problem_set2

March 11, 2019

1. Explain how to solve the following two problems using heaps. First, give an O(nlogk) algorithm to merge k sorted lists with n total elements into one sorted list. ‘

Given k sorted input arrays A_1, A_2, ..., A_k, where the sum of the length of the two arrays is n. Create a heap B of size k, where each element in B is a single element from each of the k input ararys. Furthermore, each element is taken from the last most entry of each input array. This ensures that B will always be made up of the largest possible values out of all input arrays. Call Build_Max_Heap on B to ensure that the heap statisfies the max heap property. This will run in $O(k)$ time. Next remove the root node from B and insert to results array. Add new number from the same array that the element in the root node came from. This ensures that each element in B comes from each input array, which ensures that the max most value from each array is in the heap. This is necessary for proper sorting. This process is repeated until the heap becomes empty, and each element that is inserted into results is inserted backwards from n, n-1, n-2, ..., 1. When an input array becomes empty, negative infinity is inserted into B, and the algorithm continues as normal.

Pseudocode:
Input: k arrays $[A_1, A_2, \ldots, A_k]$ with total length equal to n.
Output: Single sorted array.

results $\leftarrow$ [] // Initialize results into an empty array that will contain the sorted elements.
lists $\leftarrow [A_1, A_2, ..., A_k]$ B $\leftarrow [a_1, a_2, \ldots, a_k]$ // B is formed by taking the last element from $[A_1, A_2, \ldots, A_k]$. It can be shown that this would take $O(1)$ time by using indexing.
Build_max_heap(B) // Runs in $O(k)$ time.

while $B \neq \emptyset$ {
$i, v \leftarrow$ Extract_max(B) // runs in $O(logk)$ time. Extract_max maintains the max heap property. i is the index of lists, which corresponds to the input array that the value v came from.

new_element $\leftarrow$ lists[i][length(lists[i])] // Runs in O(1) time. This indexes for the last element in the same array as v.

results[n] $\leftarrow v$ // adds the value to the end of results.

$n = n - 1$

if length(lists[i]) = 0 {
insert(B, $-\infty$) // insert also maintains the max heap property. Runtime is O(logk).
}
else{
insert(B, new_element)
}}

The while loop runs until B is empty, which means it will run n times because there are total n elements. That means that the total runtime is $O(k) + nO(logk) + nO(1) + nO(logk) = O(k + nlogk + n)$. Clearly, $nO(logk)$ dominates. Therefore, runtime is $O(nlogk)$.

Proof of correctness by loop invariant:

The i^{th} element to insert into result is always the root node in heap B in step i because that has the highest value in the heap. Therefore, the results array is correctly sorted when every element is inserted backwards from index n down to index 1. At a given instance, B always contain the max values from each input array. Therefore, the root of B is the max value possible out of all the input arrays, not including the values already in results.

Initialization: Show invariant is true before loop started.

Before the 1^{st} iteration of the while loop, B always contain the max values from each input list. Therefore, the root of B is the max value, which is the correct element to insert into results. Therefore, the algorithm will correctly sort this value.

Maintenance: Show it is true after an iteration.

After 1^{st} iteration, the max element is inserted into results. This means B contains the 2^{nd} highest value possible, and it is at the root node. Since algorithm removes element from root to add to results, this element will be correctly sorted into results.

Therefore after i^{th} iterations, the i^{th} element is correctly inserted into results[n...i...1].

Termination:

The code will terminate when $B \neq \emptyset$, which means input arrays are now empty. Therefore, all elements must be in results. Since algorithm correctly sorts at each iteration, the final output must be correctly sorted. QED.

1. Second, say that a list of numbers is k-close to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an O(nlogk) algorithm for sorting a list of n numbers that is k-close to sorted.

Given an input array A of length n with numbers that are k-close to sorted. For every value that is k-close to sorted, create a binary heap, B, of size $k+1$. B is made up of the last $k+1$ elements from A. This ensure that B will always contain the max value from A. Initialize an empty array called results. Call Build_Max_Heap on B once. This runs in O(k). Now remove the root node from B and insert it into results array. Remove the value in the last position of A and insert it into B. This is repeated n times, and at each iteration, the value is inserted into results array going backwards, from the last index of results to the first.

