ANLY550 Homework2

Hongyang Zheng

Problem 1

First, give an O(nlogk) algorithm to merge k sorted lists with n total elements into one sorted list

Assume there is an empty list with size n which is used to store the results.

Step 1: Take the first element from each sorted list and use them to build a heap using Build_Max_Heap function.

Step 2: Extract the minimum value from the heap using extract_min function(which is like extract_max, but returns and deletes the minimum value from the heap) and append it on the result list.

Step 3: Take the next element from the list where the minimum value in step 2 comes from, insert it to the heap in the correct place and then go back to step 2 (This process will continue until all lists become empty).

Step 4: Put the values left in the heap into the list using extract_min function.

Correctness: The initial max-heap tree has the largest number in the total lists. Everytime we extract $H[1]$ (which is always the largest number in the remaining lists) and insert the exact successor of it in that list into H, we correct the tree. Therefore, the output is from the largest number to the smallest number.

Analysis of time: Since there are k elements used to build the initial heap in step 1, step 1 will take $O(k)$ and there are $n-k$ elements left. For extract_min function and insert function, both will take $O(logk)$ for each time. Since there are $n-k$ elements left, we will use insert for $n-k$ times and use extract_min for $n-k+k=n$ times. Therefore, the total time is $O(k-klogk+2nlogk)=O(nlogk)$.

Second, give an O(nlogk) algorithm for sorting a list of n numbers that is k-close to sorted.

Assume there is an empty list with size n which is used to store the results.

Step 1: Take the first $k+1$ elements from the list and use them to build a heap using Build_Max_Heap function.

Step 2: Extract the minimum value from the heap using extract_min function and append it on the result list.

Step 3: Take an element from the list and insert it to the heap in the correct place and then go back to step 2 (this step will continue until the list is empty).

Step 4: Put the values left in the heap into the result list using extract_min function.

Analysis: Since there are $k+1$ elements used to build the initial heap in step 1, step 1 will take $O(k+1)$ and there are $n-k-1$ elements left. For extract_min function and insert function, both will take $O(logk)$ for each time. Since there are $n-k-1$ elements left, we will use insert for $n-k-1$ times and use extract_min for n times. Therefore, the total time is $O(k+1-klogk-logk+2nlogk)=O(nlogk)$.

Problem 2

(a)

Assume there are two numbers X and Y, which can be represented by

$$X = a * 10^{2n/3} + b * 10^{n/3} + c$$
$$Y = d * 10^{2n/3} + e * 10^{n/3} + f$$

Therefore:

$$X * Y = ad * 10^{4n/3} + ae * 10^{n} + af * 10^{2n/3} + bd * 10^{n} + be * 10^{2n/3} + bf * 10^{n/3} + cd * 10^{2n/3} + ce * 1$$

$$X * Y = ad * 10^{4n/3} + (ae + bd) * 10^{n} + (af + be + cd) * 10^{2n/3} + (bf + ce) * 10^{n/3} + cf$$

After observation, I got:

$$ae + bd = (a + b)(e + d) - ad - be$$
$$bf + ce = (b + c)(f + e) - be - cf$$
$$af + cd = (a + c)(f + d) - ad - cf$$

Let:

$$part1 = ad$$
$$part2 = be$$
$$part3 = cf$$
$$part4 = (a + b)(e + d) - ad - be = ae + bd$$
$$part5 = (b + c)(f + e) - be - cf = bf + ce$$
$$part6 = (a + c)(f + d) - ad - cf = af + cd$$

Therefore, $X * Y$ becomes

$$part1 * 10^{4n/3} + part4 * 10^{n} + (part6 + part2) * 10^{2n/3} + part5 * 10^{n/3} + part3$$

where each part represents one multiplication. As a result, this method only uses six multiplications.

Here is the algorithm:

Step 1: Represent the two numbers like

$$X = a * 10^{2n/3} + b * 10^{n/3} + c$$
$$Y = d * 10^{2n/3} + e * 10^{n/3} + f$$

Step 2: Calculate part1 - part6 like above formula

Step 3: Shift each part properly (multiply 10^k is like to shift the number to the left k position) and add them to get

$$part1 * 10^{4n/3} + part4 * 10^{n} + (part6 + part2) * 10^{2n/3} + part5 * 10^{n/3} + part3$$

(b)

Determine the asymptotic running time of your algorithm

Assume that the multiplication of two n-digit numbers takes time $T(n)$, and we know the time for adding numbers or shifting numbers can be represented by $O(n)$.

