CSCE629 Analysis of Algorithms

Homework 3

Texas A&M U, Fall 2019 *09/13/19*

Lecturer: Fang Song *Due: 10am, 09/20/19*

Instructions.

- Typeset your submission by LaTeX, and submit in PDF format. Your solutions will be graded on *correctness* and *clarity*. You should only submit work that you believe to be correct, and you will get significantly more partial credit if you clearly identify the gap(s) in your solution. You may opt for the "Ill take 15%" option (details in Syllabus).
- You may collaborate with others on this problem set. However, you must **write up your own solutions** and **list your collaborators and any external sources** for each problem. Be ready to explain your solutions orally to a course staff if asked.
- For problems that require you to provide an algorithm, you must give a precise description of the algorithm, together with a proof of correctness and an analysis of its running time. You may use algorithms from class as subroutines. You may also use any facts that we proved in class or from the book.

This assignment contains 4 questions, 10 pages for the total of 60 points and 10 bonus points. A random subset of the problems will be graded.

1. (Akinator's trick) Play the game Akinator online (`https://en.akinator.com/`), and answer the questions below.
   (a) (10 points) Given a *sorted* array A with distinct numbers, we want to find out an i such that $A[i] = i$ if exists. Give an $O(\log n)$ algorithm.
   (b) (10 points) Consider a sorted array with distinct numbers. It is then rotated k (k is unknown) positions to the right, and call the resulting array A. (Example: (8,9,2,3,5,7) is the sorted array (2,3,5,7,8,9) rotated to the right by 2 positions) Design as efficient an algorithm as you can to find out if A contains a number x.

   Exercise (do not turn in). Can you think of some real-world problems that the techniques in your algorithms could be useful?

Solution:

(a) **Binary Search Algorithm:** We are given an array $A[1, \ldots, n]$ such that it has n distinct numbers arranged in a way such that $A[1] < A[2] < \ldots < A[n]$.

Suppose we define a new array $B[1, \ldots, n]$ by setting $B[i] = A[i] - i$ for all i. For every index i we have,

$$B[i] = A[i] - i \leq (A[i+1] - 1) - i = A[i+1] - (i+1) = B[i+1],$$

hence array $B[i]$ is arranged in such a way that it is in non-decreasing order and when $B[i] = 0$, it means that $A[i] = i$. So we can find an index i such that $A[i] = i$ by performing a binary search in B. We don't actually need to compute Bin advance; instead, whenever the binary search needs to access some value $B[i]$, we can just compute $A[i] - i$ on the fly instead.

Here are two formulations of the resulting algorithm, first recursive (keeping the array A as a global variable), and second iterative.

Alogrithm 1 *Recursive Binary Search* on $A[1, \ldots, n]$
Return an index i such that $l \leq i \leq r$ and $A[i] = i$
BinarySearch(l, r):
if $l > r$
  return NONE
mid $\leftarrow (l + r)/2$
if $A[\text{mid}] = \text{mid}$
  return mid
else if $A[\text{mid}] < \text{mid}$
  return BinarySearch$(mid + 1, r)$
else $A[\text{mid}] > \text{mid}$
  return BinarySearch$(l, mid - 1)$

Alogrithm 2 *Iterative Binary Search* on $A[1, \ldots, n]$
Return an index i such that $l \leq i \leq r$ and $A[i] = i$
BinarySearch$A[1, \ldots, n]$:
$r \leftarrow n$
$l \leftarrow 1$
while $l \leq r$
  mid $\leftarrow (l + r)/2$
  if $A[\text{mid}] = \text{mid}$
    return mid
  else if $A[\text{mid}] < \text{mid}$
    $l \leftarrow mid + 1$
  else $A[\text{mid}] > \text{mid}$
    $r = mid - 1$
return NONE

In both the formulations, the algorithm is binary search. Worst case running time of binary search is $O(\log n)$ where n is the size of the array. Assume

if that the array did not contain any element which satisfied this property $A[i] = i$, then this will lead to worst case possible for binary search. We know that the problem is being split into two halves every time we split hence the maximum possible no. of splits will be $n/2^i = 1 \implies i = \log_2 n$. At each level we compare/check the middle element once, hence at worst case we end with $O(\log_2 n)$ running time.

(b) To solve this problem we have to observe a key point that if we apply binary search or just split the given array into halves then we observe that at least one half of the arrays will be sorted. If the pivot element about which the array was rotated happens to be the middle one then when we split the array we will get two halves which will be sorted.