Pseudocode:

Input: A, unsorted and k-close to sorted. A is length n. $A[1, ..., n]$.
Output: return a sorted array called results.

results $\leftarrow$ [] // Initialize results into an empty array that will contain the sorted elements.
B $\leftarrow A[k-1, k-2, ..., n]$ // B is formed by taking the last $k+1$ elements from A. It can be shown that this would take $O(1)$ time by using indexing.
Build_max_heap(B) // Runs in $O(k)$ time.

while $B \neq \emptyset$ {
$v \leftarrow$ Extract_max(B) // runs in $O(logk)$ time. Extract_max maintains the max heap property. i is the index of lists, which corresponds to the input array that the value v came from.

results[n] $\leftarrow v$ // adds the value to the end of results.

$n = n - 1$

insert(B, $A[n]$) // insert also maintains the max heap property. Runtime is O(logk).
}

The while loop runs n times. Therefore runtime is O(nlogk) + O(n) + O(k). The dominant term is O(nlogk); therefore, the overall runtime is O(nlogk).

Proof of correctness by loop invariant:

At i^{th} step, the max value of B will be the correct element to insert to results going from last index to first index of the array. In other words, the i^{th} element to be inserted into results is found in B in the i^{th} step. This implies that results will be correctly sorted when the i^{th} element is inserted into results from index $n...i$.

Initialization: Show invariant is true before loop started.

Before the first step, B contains the max value in A because the max value must be k positions away from correct sorted position, and B contains $k+1$ elements starting with values from the end of A. This ensures that B will include the max value. Therefore, the correct value will be inserted into results, which is found in the root of B. The root of B is the max most value.

Maintenance: Show it is true after an iteration.

After first iteration, the max most value is found in results. Because B always contain the current max value of A, we can be sure that the correct value will be inserted into results by the algorithm.

Therefore, after i^{th} iteration the i^{th} element will be correctly inserted to results.

Termination: Show termination of loop results in desired outcome.

It follows that after n iterations, loop terminates and the results array contains all the sorted elements in correct order. QED.

2. Consider an algorithm for integer multiplication of two n-digit numbers where each number is split into 3 parts, each with n/3 digits. Design and explain such an algorithm, similar to the integer multiplication algorithm (karatsuba's algo) presented in class. Your algorithm should describe how to multiply the two integers using only 6 multiplication on the smaller parts instead of the straight forward 9.

Given 2 integers X and Y. Each integer has length n and n is divisible by 3. Let $X = a10^{2n/3} + b10^{n/3} + c$ and $Y = d10^{2n/3} + e10^{n/3} + f$. Then the multi algorithm will make 6 recursive calls on inputs of size $n/3$. The base case is when the two inputs are length 1. At this recursive level, the algorithm will do simple integer multiplication because the input is small enough where the runtime will be $O(1)$ for this step. Finally, the algorithm will use the results of the 6 recursive calls to calculate the product of X and Y.

Pseudocode:

multi(X,Y){

Input: 2 positive integers X and Y where the length of each integer is n. Assume n is divisible by 3.

Output: Returns $X \cdot Y$.

if X or Y are length 1{

return product of X and Y // Integer multiplication on sufficiently small numbers has runtime O(1) time.

}

$X = a10^{2n/3} + b10^{n/3} + c$

$Y = d10^{2n/3} + e10^{n/3} + f$

temp1 $\leftarrow$ multi(a,d)

temp2 $\leftarrow$ multi($a+b, e+d$)

temp3 $\leftarrow$ multi(b, e)

temp4 $\leftarrow$ multi($c+b, e+f$)

temp5 $\leftarrow$ multi(c, f)

temp6 $\leftarrow$ multi($f+d, a+c$)

return $(temp1)10^{4n/3} + [temp2 - temp1 - temp3]10^n + [temp4 - temp5 - temp3]10^{n/3} + [temp6 - temp5 - temp3]10^{2n/3} + (temp5)$ }

Proof of correctness by induction:

Input: 2 positive integers X and Y of length n.

