CS 141, Fall 2017 Homework 4

Posted: October 23th, 2017 Due: October 30th, 2017

Name: Zhenxiao Qi

Student ID #: 500654348

- You are expected to work on this assignment on your own
- Use pseudocode, Python-like or English to describe your algorithms. Absolutely no C++/C/Java
- When designing an algorithm, you are allowed to use any algorithm or data structure we explained in class, without giving its details, unless the question specifically requires that you give such details
- Always remember to analyze the time complexity of your algorithms
- Homework has to be submitted electronically on Gradescope by the deadline. No late assignments will be accepted

Problem 1. (25 points)

The *Hadamard* matrices $H_0, H_1, H_2, \ldots$ are defined as follows.

$$H_k = \begin{cases} [1] & k = 0 \\ \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right] & k > 0 \end{cases}$$

Note that H_k is a $2^k \times 2^k$ matrix. Design a $O(n \log n)$ divide-and-conquer algorithm that given a column vector v of length $n = 2^k$, computes the matrix-vector product $H_k v$. Analyze the time complexity of your algorithm.

Answer:

$H_k \cdot v = [a, b]^T$

$a = H_{k-1}v_1 + H_{k-1}v_2$

$b = H_{k-1}v_1 - H_{k-1}v_2$

we compute $H_K \cdot V$ with 2 multiplications of size $n/2$ array and matrix, and then add(or minus) them. the adding process takes $O(n)$

The running time is given by $T(n) = 2T(n/2) + cn$

So the time complexity is $O(n \log n)$ by master theorem method.

The algorithm is as follow:

```
def multiply(H[][],v[]):
  H[][]=H[:n/2][:n/2]
  v1[]=v[:n/2]
  v2[]=v[n/2:]
  x=multiply(H[][],v1[])
  y=multiply(H[][],v2[])
  a=x+y
  b=x-y
  result[]=zip([a,b]) #transpose
  return result
```

Problem 2. (25 points)

You are given two sorted list of size m and n. Give a $O(\log k)$ time algorithm for computing the k-th smallest element in the union of the two lists. **Note:** Observe that the k-th smallest element in the union of the arrays $a[1 \ldots m]$ and $b[1 \ldots n]$ has to be contained in $a[1 \ldots k]$ or $b[1 \ldots k]$.

Answer:

$\because$ the algorithm should be $O(\log K)$

$\therefore$ we use binary search based on K, which means we divide k by 2 until it becomes 1

Firstly, we compare the $(k/2)th$ elements in array a and b.

if $(a[k/2] == b[k/2])$

then a[k/2] is the k_{th} element

if $(a[k/2] < b[k/2])$

Drop the fist $k/2$ elements in array a, $k = k/2$, $drop+ = k/2$

if $(a[k/2] > b[k/2])$

Drop the first $k/2$ elements in array b, $k = k/2$, $drop+ = k/2$

Then we compare the k_{th} (k has been divided by 2) elements in the remain arrays as the steps above, and drop the elements before the smaller one

why we drop the elements? Because the elements before the smaller $(k/2)_{th}$ element are definitely before the k_{th} element.

At the mean time, we count the drop,when we drop k-1 elements, the next element is the k_{th} element in the union array.

Based on the analysis above, the time complexity is $O(\log k)$

Problem 3. (25 points)

Describe and analyze an algorithm that takes an unsorted array A of n integers (in an unbounded range) and an integer k, and divides A into k equal-sized groups, such that the integers in the first group are lower than the integers in the second group, and the integers in the second group are lower than the integers in the third group, and so on. For instance if $A = \{4, 12, 3, 8, 7, 9, 10, 20, 5\}$ and $k = 3$, one possible solution would be $A_1 = \{4, 3, 5\}, A_2 = \{8, 7, 9\}, A_3 = \{12, 10, 20\}$. Sorting A in $O(n \log n)$-time would solve the problem, but we want a faster solution. The running time of your solution should be bounded by $O(nk)$. For simplicity, you can assume that is n a multiple of k, and that all the elements are distict. **Note:** k is an input to the algorithm, not a fixed constant.

Answer:

We can use linear selection for k times

First, we find the k_{th} elements in A, and then we call linear selection recursively to find the K_{th} element in the right part.

Finally we find all the groups in k times linear selection

def **Select**(A,start,end,k)

1. divide the array A into small groups of size 5
2. find the median of each group by sorting the group and pick the middle one
3. call **Select** recursively to find the median of the medians

4 partition, then split it into 2 arrays L(elements smaller than median),R(elements bigger than median)

5. $k = |L| + 1$

if (i==k) call **Select** to find K_{th} elements in R,return L

if ($i < k$) **Select**(L,i)

else **Select**(R,i-k)

The running time is bounded by $O(nk)$ for we call linear selection for k times

Problem 4. (25 points)

Given an array of numbers $X = \{x_1, x_2, \ldots, x_n\}$, an *exchanged pair* in X is a pair (x_i, x_j) such that $i < j$ and $x_i > x_j$. Note that an element x_i can be part of up to $n-1$ exchanged pairs, and that the maximal possible number of exchanged pairs in X is $n(n-1)/2$, which is achieved if the array is sorted in descending order. Develop a divide-and-conquer algorithm that counts the number of exchanged pairs in X in $O(n \log n)$ time. Argue why your algorithm is correct, and why your algorithm takes $O(n \log n)$ time. You can assume that n is a power of two.

Answer:

We can modify Mergesort algorithm to solve this problem.

In the mergesort tree, the index of the element left part array is always smaller than that of the element in the right part.

So we only need to find the element in the left part which is greater than one element in the right part.

So in the merge function of Mergesort, if $L[i] > R[j]$, count++.

the algorithm is as follow:

```
def func(A):
  if len(A)<2
    return A
  else
    m=len(A)/2
    L=func(A[:m])
    R=func(A[m:])
    reutrn FindPair(L,R)

def FindPair(L,R):
  i,j,count=0, 0,0
  while i<len(L) and j<len(R):
    if L[i]>R[j]:
      count+=1
      i+=1
    else :
      j+=1

  return count
```