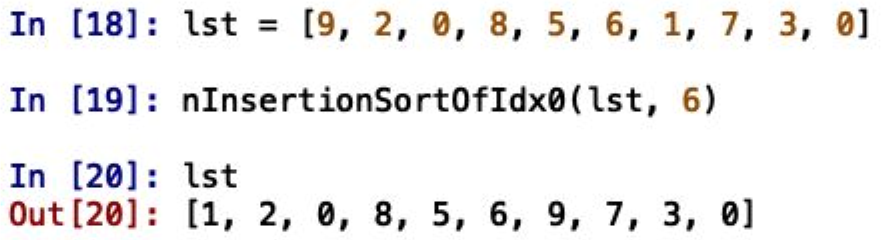




Write a function `hInsertionSortOfIdx0()` that performs h-insertion sort on the input list in ascending order. The function should have two input arguments: `lst` and `h` which represent a list and an increment respectively. The function should sort the h-spaced elements in the input list, like the following two examples.



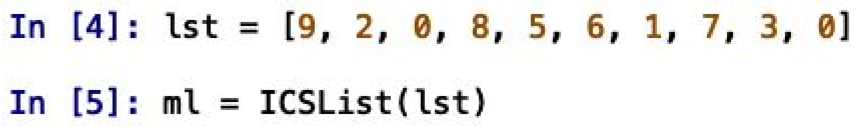
```
In [15]: lst = [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]

In [16]: nInsertionSortOfIdx0(lst, 4)

In [17]: lst
Out[17]: [3, 2, 0, 8, 5, 6, 1, 7, 9, 0]
```



```
In [18]: lst = [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]

In [19]: nInsertionSortOfIdx0(lst, 6)

In [20]: lst
Out[20]: [1, 2, 0, 8, 5, 6, 9, 7, 3, 0]
```

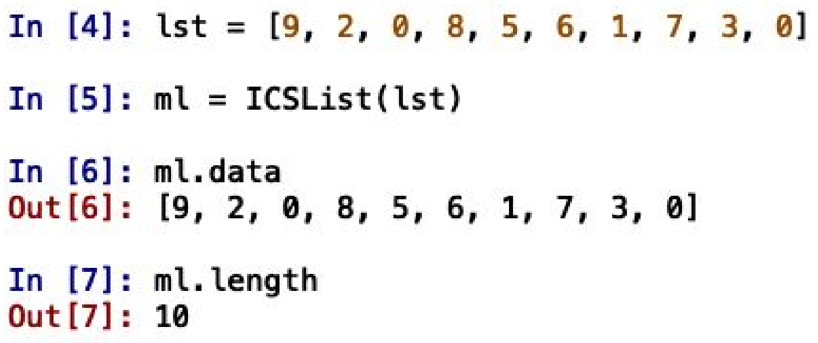
Problem 2. Implement a ICSList class

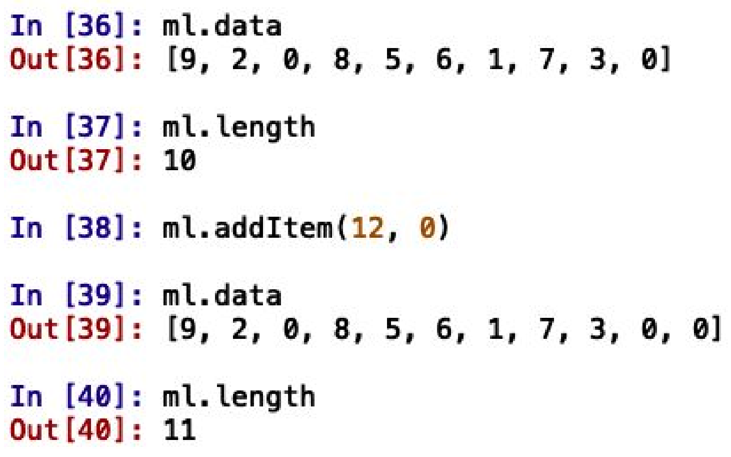
We want to create a new class `ICSList` to sort data. It stores a list of data items and has a few methods for data manipulation. For example, given a 10-element list we can initialize a instance ml as follows:

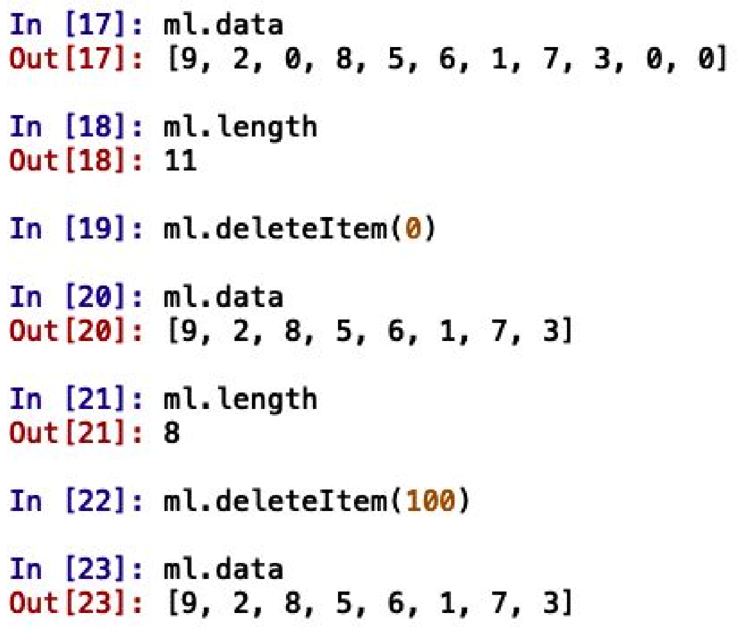


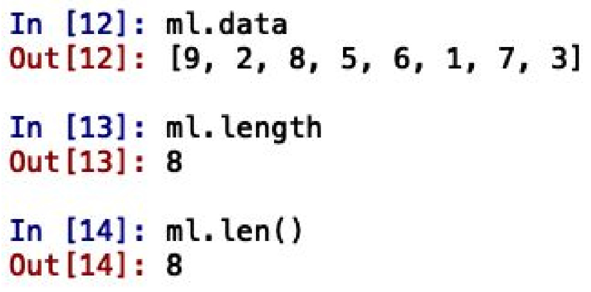
```
In [4]: lst = [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]

In [5]: ml = ICSList(lst)
```

(a) ICSList has several elementary methods. They are listed in the following table.

`init(l=[])`	Initialize with a given list l. The class has two attributes. One is the `data` which is the given list; another one is `length`, which is the length of the list.
	
	`In [4]: lst = [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]`
	
	`In [5]: ml = ICSList(lst)`
	
	`In [6]: ml.data`
	`Out[6]: [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]`
	
	`In [7]: ml.length`
	`Out[7]: 10`

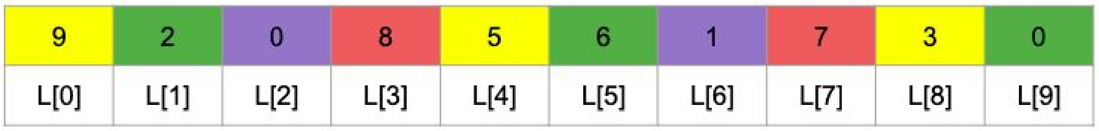
`addItem(idx, x)`	This method inserts x into the `data` by index specified. If the index is larger than the length of the data, it appends x at the end of the data.
	
	`In [36]: ml.data`
	`Out[36]: [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]`
	
	`In [37]: ml.length`
	`Out[37]: 10`
	
	`In [38]: ml.addItem(12, 0)`
	
	`In [39]: ml.data`
	`Out[39]: [9, 2, 0, 8, 5, 6, 1, 7, 3, 0, 0]`
	
	`In [40]: ml.length`
	`Out[40]: 11`

`deleteItem(x)`	This method removes all x in the dataset.
	
	`In [17]: ml.data`
	`Out[17]: [9, 2, 0, 8, 5, 6, 1, 7, 3, 0, 0]`
	
	`In [18]: ml.length`
	`Out[18]: 11`
	
	`In [19]: ml.deleteItem(0)`
	
	`In [20]: ml.data`
	`Out[20]: [9, 2, 8, 5, 6, 1, 7, 3]`
	
	`In [21]: ml.length`
	`Out[21]: 8`
	
	`In [22]: ml.deleteItem(100)`
	
	`In [23]: ml.data`
	`Out[23]: [9, 2, 8, 5, 6, 1, 7, 3]`

`len()`	This method is a getter of attribute length, which returns the length of the data.
	