Output: Product of X and Y.

Base case:

Suppose the length of X and Y is 1. Then $X \cdot Y$ is returned, which is the desired outcome.

Inductive Case:

Assume n is divisible by 3 and n is length of X and Y. Let $X = a10^{2n/3} + b10^{n/3} + c$ and $Y = d10^{2n/3} + e10^{n/3} + f$. Assume that algorithm returns correct integer multiplication on input of size smaller than n. Suppose we call algorithm on inputs of X and Y with length n. Therefore, the 6 recursive calls made by the algorithm will return the correct product because the recursive calls are on inputs of size $n/3$ and the inductive hypothesis guarentees that this will return correctly. Since it is a fact that $X \cdot Y = (ad)10^{4n/3} + (ae + db)10^n + (fa + be + dc)10^{2n/3} + (ce + bf)10^{n/3} + (cf)$ and $(temp1)10^{4n/3} + [temp2 - temp1 - temp3]10^n + [temp4 - temp5 - temp3]10^{n/3} + [temp6 - temp5 - temp3]10^{2n/3} + (temp5)$ is equal to this, the final return call by the algorithm returns the correct product. QED

2. Determine the asymptotic running time of your algorithm. Would you rather split it into two parts(with 3 multiplications on the smaller parts) as in karatsuba's algorithm?

There are 6 recursive calls on input of $n/3$. Adding 2 n-digit numbers has $O(n)$ runtime. Multiplication by powers of 10 is O(n) runtime. Therefore, the post processing is $O(n) + O(n) = O(n)$. Let $T(n)$ represent the number of steps. Then, $T(n) = 6T(n/3) + O(n) = 6T(n/3) + cn$. Let $a = 6$, $b = 3$, $k = 1$. Since $6 > 3$, it follows from the master theorem that the runtime is upper bounded by $O(n^{log_3 6}) = O(n^{1.63})$. Since karatsuba's algorithm runs in $O(n^{1.58})$, karatsuba's algorithm is slightly faster, so it would be better to split it into two parts.

2. Suppose you could use only 5 multiplications instead of 6. Then determine the asymptotic running time of such an algorithm. In this case, would you rather split it into 2 parts or 3 parts?

$T(n) = 5T(n/3) + O(n) = 6T(n/3) + cn$. Let $a = 5$, $b = 3$, $k = 1$. Since $5 > 3$, the runtime of this algorithm would be $O(n^{log_3 5}) = O(n^{1.46})$. In this case, splitting into 3 parts would be faster than karatsuba's algorithm.

3. An inversion in an array A[1, . . . , n] is a pair of indices (i,j) such that i < j and A[i] > A[j]. The number of inversions in an n-element array is between 0 (if array is sorted) and n(n-1)/2 (if an array is sorted backwards). Describe and analyze an algorithm to count the n umber of inversions in an n-element array in O(nlogn) time. Modify mergesort.

Given an array $A[1, 2, ..., n]$ and n is the length. The algorithm does a recursive call on the first half of A and another recursive call on the second half of A. These recursive calls return a value for the number of inversions and the sorted array. The merge subroutine is called to merge the two halves of A into 1 sorted array. At the same time, the number of inversions between the two halves are calculated. The algorithm then returns the sum of all inversions from the 3 parts.

Pseudocode:

Input is array $A[1, 2, ..., n]$. The length of A is n. Output is number of inversions.