Example: Let $A = [4, 5, 6, 7, 8, 9, 1, 2, 3]$, then mid element is 8 and left half is sorted but right half is not.

Example: Let $A = [6, 7, 8, 9, 1, 2, 3, 4, 5]$, then mid element is 1 and both left and right halves are sorted.

Hence the interesting property of a sorted plus rotated array is that when you divide it into two halves, **at least one** of the two halves will always be sorted. We can know which half is sorted by comparing the two extreme elements of that half. Let L be the sorted half and R be the other half. Then we can just compare our given value with the extreme elements of L to know if the elements lies in the sorted array L. If it doesn't lie in L then we discard it and search in the other half i.e., R. We can keep doing this until we find the value. Since we are discarding one half each time this algorithm we take same running time as binary search i.e., $O(\log n)$.

Note: This algorithm works only for array with distinct elements. If the array contains elements that repeat then we need to more modifications.

Alogrithm 3 *Modified Binary Search* on $A[1, \ldots, n]$
Return an index i such that $l \leq i \leq r$ and $A[i] = x$, target value (key)
ModBinarySearch(l, r):
if $l > r$
  return NONE
mid $\leftarrow (l + r)/2$
if $A[\text{mid}] = \text{mid}$
  return mid
if $A[l] \leq A[mid]$
  if $A[l] \leq x$ **and** $A[mid] \geq x$
    return ModBinarySearch$(l, mid - 1)$
  return ModBinarySearch$(mid + 1, r)$

else
if $A[mid] \leq x$ **and** $A[r] \geq x$
return ModBinarySearch$(mid + 1, r)$
return ModBinarySearch$(l, mid - 1)$

One of the applications of the binary search is in optimization and reinforcement learning (and other numerical methods) where we need to do root finding. We can apply bisection method on functions which have opposite signs on extremum values. Another application is in mathematical optimization of packing problems. The goal is to either pack a single container as densely as possible or pack all objects using as few containers as possible. Many of these problems can be related to real life packaging, storage and transportation issues. Each packing problem has a dual covering problem, which asks how many of the same objects are required to completely cover every region of the container, where objects are allowed to overlap. Other applications can be

- Voting theory
- Collaborative filtering
- Measuring the sortedness of an array
- Sensitivity analysis of Googles ranking function
- Rank aggregation for meta-searching on the Web
- Nonparametric statistics (e.g., Kendalls tau distance).

2. (Counting inversions) Given a sequence of n *distinct* numbers $a_1, \ldots, a_n$, we call (a_i, a_j) an *inversion* if $i < j$ but $a_i > a_j$. For instance, the sequence $(2, 4, 1, 3, 5)$ contains three inversions $(2, 1)$, $(4, 1)$ and $(4, 3)$.

   (a) (15 points) Given an algorithm running in time $O(n \log n)$ that counts the number of inversions. (Hint: does Merge-sort help?) Can you also output all inversions?

   (b) (10 points (bonus)) Let's call a pair a *significant inversion* if $i < j$ and $a_i > 2a_j$. Given an $O(n \log n)$ algorithm to count the number of significant inversions.

Solution:

(a) **Modified Merge-sort:** We will use merge-sort and modify it such a way that it helps us in counting all possible inversions. This basically tell us how close we are from sorted array in some notion. Idea is that if we keep track inversion in left and right sub array which are sorted, then we can compute

the no. of inversions required, while doing the merge step. Finally adding these three values will give us total no.of inversions present in the given array. In merge step itself we can have a additional print step that will give us all the inversion present in the graph.

Hence the idea for modified merge-sort is that sort each half sub array during the recursive call, then count inversions while merging the two sorted lists (merge-and-count). In terms of Divide-and-Conquer algorithm this will be

- **Divide:** separate the list into two halves A and B.
- **Conquer:** recursively count inversions in each list.
- **Combine:** count inversions (a, b) with $a \in A$ and $b \in B$.
- Return sum of three counts.

The modified step here is the *combine* step. While combining we will count the inversions in the following way,

- Assume $a \in A$ and $b \in B$ and A and B are sorted.
- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, then a_i is not inverted with any element left in B.
- If $a_i > b_j$, then b_i is not inverted with any element left in A.
- Keep track of all the inversions and append smaller element to sorted list C.

