Yunjia Zeng
yz682

Homework 2

ANLY 550
March 11, 2019

Problem 1

1. **Part 1**

   We solve this problem by using a min heap with k elements to keep the smallest elements from all lists at each round, so that we track the smallest number from all lists that we are going to merge. We assume the output array is in ascending order. The min heap is just the opposite as the max heap that we've been introduced in class: parent nodes are smaller than or equal to their children. We then continuing:

   (a) swapping the root, which is the smallest in the heap, with the last node
   (b) removing the last node from the heap
   (c) add the next element in the list of unmerged lists to the root of the heap
   (d) rearrange the heap by performing build-min-heap since the newly added element may violate the min-heap property

   until all the elements of all lists are added in our result array. Like what we've mentioned, and since the lists that are being merged are all sorted, we take out the smallest - i.e. the first entry of each list repeatedly. This takes $k-1$ comparisons per element, therefore in total the time complexity is $O(k \cdot n)$. Since each time we will remove the minimal nodes, i.e. the root of the min heap and then we swap the root of the reordered heap with the last node remove it, lastly add it to the result array. It takes $O(k)$ time to build the heap; for every element we need $O(1)$ time to swap and remove the root node; and we perform build-min-heap to reorder the heap, which takes $O(n \log k)$ time since reversed version of max-heapify - min-heapify also takes $O(\log k)$ each time and we call it in build-min-heap recursively from 1 to k. Therefore, in total it takes $O(k + n \log k + 1) = O(n \log k)$ time.

Algorithm 1: min-heapify

```
l = left(i)
r = right(i)
if (l >= heap-size(A) and A[l] < A[i]):
    smallest = l
else:
    smallest = i
if (r >= heap-size(A) and A[r] < A[smallest]):
    then smallest = r
if smallest != i:
    exchange A[i] and A[smallest]
    min-heapify(A, smallest)
```

Algorithm 2: build-min-heap

```
for index i = n/2 down to 1:
    min-heapify(A, i)
```

Part 2

For this problem, we assume the result array is in ascending order, and we're using the property of min heap. We first create a min heap of size $k+1$ with first $k+1$ elements in the k-close sorted array. The time complexity of this process is $O(k)$. Then we remove the minimal element one-by-one in the min heap, and put then in the result array. Meanwhile, we add new element to the min heap from the remaining elements of the k-close sorted array. Since removing and adding a new element to the min heap takes $O(\log k)$ time, therefore the time complexity in total is $O(k + (n-k) \cdot \log k) = O(n \log k)$.

Problem 2

1. **(a)** Suppose there are two n-digits number x and y, we represent x and y by

$$x = x_0 \cdot 10^{2n/3} + x_1 \cdot 10^{n/3} + x_2$$
$$y = y_0 \cdot 10^{2n/3} + y_1 \cdot 10^{n/3} + y_2$$

Therefore, we compute $x \cdot y$ as

$$x \cdot y = x_0 y_0 \cdot 10^{4n/3} + (x_0 y_1 + x_1 y_0) \cdot 10^n + (x_0 y_2 + x_1 y_1 + x_2 y_1) \cdot 10^{2n/3} + (x_1 y_2 + x_2 y_1) \cdot 10^{n/3} + x_2 y_2$$

Then we can use x_0y_0, x_1y_1, and x_2y_2 to calculate as follows:

$$\begin{aligned} x_0y_1 + x_1y_0 &= (x_0 + x_1) \cdot (y_0 + y_1) - x_0y_0 - x_1y_1 \\ x_0y_2 + x_1y_1 + x_2y_1 &= (x_0 + x_2) \cdot (y_0 + y_2) - x_0y_0 - x_2y_2 + x_1y_1 \\ x_1y_2 + x_2y_1 &= (x_1 + x_2) \cdot (y_1 + y_2) - x_1y_1 - x_2y_2 \end{aligned}$$

Now we can recursively solve the multiplication problem by divide the problem to six sub-problems and combine them at last. For total it takes six multiplications of $n/3$ numbers and hence the running time is

$$T(n) = 6T(\frac{n}{3}) + O(n)$$

2. **(b)** As stated above, the running time of this algorithm is

$$T(n) = 6T(\frac{n}{3}) + O(n).$$

By consider the equation in general form, assuming the parameter in the recurrence is greater than one, we have

$$\begin{aligned}T(n) &= p \cdot T(\frac{n}{3}) + O(n)\\ &= p^{\log_3 n}\\ &= n^{\log_3 p}.\end{aligned}$$

Therefore, when we have 6 multiplication, we have $T(n) = n^{\log_3 6} \approx n^{1.63}$, which is less efficient than Karatsuba's algorithm that with $T(n) = n^{1.59}$. Thus it would be better to just split the n-digit number to two smaller parts as what we will do by using Karatsuba's algorithm.