Counter(A){

if $n = 1$\{ // Base case.
return (0, A)
\} else\{
// recursively call counter on 1st half of A and 2nd half of A.
$(X, x) \leftarrow Counter(A[1, 2, ..., n/2])$ // X is the number of inversions in 1st half of A and x is the sorted array for first half.
$(Y, y) \leftarrow Counter(A[n/2 + 1, n/2 + 2, ..., n])$ // Y is the number of inversions in 2nd half of A and y is the sorted array for second half.
$(Z, z) \leftarrow merge(x, y)$ // Z is the number of inversions between 1st half of A and 2nd half of A. z is the sorted array.
return $(X + Y + Z)$
\}
merge(A, B)\{
// A and B are sorted arrays.
// if $A[i] > B[j]$, then $A[i, i + 1, ..., n/2] > B[j]$. This means the number of inversions on $B[j]$ is equal to the length of $A[i, i + 1, ..., n/2]$.
count $\leftarrow 0$
c $\leftarrow$ []
n $\leftarrow 1$
while length(A) > 0 or length(B) > 0\{
if $A[0] > B[0]$\{
remove $B[0]$ from B
$c[n] \leftarrow B[0]$
$n \leftarrow n + 1$
count $\leftarrow$ count + length(A)
\} if $A[0] < B[0]$\{
remove $A[0]$ from A
$c[n] \leftarrow A[0]$
$n \leftarrow n + 1$
\}
if length(A) $= 0$\{
add the rest of B to c
\}
if length(B) $= 0$\{
add the rest of A to c
\}
return (count,c)
\}\}

The runtime or merge subroutine is $O(n)$ because the while loop occurs atmost $n/2 + 1$ times and all operations inside the loop is constant time. There are 2 recursive calls in the Counter algorithm. Therefore, the runtime should be the same as merge sort. Runtime is $O(nlogn)$.

Counter algorithm proof of correctness by induction:

Input is array $A[1, 2, ..., n]$ is an unsorted array of length n.
Output is the number of inversions in A.

Base case:
Suppose $n = 1$. Then, there are no inversions. Therefore, return 0 is the correct output.

Inductive case:
Let X represent first $1/2$ of A, and Y represent second $1/2$ of A. Assume length of A is divisible

by 2. Assume that the Counter algorithm returns corrrect number of inversions for input size less than n. Suppose we call algorithm on input of size n. Then algorithm will recursively call on X and Y. By the inductive hypothesis, these calls will return the correct number of inversions. Since the merge step works correctly, then the algorithm returns correct number of inversions after summing up all the inversions from the 2 recursive calls and the 1 merge step. QED

Merge subroutine proof of correctness:

The Merge subroutine uses the following property: if $A[0] > B[0]$, then $A[0, 1, ..., n/2] > B[0]$. This means the number of inversions on $B[0]$ is equal to the length of $A[0, 1, ..., n/2]$. At each iteration of the while loop, count is only increased when the property is satisfied. The while loop runs until A and B are empty, which occurs atmost $(n/2 + 1)$ times. After the loop terminates, the counter will successfully sum up all the iterations between the two halves. QED

4. Recall that when running depth first search on a directed graph, we classified edges into 4 categories: tree edges, forward edges, back edges, and cross edges. Prove that if a graph is undirected, then any depth first search on G will never encounter a cross edge.

Proof by contradiction:

Assume that {u,v} is a cross edge. That means DFS has visited both u and v, but the edge {u,v} is not marked as a tree edge. Since it was not marked as a tree edge, it means that the edge {u,v} was not visited. Suppose DFS visited u first and not v. DFS would explore all edges connected to u. Therefore, it would of explored u and then explore the edge {u,v}. This means this edge is a tree edge. This is a contradiction, which means that {u, v} cannot be a cross edge. Both edges can't be visited at the same time and once either u or v is visited, {u,v} will be explored because the edge is undirected. QED.

5. In the shortest-path algorithm, we are concerned with the total length of the path between a source and every other node. Suppose instead that we are concerned with the length of the longest edge between the source and every node. That is, the bottleneck of a path is defined to be the length of the longest edgte in the path. Design an efficient algorithm to solve the single source smallest bottleneck problem. (find the paths from a source to every other node such that each path has the smallest possible bottleneck)

The proposed algorithm is simply a change on the optimization condition of Dijkstra's algorithm. Instead of creating paths that have the smallest distance to v from s, the proposed algorithm create paths that have the smallest bottleneck. An array called max is used to keep track of the max edge weight in a path. For example, $max[e]$ would return the max edge weight in the path from s to e. If a new path is found where the max edge weight to v is smaller than the path currently found, the algorithm will modify the dist array, max array, and prev array to match the new found path. Everything else, is the same as Dijkstra's algorithm.

Pseudocode:

Input is $G =$ Graph(E,V), $length[1, 2, ..., n]$, and source s such that $s \in V$. $length[e]$ is the weight of edge e.

