Lab 4

SORT

Problem 1: Remove Duplicates

- Describe and analyze an efficient method for removing all duplicates from a list *L of n elements*
- After removing duplicates, the remaining elements should retain the order they had before
- Required Time - O(n)
- Example:

L = [7,2,2,5,7,2,1,7,3,100000000]

remove_dups(L) => [7,2,5,1,3,100000000]

L = [5,0,1,0,9,2,1,0,5]

remove_dups(L) => [5,0,1,9,2]

Problem 2: Has_Dup

- Given a list L of *n integers, write an efficient algorithm* ***has_dup for determining whether there are two equal elements in L***
- What is the running time of your method? Is it the best running time possible?
- Examples:
  - **L = [5,1,0,4,2,9,7,4,3]**

    has_dup(L) => True
  - **L = [1,2,3]**

    has_dup(L) => False

Problem 3: Is_Element_Sum

- Let *A and B be two lists of n integers each. Given an integer m, required:*
- ***is_elements_sum for determining if there is an integer a in A and an integer b in B such that m = a+b***
  - *Time complexity – O(n)*
  - *Space complexity – O(n)*

    - **A = [5,1,0,4,2,9]**
    - **B = [2,4,0,7,1,8]**
    - **m = 13**
    - **is_elements_sum(m, A, B)** **=> True! 13 = 9 + 4 (or 5+8)**
    - **is_elements_sum(18, A, B)** **=> False**

Problem 3b: Is_Element_Sum

- Let *A and B be two* ***<u>sorted</u>*** *lists of na,nb integers each. Given an integer m, required:*
- ***is_elements_sum for determining if there is an integer a in A and an integer b in B such that m = a+b***
  - *Time complexity – O(n)*
  - *Space complexity - O(1)*
    - **A = [0,1,2,4,9]**
    - **B = [0,1,2,4,7,8]**
    - **m = 13**
    - **is_elements_sum(m, A, B) => True! 13 = 9 + 4 (or 5+8)**
    - **is_elements_sum(18, A, B) => False**

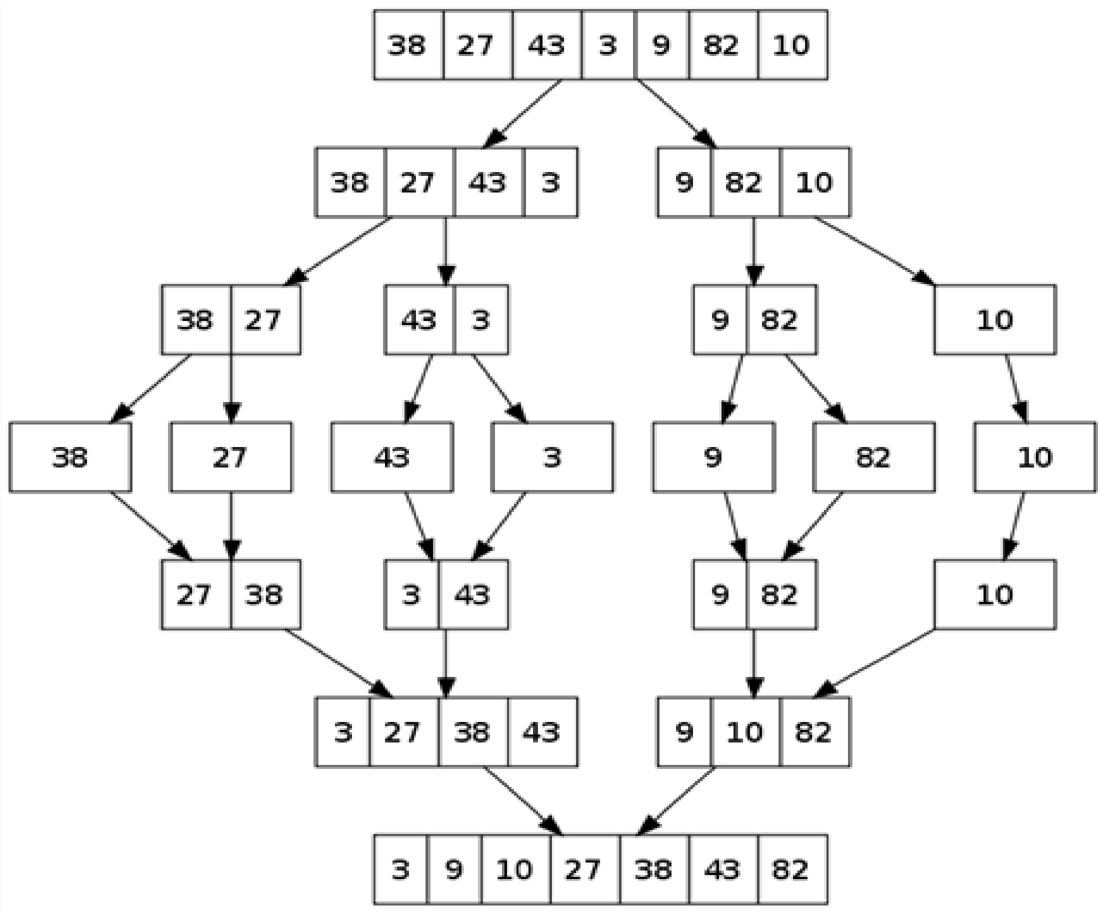
Problem 4:

- If L is a list of n integers smaller than n^3 then it can be sorted in O(kn) time (where k is constant for all n's)
- Find such an algorithm and prove that its time complexity is O(n)

  space complexity: O(n)

Hint: look again on Radix sort

Problem 5: Merge-Sort



Merge-Sort NOT in place

- Run the Merge not in-place algorithm. Merge_sort.py
- Use the sort_bench.py to test it.
- Write the merge-sort **in_place** algorithm.
- Write a bench program to compare the two algorithms.
- Fill the following table (what are your conclusions ?)

	Best		Worst		Average	
n	NIP	IP	NIP	IP	NIP	IP
10000						
20000						
40000						
80000						
160000						
320000						
640000						

Improved Merge-Sort

- For small arrays, Insertion sort is more efficient than Merge-Sort
- Add to the Merge-Sort Not in place a forth parameter k.
- If the length of the array is smaller than k, Insertion sort will be used instead.
- Check by experimenting, what is the optimum k size.