3. **(c)** By defining new equations we can make the algorithm with only 5 multiplications:

$$\begin{aligned}W_0 &= x_0y_0\\ W_1 &= (x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2)\\ W_2 &= (x_0 + 2x_1 + 4x_2) \cdot (y_0 + 2y_1 + 4y_2)\\ W_3 &= (x_0 - x_1 + x_2) \cdot (y_0 - y_1 + y_2)\\ W_4 &= (x_0 - 2x_1 + 4x_2) \cdot (y_0 - 2y_1 + 4y_2)\end{aligned}$$

since

$$\begin{aligned}x_0y_0 &= W_0\\ 12(x_1y_0 + x_0y_1) &= 8W_1 - W_2 - 8W_3 + W_4\\ 24(x_2y_0 + x_1y_1 + x_0y_2) &= -30W_0 + 16W_1 - W_2 + 16W_3 - W_4\\ 12(x_2y_1 + x_1y_2) &= -2W_1 + W_2 + 2W_3 - W_4\\ 24x_2y_2 &= 6W_0 - 4W_1 + W_2 - 4W_3 + W_4\end{aligned}$$

so that we can successfully substitute the terms in the original equation in part (a). Similarly, the run time is

$$T(n) = 5T(\frac{3}{n}) + O(n) = n^{\log_3 5} \approx n^{1.47}.$$

Now this algorithm is a little bit more efficient than Karatsuba's algorithm, which with $T(n) = n^{1.59}$, therefore it is better to split the two n-digits numbers into 3.

4. **(d)**

Problem 3

We can count the number of inversions by an enhanced merge sort algorithm. Suppose we have an array A, and we split it into two parts - $lefthalf$, with length l, and $righthalf$, with length r. Let length of the array A be n, and therefore if n is an odd number we have $l = r + 1$; if n is an even number we have $l = r$. The idea is that in merge sort, we split the array to half, and recursively sort the two half arrays and then merge. Therefore, we will know the inversions in the left half, the right half and also the inversions in the merging step. The total number of inversions is to add the number of inversions in the those three parts. One thing that we need to pay attention is, during merge process, we shall only count the inversions that i is in the left half of the subarray, and j is in the right half of the subarray, such that $lefthalf[i] > righthalf[j]$ and there are $l - i$ inversions. This is because both left and right subarrays are sorted in the first and second recursive calls, therefore all the remaining elements in $lefthalf$ - i.e. $lefthalf[i + 1], lefthalf[i + 2], \ldots, lefthalf[l]$, are greater than $righthalf[j]$.

Algorithm 3: CountInv

```
mcount = 0
lcount = 0
rcount = 0
if n > 1:
    mid = n // 2
    lefthalf = A[:mid]
    righthalf = A[mid:]
    lcount = CountInv(lefthalf)
    rcount = CountInv(righthalf)
    i = 0
    j = 0
    while i < l and j < r:
        if lefthalf[i] < righthalf[j]:
            i = i + 1
        else:
            j = j + 1
            mcount += length(lefthalf[i:])
    while i < l:
        i = i + 1
    while j < r:
        j = j + 1
return lcount + rcount + mcount
```

Problem 4

Proof. Suppose there exists a cross edge $(v, w) \in G$, where G is a undirected graph. By definition, there is no ancestral relationship in the DFS tree between the two endpoints of the edge. Therefore, w has already been visited and all outgoing edges of w have been processed and left set S. Furthermore, since the graph G is undirected, edge (w, v) has also been processed. But meanwhile vertex v has not been visited and by definition, (w, v) is a tree edge since the edge connecting a parent to a child in the DFS tree. Then (v, w) becomes a back edge by definition since w lies on the edge go from the root to v, which is an ancestor in the DFS tree. Thus it contradict with our assumption initially and we conclude that any DFS search of G will never encounter a cross edge by contrudition. $\square$

Problem 5

For this problem, instead of thinking about the so-called longest edge, we think of the graph as a weighted graph and the edge weight is like the width of the edge. This is very similar as the example that illustrated in class - to consider the edge weights as the length of the edge, therefore an edge with weight 5 will be considered as an edge with length 5.

We can modified Dijkstra's algorithm to solve this bottleneck problem. The finding smallest bottleneck are actually the finding the path with minimum max-weight. Therefore we now maintain the bottleneck rather than the shortest path length to a vertex. The original version of the Dijkstra's algorithm is from the class note by Dr. Thaler.