	`In [12]: ml.data`
	`Out[12]: [9, 2, 8, 5, 6, 1, 7, 3]`
	
	`In [13]: ml.length`
	`Out[13]: 8`
	
	`In [14]: ml.len()`
	`Out[14]: 8`

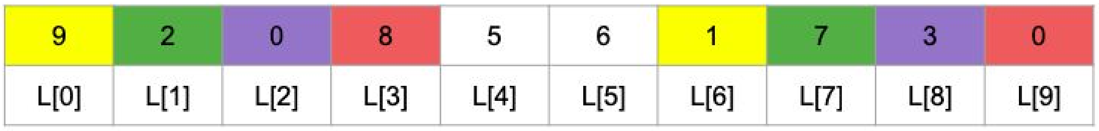
(b) ICSList can sort its data by Shell sort. Shell sort is a generalized insertion sort. It sorts the data set by running h-insertion sort over all h-spaced sublists for several times with a different increment h each time. For example, the following figure is a 10-element list. When h = 4, Shell sort conducts the 4-insertion sort, that is, it sorts the index 0 based sublist: [9, 5, 3], index 1 based: [2, 6, 0], index 2 based: [0, 1] and index 3 based: [8, 7]. When h = 6, Shell sort conducts the 6-insertion sort, that is, sorting every of the 6-spaced sublists [9, 1], [2, 7], [0, 3], [8, 0]. (The 4-spaced and 6-spaced sublists of a 10-element list are shown in the following figure. Each sublist is represented by a color.)

All 4-spaced (increment h = 4) sublists in a list L which has 10 elements.



9	2	0	8	5	6	1	7	3	0
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]	L[7]	L[8]	L[9]

All 6-spaced (increment h = 6) sublists in a list L which has 10 elements.

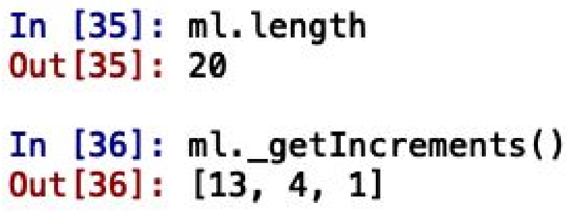


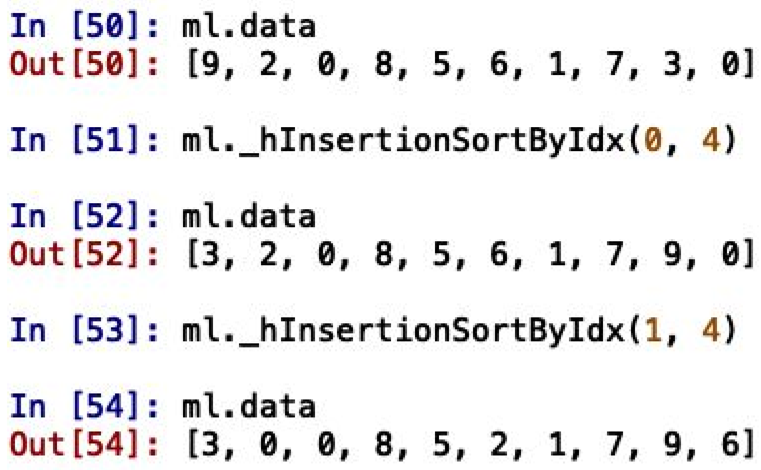
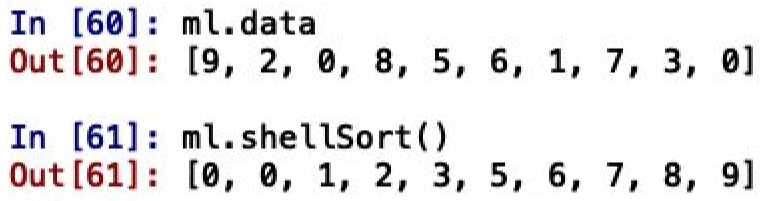
9	2	0	8	5	6	1	7	3	0
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]	L[7]	L[8]	L[9]

Given a set of increments, Shell sort starts the h-insertion sort with the largest gap, then the second largest one and so on until it becomes 1. For example, given a set of increments {1, 4, 13}, Shell sort sorts the list by firstly running 13-insertion sort on the list, then, 4-insertion sort, and finally 1-insertion sort.

For Shell sort, it always contains the smallest increment $h_1 = 1$, and the other increments are generated by the following equation:
$h_{n+1} = h_n \times 3 + 1$
where h_n is the n-th smallest increment (i.e., the largest increment currently). The generation of increments stops when h_{n+1} > the size of the data set. For example, for a 10-element list, we have
$h_1 = 1$
$h_2 = h_1 \times 3 + 1 = 4$
$h_3 = h_2 \times 3 + 1 = 13 > 10$, the generation stops.
So, the increment sequence is [4, 1].

To implement Shell sort, it needs the functions list in the following table.