Algorithm 4 *Modified Merge-sort* on list of elements L
Return number of inversions in L and L in sorted order
Sort-and-Count(L, l, r):
if List L has one element
  return $(0, L)$
{Divide list into two halves A and B}
$mid \leftarrow (l + r)/2$
$A \leftarrow L[l, mid]$
$B \leftarrow L[mid + 1, r]$
$(r_A, A) \leftarrow Sort - and - Count(A, l, mid)$
$(r_B, B) \leftarrow Sort - and - Count(B, mid + 1, r)$
$(r_{AB}, L) \leftarrow Merge - and - Count(A, B, l, mid, r)$
return $(r_A + r_B + r_{AB}, L)$

Merge-and-count(A, B, l, mid, r)
{Assume A and B are sorted}
$p = l$; $q = mid + 1$; $idx = l$; $r_{AB} = 0$

while $p \leq mid$ and $q \leq r$
if $A[p] \leq B[q]$
{Doesn't form an inversion pair}
$L[idx] = A[p]$
$p = p + 1$
else
{Yes we have inversion pairs count them and print them}
$L[idx] = B[q]$
$q = q + 1$
$r_{AB} = r_{AB} + (mid - p + 1)$
for $i = p$ to mid
print *Inversion pair* $B[q]$, $A[i]$
$idx = idx + 1$
if $p > mid$
{Copy all the leftover elements from B to L}
for $i = q$ to r
$L[idx] = B[i]$
$idx = idx + 1$
else
{Copy all the leftover elements from A to L}
for $i = p$ to mid
$L[idx] = A[i]$
$idx = idx + 1$
return L and r_{AB}

Summary of the $Merge - and - Count$ will be first check if there is any kind of inversion by comparing elements from A and B. If yes then keep track of them and print all the inversion pairs and merge the elements into L. If no then just merge the elements L. Finally merge all the left over terms and return the count and list/array L.

Using the Divide-and-Conquer approach we can write the recurrence relation. We need constant time in dividing the problem in to half i.e., $O(1)$. When we combine we will be merging at most n elements hence at each step combining will take $O(n)$ time. At each step we divide the problem to 2 sub problems which are half the size of the original problem. Final equation will be

$$T(n) = \begin{cases} \Theta(1) & if n = 1 \\ 2T(n/2) + O(n) & otherwise \end{cases}$$

Using master theorem we get $T(n) = O(n \log n)$.

(b) The algorithm for counting *significant inversion* will be similar to for counting the *inversion* except for small modification to $Merge - and - Count$. The difference is that we will split the merge step into two steps. First step will be for normal merging and sorting the array. Second step will be for counting the significant inversion in array $B^{'}$, which is constructed as $B^{'}[i] = 2B[i]$. Below I show the pseudo code only for the modified part,

Merge-and-count(A, B, l, mid, r)
{Assume A and B are sorted}
$p = l$; $q = mid + 1$; $idx = l$; $r_{AB} = 0$
while $p \leq mid$ and $q \leq r$
 if $A[p] \geq 2 * B[q]$
{Yes we have inversion pairs count them and print them}
  $q = q + 1$
  $r_{AB} = r_{AB} + (mid - p + 1)$
  for $i = p$ to mid
   print *Inversion pair* $B[q]$, $A[i]$
 else
{Doesn't form an inversion pair}
  $p = p + 1$
{Reset counters and do normal merge}
$p \leftarrow l$
$q \leftarrow mid + 1$
while $p \leq mid$ and $q \leq r$
 if $A[p] \leq B[q]$
  $L[idx] = A[p]$
  $p = p + 1$
 else
  $L[idx] = B[q]$
  $q = q + 1 \quad idx = idx + 1$ **if** $p > mid$
{Copy all the leftover elements from B to L}
 for $i = q$ to r
  $L[idx]$ = $B[i]$
  $idx = idx + 1$
else
{Copy all the leftover elements from A to L}
 for $i = p$ to mid
  $L[idx]$ = $A[i]$
  $idx = idx + 1$
return L and r_{AB}

This algorithm is just modified form of Merge-sort and we can apply analysis that we did above as it is to conclude that running time of this algorithm will be $O(n \log n)$.

3. (10 points) (Cycles) Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one (not all cycles, just one of them). The running time of your algorithm should be $O(m+n)$ for a graph with n nodes and m edges.

Solution: Without loss of generality, assume the graph is connected (if not, do the following for each connected component).

Pick an arbitrary vertex as root and do a DFS, while maintaining a DFS tree. The DFS tree initially contains the root and at each point, when you encounter an edge to a new vertex, the new vertex is added to the tree and the edge is added between the new vertex and the parent. If at any point you encounter an edge from a node to a seen vertex, then there is a cycle in the graph, and you can return the cycle by tracing the path from each end point of the latest edge backwards in the DFS tree till they meet (linear time), and displaying it in the correct order. If the DFS ends without incident, there is no cycle. DFS runs in linear time on the edges/vertices, as each edge/vertex is visited only once.

