Homework3

1. Find the n/4 number of a list of n numbers.

   Solution:

   This problem is equivalent to the following problem:

   - finding the k-th number of a list of numbers. (k = n/4)

   Following is the solution to find the k-th number of a list.
   We can start with randomly selecting a number from the list as a pivot. Using this pivot number, we partition the list of numbers into 2 segements, where the left segament has all elements smaller than the pivot number while the right segament has all elements larger than the pivot number. Suppose, the index of the povit number is **p**;

   - if k == p, we are done, return list[p] as the result;
   - if k < p, we apply recursion to the list in the range from 0 - p-1, that is to find k-th number in the list[0...p-1];
   - otherwise, k must be bigger than p, in this case, the number we want to find must fall into the right segament. so the number we want to find is (k-p)-th number in list[p+1 .. n-1].

   pseudo code is as follows:

```
findKth(list[0...n-1], k){
    p = random(0,n-1);
    pivot = list[p];
    define two lists, A1 and A2;
    for(int i; i < list.size(); i++){
      if(list[i] < pivot){
        a1.add(list[i]);
      }
      if(list[i] > pivot){
        a2.add(list[i]);
      }
    }
    if (k == p) return pivot;
    if (k < p) return findKth(A1, k);
    if(k > p) return findKth(A2,k - p);
}
```

Analysis of the algorithm

Using randomized pivot number ensures that on average reduce the input scale by half. so the recursion can be denoted as follows.

```
T(n) = T(n/2) + n
```

solve this recursion: Acoording to Master Theorem:
a=1, b=2, log1 = 1, so T(n) = O(n);
therefore the time complexity of this proposed algorithm is O(logn).

2. Show an example where heap sort requires cnlogn steps,c a constant.

Solution:

- Given an unsorted array: A[0,...,n-1];

Steps:

1. Build a max-heap on the array A[0,...,n-1],where n = A.length.
2. Due to the maximum element of the array is stored at the root A[1], we can delete it and put it into its correct final position by exchanging it with A[n-1].Then we need to max-heapify the children of the root(A[0,...,n-2]) for restoring the max-heap property. Then,Loop step2.

pseudo code is as follows:

```
Heapsort(A)

  //Build-Max-Heap(A)
A.heap-size = A.length;
for i = A.length/2 downto 1
   max-heapify(A,i)

   //delete the root and put it to correct position
for i = A.length -1 downto 2
    exchange A[0] with A[i]
    A.heap-size = A.heap-size -1
    max-heapify(A,0)
```

example is as follows:
Given an array A = [4,2,5,10,12,8,11],

3. **Join k sorted list into one in O(nlogk) time, where n is the total number of the items.**

   Solution:

   Algorithm Description:

   1. create **an array of size n** to store the result. and create **a min heap with its capacity being k**.
   2. put the first element in each list to the heap
   3. remove the min value *element_min* from the heap and push it to the result array. at the same time, insert the first element from the list where the *element_min* was from.
   4. repeat step #3 until all list is empty.

   Time complexity:

   - For each step the we need to remove an element and push an element on the heap. the cost of doing them are all O(logk). and this step need to be repeated n times (until we iterated all elements), so the total running time is n*O(logk)= **O(nlogk).**

4. **Radix Sort numbers (using base 10) in O(n+k)**
   Solution:

   Algorithm Description:
   This algorithm consists of two steps.

   - Using counting sort to put numbers into different bucket according to their length. Since the length of each number is limited, we can count the lengths of numbers in O(n) using an external array. and numbers with the same length are put in one bucket. (note: the largest possible integer is 2,147,483,647, which has a length of 10, so we can use an array with size of 10 to count length of each numbers. for long integers, we just need to use a bigger array, with the same time complexity)
   - suppose the number of elements in the buckets are k1, k2, k3 ..., km. in each bucket we just need to compare k_i times to determine the order in each bucket (from MSB to LSB), where k_i is the length of numbers in bucket i. so in total, we need O(k1+ k2 + ,,, + km) = O(k)

   Time complexity:

   - Step 1 cost O(n) and Step 2 costs O(k), so overall time complexity is O(n+k).

5.**Sort sequence S with n/logn subsequence and each subsequence contains logn elements**

Solution:

Let k be the number of subsequence, and i be the number of elements in each subsequence. k = n/logn, i = logn

- We consider orders of elements of sequence S as states. That is, different order of sequence S represents a different state. the input order can be in any order, and the desired order is the only one order where all elements are aligned in ascending order. we complete the sorting through a series of steps of comparisons. each comparison decides on the next state of the sequence S. Such a process can be perceived as a decision making tree, where the root node is the initial state of the sequence and each branching is one comparion.
- For each subsequence, there exists i * (i-1) * (i-2)* ... * 2 * 1 permutations of the sebsequence.
- There are k such subsequences, so in total there are (i!)^k permutations of sequence S.
- Thus, the decision tree for sorting S may have at least (i!)^k leaves (because there can be leaves in the same state (order), when that happens, we have more than (i!)^k leaves).
- Therefore, the height of the tree H is the cost of the sort, which is log(# of leaves), H has a lower bound of log((i!)^k).
- log((i!)^k) > log((i * (i-1) * (i/2))^k) > log(((i/2)^i/2)^k) = (k*i/2)log(i/2), we know that k = n/logn, i = logn. so we have the lower bound of H is (n/2)*log(logn/2). remove the constant from the result, we have the lower bound of the number of comparisons is **nloglogn**