Output is path to v from s such that bottleneck is minimized.

$H \leftarrow \{(s, 0), (v, \infty) : v \in V, v \neq s\}$ // H is a priority heap with size n where $n = |V|$.

$dist[s] \leftarrow 0$

$dist[v] \leftarrow \infty$ for all $v \neq s$ // runtime is $O(n)$ because there are n vertices.

$prev[v] \leftarrow \emptyset$ for all $v \in V$ // runtime is $O(n)$ because there are n vertices.

while $H \neq \emptyset$\{ $v \leftarrow$ delete_min(H) // delete_min gets the minimum value from H. It maintains minimum heap property. Runtime is $O(logn)$.

For each edge $(v, w) \in E${
if $max[w] >$ max_value(max[w], length(v, w)){ // max_value returns the max value out of the two inputs.
$max[w] > $ max_value(max[w], length(v, w))
$prev[w] \leftarrow v$
$dist[w] \leftarrow dist[w] + length(v, w)$ $Insert(H, w, dist[w])$ // Maintains minimum heap property. Runtime is $O(logn)$ } } }

Because this algorithm only differs from Dijkstra's algorithm on the optimization condition, the runtime of this algorithm is the same as Dijkstra's algorithm. Therefore, the runtime is $O((m + n)logn)$ where m is the number of edges and n is the number of vertices.

Proof of correctness:
Assume Dijkstra's algorithm is correct. The proposed algorithm is a modification on Dijkstra's algorithm on the optimization condition. Instead of minimizing the total length of a path to v, this algorithm minimizes the bottleneck of a path to v. Therefore, it simply follows from Dijkstra's algorithm that the algorithm will correctly output paths that minimize the max edge weight in a path.

6. Consider the shortest paths problem in the special case where all edge costs are non-negative integers. Describe a modification of Dijkstra's algorithm that works in time O(|E| + |V|M), where M is the maximum cost of any edge in the graph.

Let H be an array of length $k|V|$. $k|V|$ represents the maximum possible distance from source. k is the max edge weight in the graph $G = (V, E)$. An index i in H is empty if there are no vertices that has distance i from s. Each index starting from index 0 represents the distance from s, and each index i can hold more than 1 vertex. Initially, s is set to index 0 and every other vertice is set to ∞. The Insert subroutine is implemented such a way that it inserts a vertex into a specified index i in the array H. Therefore, the runtime on Insert is O(1), constant time. The Delete_min subroutine is implemented such that it starts from index 0 in H and delete empty entries from list until encounter a vertex. Each deletion until encounter vertex will be constant amount of steps and varies for each iteration. A dist array is initialized to keep track of all the distances from s to v. A prev array is used to keep track of the node that came before node v. A vertex is retrieved from H and every adjacent edge of v is explored. Lets say there is an edge (v, w) from v that connects to w. If the path to w is shorter, then update that path to the shortest distance and update prev such that the previous node to w is updated to reflect the new path. Repeat this process until every edge and vertice is visited.

Pseudocode:

Input is Graph $G = (V, E)$, $length[1, 2, ..., n]$, source $s \in V$. $length[v]$ is the weight of edge v. V is the set of all vertices, and E is the set of all edges.
Output is distance to every reachable v from s.

Let H match the description used above.
$dist[s] \leftarrow 0$
$dist[v] \leftarrow \infty$ for all $v \neq s$
$prev[v] \leftarrow \emptyset$ for all $v \in V$
while $H \neq \emptyset${
$v \leftarrow$ Delete_min(H)
For each edge $(v, w) \in E${
if $dist[w] > dist[v] + length(v, w)${
$dist[w] \leftarrow dist[v] + length(v, w)$

$prev[w] \leftarrow v$
$Insert(H, w, dist[w])$ // Runtime is $O(1)$
} } }

The Insert subroutine is implemented such a way that it inserts a vertex into a specified index i in the array H. Therefore, the runtime on Insert is O(1), constant time. The Delete_min subroutine is implemented such that it starts from index 0 in H and delete empty entries from list until encounter a vertex. Each deletion until encounter vertex will take a constant amount of steps and varies for each iteration. However, we know that each vertice is visited 1 time and the total length of H is $k|V|$. Therefore, we know that the total amount of steps that Delete_min takes after the algorithm terminates is $k|V|$. Therefore, Delete_min has a total runtime of $O(k|V|)$ when algorithm terminates. Formally, let $X_1, X_2, ..., X_{k|V|}$ be the number of steps of Delete_min for each iteration of the while loop. It is true that $\sum_{i=1}^{k|V|} X_i = k|V|$ and each Insert call has runtime $O(1)$.