Note: In an undirected graph, the edge to the parent of a node should not be counted as a back edge, but finding any other already visited vertex will indicate a back edge (source).

4. (15 points) (Shortest cycles containing a given edge) Give an algorithm that takes as input an undirected graph $G = (V, E)$ and an edge $e \in E$, and outputs a shortest cycle that contains e (if no cycle containing e exists, the algorithm should output "none").

(Note: Give as efficient an algorithm as you can.)

Solution: **Special property of BFS:** This problem is about finding shortest cycle in graph $G = (V, E)$ containing an edge $e \in E$. If no cycle containing e exists the algorithm should output NONE.

Our main idea behind solving this problem is that the shortest cycle containing an given edge (which means that we are given two fixed points/vertices/nodes on the graph) will be the shortest path possible between two the vertices plus the edge. Let edge $e = (a, b)$, joining vertices a, b, then shortest cycle S will be,

$S(e) = S(a, b)$ = Shortest path between (a, b) + Edge e.

Now our problem has been simplified to finding the shortest path between the given vertices a, b joined by the edge e. To solve the problem of finding shortest path can be done using BFS (Chapter 4 of Algorithms by Dasgupta).

Let an edge joining two vertices u and v be $e = (u, v) \in E$. Now we need to find a shortest cycle containing it. First we remove the edge e from the given graph $G = (V, E)$. After removing the edge e from $G = (V, E)$ we get new graph $G^{'} = (V, E \setminus e)$. Now we run the BFS algorithm on the new graph $G^{'} = (V, E \setminus e)$ starting from u and find the shortest path from u to v. If we find such path between u and v then the shortest cycle will be shortest path plus the edge that we removed. The total distance of this cycle will be $d(u, v) + 1$, where $d(u, v)$ is length of the shortest path between u and v and additional one is for the edge e between u and v (assuming the graph has unit weights for all edges). If we fail to find an shortest path between u and v then the edge e is the only way to go from vertex u to vertex v; which means that there is no shortest cycle containing edge e.

The assumption of all edges weights equal to one is important because for connected undirected graph G the algorithm for computing the shortest path will vary based upon the assumption on the weights of the edges (source). We might need to use different algorithms like Dijkstra's algorithm (if weights are non-negative), Bellman-Ford algorithm (if weights can be negative). But luckily in the case of undirected graph with all edge weights equal to one, the shortest path trees coincide with breath-first search (BFS) trees. We can state the theorem for shortest path tree for BFS algorithm as follows,

Theorem: The BFS algorithm

- visits all and only nodes reachable from source node s,
- for all nodes v sets $d(v)$ to the shortest path distance from s to v,
- sets parent variables to form a shortest path tree.

The above theorem has been taken from Chapter 8 of Algorithms by Jeff Erickson.

Alogrithm 5 *Modified BFS* on $G = (V, E)$ and edge $e = (a, b) \in E$
Return the shortest cycle S such that it contains e and $min(S_i(e))$ i.e., distance wise minimum of all possible cycles
ModBFS(G, e):
Construct Graph $G^{'} = (V, E \setminus e)$ (graph G with edge e removed)
parent, d = BFS($G^{'}$, a)
if $parent[b] = nil$

return NONE (*Graph G has no shortest cycle containing edge e*)
print *Shortest cycle starting from b to a*
print *Node b to*
$temp \leftarrow b$
while $parent[temp]$!= s
print *Node* $parent[temp]$ *to*
$temp \leftarrow parent[temp]$
print *Finally edge e which connects the node a and b*
print *Total distance we need to travel is* $d[b] + 1$

Do BFS on the given graph G and source node s
BFS(G, s):
Mark all nodes in $v \in V$ as unvisited
$parent[v] \leftarrow nil$
$d[v] = \infty, \forall v \in V$
Mark source node s as visited
$parent[s] = s$
$d[s] = 0$
Let Q be a queue, $Q.enqueue(s)$
while Q *not empty*
u = $Q.dequeue$
for all neighbours v of u in graph G
if v is not visited
Mark v as visited
$parent[v] = u$
$d[v] = d[u] + 1$
return $parent$, d

Running time for this algorithm will $O(n + m)$ where n is total no. of nodes in the graph and m is total no. of edges in the graph, since the major computation time will be spent on doing the BFS on the graph $G^{'}$. Hence running time for finding the shortest cycle for given edge e in graph G will be linear in time.