`_getIncrements()`	This is a private method. It **returns** a list of increments [h_n, …, h_1] in descending order. The increments are calculated by $h_{n+1} = 3*h_n + 1$, where $h_1 = 1$ and h_n <= `length`.
	
	`In [35]: ml.length`
	`Out[35]: 20`
	
	`In [36]: ml._getIncrements()`
	`Out[36]: [13, 4, 1]`

`_hInsertionSortByIdx(idx, h)`	This is a private method. It conducts h-insertion sort on `data` with a specified index. It has two arguments: the base index and the increment. You can modify the `hInsertionSortOfIdx0()` in Problem 1 (d) to make it work with a given index. In the following example, we run `_hInsertionSortByIdx()` twice with h = 4 and idx = 0 and 1 respectively.  `In [50]: ml.data` `Out[50]: [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]` `In [51]: ml._hInsertionSortByIdx(0, 4)` `In [52]: ml.data` `Out[52]: [3, 2, 0, 8, 5, 6, 1, 7, 9, 0]` `In [53]: ml._hInsertionSortByIdx(1, 4)` `In [54]: ml.data` `Out[54]: [3, 0, 0, 8, 5, 2, 1, 7, 9, 6]`
`shellSort()`	This method performs Shell sort on the `data` and **returns** it. Note: You should call the `_getIncrements()` and `_hInsertionSortByIdx()`  `In [60]: ml.data` `Out[60]: [9, 2, 0, 8, 5, 6, 1, 7, 3, 0]` `In [61]: ml.shellSort()` `Out[61]: [0, 0, 1, 2, 3, 5, 6, 7, 8, 9]`

Problem 3. Burst balloons

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] * nums[i] * nums[right] coins. Here left and right are adjacent indices of i. After the burst, the left and right then become adjacent. Find the maximum coins you can collect by bursting the balloons wisely.
Note:

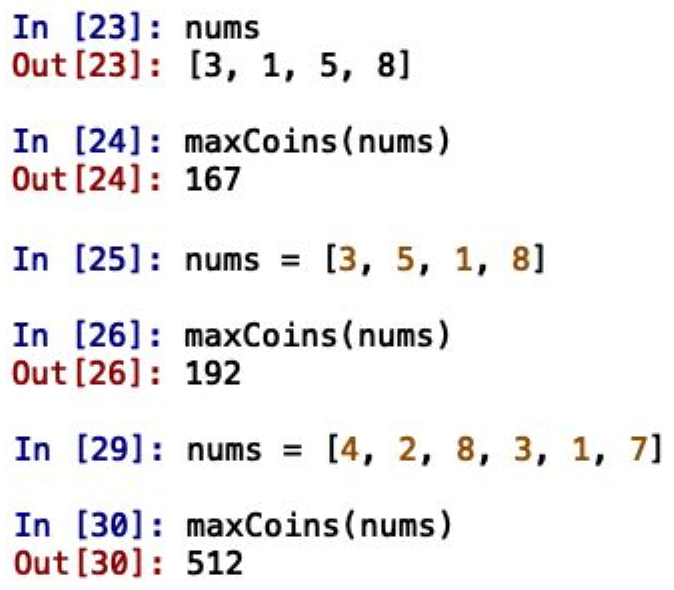
- You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.

Example:
Given [3, 1, 5, 8]
Return 167
nums = [3,1,5,8] -- >[3,5,8] -- > [3,8] -- > [8] -- > []
coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167

Given [3, 5, 1, 8]
Return 192
nums = [3, 5,1,8] -- >[3,5,8] -- > [3,8] -- > [8] -- > []
coins = 5*1*8 + 3*5*8 + 1*3*8 + 1*8*1 = 192

Given [4, 2, 8, 3, 1, 7]
Return 512
nums = [4, 2, 8, 3, 1, 7] -- >[4,8,3,1,7] -- > [4,8,3,7] -- > [4,8,7] -- > [4, 7]-- >[7] -- > []
coins = 4*2*8 + 3*1*7 + 8*3*7 + 4*8*7 + 1*4*7 + 1*7*1 = 512

(a) Write a function `maxCoins()` that returns the maximum coins you can collect by bursting the balloons. [Hint: you may use recursion.]



```
In [23]: nums
Out[23]: [3, 1, 5, 8]

In [24]: maxCoins(nums)
Out[24]: 167

In [25]: nums = [3, 5, 1, 8]

In [26]: maxCoins(nums)
Out[26]: 192

In [29]: nums = [4, 2, 8, 3, 1, 7]

In [30]: maxCoins(nums)
Out[30]: 512
```

(b) (Bonus) Write a function `fast_maxCoins()` by using dynamic programming.