$[O(1) + O(1), ... + O(1)] + [X_1, X_2, ..., X_{k|V|}] = |E|O(1) + k|V| = O(|E| + k|V|)$

Proof of correctness:

Assume that Dijkstra's algorithm is correct. The only change in the proposed algorithm is that the implemented data structure is different. The Insert subroutine inserts vertices into index that correspond to the distance of vertex v from s. This ensures that vertices are ordered from smallest distance to highest distance like in a binary heap. The Delete_min operation always return minimum vertex distance, which is the same as in Dijkstra's implementation. Therefore, it follows that the proposed algorithm is correct. QED

7. The risk-free currency exchange problem offers a risk-free way to make money. Suppose we have currencies $c_1, c_2, ...c_n$. For every two currencies c_i and c_j, there is an exchange rate $r_{i,j}$ such that you can exchange one unit of c_i for $r_{i,j}$ units of c_j. Note that if $r_{i,j} \cdot r_{j,i} > 1$, then you can make money simply by trading units of currency i into units of currency j and back again. This almost never happens, but occasionally (because the updates for exchange rates do not happen quickly enough) for very short periods of time exchange traders can find a sequence of trades that can make risk-free money. That is, if there is a sequence of currencies $c_{i_1}, c_{i_2}, ..., c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot ... \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$, then trading one unit of c_{i_1} into c_{i_2} and trading that into c_{i_3} and so on will yield a profit. Design an efficient algorithm to detect if a risk-free currency exchange exists. (Need not actually find the path, just Yes or No)

Have $c_1, c_2, ...c_n$ make up the vertices of a graph where $r_{c_1,c_2}, r_{c_2,c_3}, ..., r_{c_{n-1},c_n}$ are edge weights for corresponding edges. Do a $-log(x)$ transformation on all edge weights, and then call Bellman_ford on the graph with transformed edge weights.

Pseudocode:

Input: Graph $G = (V, E)$, where $c_1, c_2, ...c_n$ make up the vertices. Let the following $r_{c_1,c_2}, r_{c_2,c_3}, ..., r_{c_{n-1},c_n}$ be edge weights for the corresponding edge, and let length be the array that contains the edge weights. So $length[(c_1, c_2)]$ contains the edge weight r_{c_1,c_2}.
Output: Determines if there is a risk-free currency exchange or no there isn't.

for edge $\in$ E{
$length[edge] \leftarrow -log(length[edge])$ // Doing a negative log transformation for every edge. Runtime is $O(m)$ because there are m edges.
}

Bellman_Ford($G = (V, E)$, length) // Bellman_Ford will output 'Yes' if a negative cycle, risk-free currency exchange, exists

The runtime is $O(|E|) + O(|E||V|) = O(|E||V|)$ because clearly $|E||V|$ term dominates. The Bellman_ford algorithm takes up the most time.

Proof of correctness:

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

$= log(r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot ... \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1}) > log(1)$

$= log(r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot ... \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1}) > 0$

$= -log(r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot ... \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1}) < 0$

$= -log(r_{i_1,i_2}) - log(r_{i_2,i_3}) ... - log(r_{i_{k-1},i_k}) - log(r_{i_k,i_1}) < 0$

Assume Bellman_ford algorithm is correct. Clearly, the proposed algorithm is correct because it only does a $-log(x)$ transformation on all the edge weights and then calls Bellman_ford algorithm. Transforming every edge weight allows us turn the condition for risk-free currency exchange into a condition that also matches the condition for negative weight cycles. Therefore, Bellman_Ford algorithm is able to detect risk-free curency exchange. QED