Algorithm 4: find-min-bottleneck

```
H = a priority queue of capacity n
dist[s] = 0
dist[v] = +inf for all v != s
prev[v] = NULL for all v in V
while H is not None:
    v = delete-min(H)
    for each (v, w) in E:
        if dist[w] > max(dist[v] + length(v, w)):
            dist[w] = max(dist[v] + length(v, w))
            prev[w] = v
            insert(H, w, dist[w])
```

The algorithm maintains two arrays of length n, where $dist[v]$ will eventually store $d(s, v)$ and $prev[v]$ will eventually store the predecessor of v on the minimum max-weight path from s to v - i.e. the smallest bottleneck path. Initially we have $H = \{(s, 0), (v, \infty) : v \in V, v \neq s\}$. By implement the priority queue H as a binary heap, each time an edge that has been added to the heap if and only if its source has just been assigned its distance. This ensures that each edge is added to the heap for at most once. Since the while loop definitely needs n times, and each step does constant time except for the last insert step, we have the runtime as $O(|V| \cdot delete - min - time + |E| \cdot insert - min - time = O((|E| + |V|) \cdot \log |V|)$.

Problem 6

For this problem, since the delete-min (from class notes) removes the pair on vertices with the smallest non-decreasing priority, each time we insert a new value $dist[v] + length(v, w)$ into the heap, it is greater than the minimum value $dist[v]$ that we will delete at the beginning of the step in which we consider the vertex v. Since we implement the binary priority heap by keeping an array H of size $|V| \cdot M$, the entries of it are the subsets of V, therefore inserting $(w, dist[w])$ is to add w to $H[dist[w]]$, and delete-min keeps a counter into H that starts at 0 and on each call to delete-min increments itself repeatedly until it points to a nonempty entry $H[i]$, and returns the first element of $H[i]$.

Since the values in both $dist$ are always the lengths of paths of at most $|V| - 1$ edges in G and no such path is greater than $|V| \cdot M$, H would not overflow. The delete-min works correctly by the observation at the beginning. Thus, each insert step cost time $O(1)$, with $H[i]$ implemented by binary heap, and the total time for all delete-min steps is at most $O(|E| + |V| \cdot M)$, since we traverse each position of H at most once, and each vertex in V at most once.

Problem 7

We consider the the sequence of exchange rate $r_{i_1,i_2}, r_{i_2,i_3}, \dots, r_{i_{k-1},i_k}, r_{k,1}$ as the non-negative path that to be determine by applying Bellman-Ford algorithm. Suppose we have a directed weighted graph $G = (V, E)$, such that each vertex of G represent a type of currencies, and the directed edges are the exchange rates. Therefore, for each pair of currencies c_i, c_j, there are directed edge pair (r_i, r_j) and (r_j, r_i). We then noticed that the total number of vertices are $|V| = n$ and the total number of edges are $|E| = \binom{n}{2}$.

Since finding

$$r_{i_1,i_2} \cdot r_{i_2,i_3} \dots r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$$

is equivalent to find

$$\frac{1}{r_{i_1,i_2}} \cdot \frac{1}{r_{i_2,i_3}} \dots \frac{1}{r_{i_{k-1},i_k}} \cdot \frac{1}{r_{k,i_1}} < 1,$$

by taking log on both sides, we have

$$\log \frac{1}{r_{i_1,i_2}} \cdot \log \frac{1}{r_{i_2,i_3}} \dots \log \frac{1}{r_{i_{k-1},i_k}} \cdot \log \frac{1}{r_{i_k,i_1}} < \log 1 \implies 0.$$

Now the problem becomes finding whether there exist a negative path in the graph, which means if G contains a path satisfied the condition (a negative weight cycle), then when Bellman-Ford terminates, there exist an $v \in V$ such that $dist[v] \neq d(s, v)$. We can determine whether there is negative weight cycle in G by adding an extra vertex v^* with 0 weight edges that connects to all other vertices. We then run Bellman-Ford from this v^* and output the result opposite to the original output of the Bellman-Ford - i.e. Boolean output true if there are no negative weight cycles and false if there is a negative weight cycle.

Adding this new dummy vertex v^* does not introduce any new cycle to G, therefore the algorithm ensures all negative weight cycles are reachable from v^*. According to the class notes, since there are for total n currencies that are connected back and force, therefore we need $O(n \cdot n) = O(n^2)$ time to construct G. Since the runtime of Bellman-Ford is descided by the total number of edges and total number of vertices, therefore the runtime is $O(n^3)$.