Therefore, the total time is: $T(n) = 6 * T(n/3) + O(n)$. According to master theorem, $a = 6, b = 3, k = 1$, $a > b^k$, $T(n) = O(n^{log_3 6}) = O(n^{1.63})$

Compared to Karatsuba's algorithm which uses $T(n) = 3T(n/2) + O(n) = O(n^{1.59})$, I would split it into two parts, because it is faster.

c)

Determine the asymptotic running time of an algorithm using five multiplications

Assume that the multiplication of two n-digit numbers takes time $T(n)$, and we know the time for adding numbers or shifting numbers can be represented by $O(n)$.

Therefore, the total time is: $T(n) = 5 * T(n/3) + O(n)$. According to master theorem, $a = 5, b = 3, k = 1$, $a > b^k$, $T(n) = O(n^{log_3 5}) = O(n^{1.46})$, which is faster than Karatsuba's algorithm with $O(n^{1.59})$. Therefore, I would split it into three parts in this case.

Problem 3

Algorithm1

Step1: Make a copy of array A, which is called array B and using mergesort to sort array A.

Step2: Take $A[0]$ which is the smallest number and find its index using binary search in array B and call it x. Since $A[0]$ ($B[x]$) is the smallest number, the numbers before $B[x]$ are all bigger than it, which can form $(x - 1)$ pairs of inversion. Let count $c = x - 1$.

Step3: Take next element in array A, find its index x using binary search in array B.

Step4: For i from $0 \rightarrow x - 1$, compare $B[i]$ and $B[x]$, if $B[i] > B[x]$, then increment c by 1. Return to step 3 until reach the end of the list.

Analysis for time

The mergesort in step 1 costs $O(nlogn)$. Since there are total n elements in array A, this algorithm will use n times binary search and each binary search uses $O(logn)$. Therefore, the total time is $O(nlogn) + O(n * logn) = O(nlogn)$.

Algorithm2

MergeSort is the function which divides the array into two subarrays every time until subarrays(left and right) only have 0 or 1 element. Then from the single element, call the function merge which is the function to sort and merge the sorted array. Since we want to count the inversions, we need to add a counter to the mergesort function that counts the numebr of inversions everytime call merge function to sort two sorted arrays and sum up those inversions.

Since we suppose the number in left list should all be smaller than the number in right list, which means if we find $left[0]$ is bigger than $right[0]$, then for this single value, it will produce size(left) inversions since $left[1, 2, \ldots, mid - 1] > right[0]$, else no inversion will appear in the single position. We should sum up inversions everytime we compare $left[0]$ and $right[0]$.

Pseudocode

```
let inversion_n = 0
mergeSort(a):
    if size(a) <= 1:
        mid = len(a) // 2
    left = mergeSort(a[:mid])
    right = mergeSort(a[mid+1:])
    return merge(left, right)

merge(left,right):
final=list()
while left is not empty and right is not empty:
    if left[0]<=right[0]:
        final.append(left.pop(0))
    else:
        final.append(right.pop(0))
        inversion_n = inversion_n + size(left)
return final+left+right
```

Analysis for time

This algorithm should have the same runtime as mergesort since we only add a counter into the function. Therefore, the runtime is $O(nlogn)$.

Problem 4

Prove that if a graph G is undirected, then any depth first search of G will never encounter a cross edge.

Assume graph G is undirected, and there is a cross edge (v, w). Since cross edge is these go from “right to left”– there is no ancestral relation (from lecture3), it means w has been visited and all edges going from w have been processed too, which means the edge (w, v) is also been processed. Since v has not been visited at that time, (w, v) is a tree edge and (v, w) must be a back edge. Therefore, there is a contradiction for the assumption. As a result, if a graph G is undirected, then any depth first search of G will never encounter a cross edge.

Problem 5

Describe an algorithm

In this algorithm, we use two arrays $dist[v]$ and $prev[v]$ and a priority queue H of capacity n.

1. $dist[v]$, will eventually contain the smallest bottleneck from s to v. We initialize it to 0 for s and to $-\infty$ for all other vertex.
2. $prev[v]$, will eventually contain the shortest path from s to v (all previous node).
3. $H = \{(s, 0), (v, -\infty) : v \in V, v \neq s\}$.

When H is not empty, everytime we extract the minimum of H (suppose it is v) and find the egde of it. For every $(v, w) \in E$, we compare the $bottle[w]$ to the maximum of $(bottle[v], length(v, w))$. If $bottle[w]$ is greater than the maximum of $(bottle[v], length(v, w))$, we update it to the maximum of $(bottle[v], length(v, w))$.

Pseudocode

smallest-bottleneck(G = (V,E,length); s ∈ V):

```
bottle[s] = 0
bottle[v] = ∞ for all v except s
prev[v] = null for all v ∈ V
H: priority heap of V
H = {s : 0},(v: ∞) where v ∈ V except s

while H is not empty:
   v = deletemin(H)

   for each (v,w) ∈  E:
     if bottle[w] > max(bottle[v],length(v,w)):
        bottle[w] = max(bottle[v],length(v,w))
        prev[w] = v
        insert(w,bottle[w],H)
```

Analysis of time

Since we only modify the if statement in the for loop and it does not influence the runtime, this algorithm has the same complexity as Dijkstra's algorithm. Therefore, the runtime is $O((|V| + |E|) * logn)$.

Problem 6

We modify the algorithm some parts:

1. we keep an priority heap H of size |V|m, the entries of which are subsets of V .
2. inserting (w, dist[w]) works by adding w to H[dist[w]].
3. deletemin works by keeping a counter into H that starts at 0 and on each call to deletemin increments itself repeatedly until it points to a nonempty entry H[i], and returns the first element of H[i].

Pseudocode

algorithm (G = (V,E,length); s ∈ V):

```
dist[s] = 0
dist[v] = ∞ for all v except s
prev[v] = null for all( v ∈ V )
H: priority heap of V of size |V|m
H = {s : 0},(v: ∞) where v ∈ V except s

while H is not empty:
   v = deletemin(H)

   for (v,w) ∈  E:
     if dist[w] > dist[v]+length(v,w):
         dist[w] = dist[v]+length(v,w)
         prev[w] = v
         insert(w,dist[w],H)
```

Correctness

Since the values in dist are always the lengths of paths of at most $|V| - 1$ edges in G and no such path is longer than $|V|m$, H won't overflow, so the deletemin operation works correctly and this algorithm is correct.

Analysis for time

We know that the runtime for Dijkstra's algorithm is: $O(|E| * insert - time + |V| * delete - min - time)$. For the above algorithm, each insert takes time $O(1)$. After initializing all of the vertices, we scan the buckets from 0 to $|V - 1|m$. When a non-empty bucket is encountered, the first vertex is removed, and all adjacent vertices are relaxed. This step is repeated until we have reached the end of the queue: this process takes $O(|V|m)$. Therefore, the total time is $O(|E| + |V|m)$

Problem 7

Describe an algorithm

This question is like Bellman-Ford algorithm we use in graph to find single source shortest path as well as to detect if there exist any negative cycle in the graph. Hence in the similar manner, starting from any source vertex, we will run the algorithm analogous Bellman-Ford algorithm, on encounter of directed edge (u, v) and the label of vertex v will be updated based on value of u:

$$\text{if } l[v] < l[u] * r_{u,v} \text{ then } l[v] = l[u] * r_{u,v}$$

Thus from source vertex s, which we label with value 1, this algorithm will give label of every vertex v equal to $l[v]$ which is the largest possible value we can get from exchanging of 1 unit of currency s with the currency in v by following the path in the tree returned by algorithm from vertex s to vertex v.

Finally at the end of algorithm if for every edge (u, v), we have $l[v] \geq l[u] * r_{u,v}$ holds then this means there is no risk-free path which is equivalent to non-existence of negative weighted cycle in single-source-shortest-path problem. And if we find any edge (u, v) with $l[v] < l[u] * r_{u,v}$ this means there is some cycle through which currency exchange can take place which will increase the value of currency.

So we will represent this problem using graph $G = (V, E)$ where each vertex represent some currency c_i and directed edge (u, v) will have weight equal to $r_{u,v}$ which indicate the currency exchange rate from currency u to currency v.

Pseudocode

```
risk-free-algorithm(G=(V,E), length=r1, r2, ..., ri):

l[s]=1
l[v]=0 for all vertex in V except s
prev[v]=NULL for all vertex in V

for i = 1, 2, ..., n-1:
    for edge (u,v) in E:
            if l[v] < l[u]*r{u,v}:
               then l[v] = l[u]*r{u,v}
               prev[v]=u

for each edge (u,v) in E:
    if l[v] < l[u]*r{u,v}
        then return ("Risk-free cycle exists")
```

Analysis for time

Above algorithm runs with time complexity = $O(|V||E|)$. Since the graph is complete graph as the exchange rate is available for every currency, therefore $|E| = O(|V|^2)$. And since $|V| = n$, therefore the time complexity is $O(n^3)$.