CS170–Spring 2017 — Homework 2Solutions

Michael Fan, SID 26697596

Collaborators: Julie Han, Erik Yang

1.

1. Overview: We sort rectangles left to right and keep track of the points at which rectangles begin or end with max heap. Array "arr" is input as a triple of (leftCoord, rightCoord, height)

2. Pseudocode:
   function outline(arr): intersections = minHeap of all l_i and r_i in arr mapped to rectangles
   activeRects = maxHeap of rectangles existing at current point sorted by height
   for each intersection:
   if intersection is in set of l_i:
   add rectangle to activeRects
   if intersection is in set of r_i:
   remove rectangle from activeRects
   if the top element of activeHeap has changed:
   add intersection to output
   return output

3. Proof of correctness:
   Height of the skyline changes when buildings are introduced or removed. Output blocks are placed right after a height change. Thus, because every start/end of rectangles is checked, we will not have missed anything and the output will be complete.

4. Runtime Analysis:
   Updating heaps takes O(n$logn$) time. Looping through the intersections takes O(n) time. Thus, checking each intersection takes O(n) time. Our algorithm is dominated by heap updates which takes O(n$logn$) time.

2.

1. Overview:
   function "majority" will take in 2 input arrays "og" and "mod". "og" will hold the original array in which we want to determine if there is a majority element and "mod" will contain a modified array that we use for recursion. "majority" will output either the majority element if there is one or "NO MAJORITY" if there is none. Starting from index 0, we pair up each consecutive couple of element in "mod". If they are different, we throw them both away. If they are the same, we append one copy of the element into the "halved" array that we will then run this function recursively on. In the case of a odd number of elements in "mod", the last element will be unpaired. We count the number of times this element appears in "og" to test if it is a majority element. If it is, we are done. If it is not, we throw it away and recurse on "halved." In this way, we reduce the number of elements left to consider by half with every recursive call and checking if a particular element is a majority element is Θ(n)

2. Pseudocode:

```
function majority(og, mod):
    halved = []
    for i=0; i < mod.length-1; i+=2:
        if mod[i] == mod[i+1]
            halved.append(mod[i])
    if mod.length is odd
        if count(og, mod[mod.length - 1]) >= (og.length + 1)/2
            return mod[mod.length - 1]
    return majority(og, halved)
function count(arr, elem):
    c = 0
    for i in arr:
        if i == elem:
            c++
    return c
```

3. Proof of correctness:
   In the case that the input list has a length that is a power of 2, we will simply keep eliminating elements until there is one element left. This last element will either be majority element if there is one or it won't be the majority element if there is not one. This is correct because given an array of n elements, the majority element has to occur at least (n + 1)/2 times. For every majority element we delete, there is a non-majority element deleted as well. Hence, the majority element will survive until the end in all cases. The odd case is clearly correct as well as it is a modification of the even case that includes an extra check for the extra element.

4. Runtime analysis:
   We model this into a recurrence problem:

$$T(n) = T(n/2) + O(n)$$

   We reduce the problem by half with every recursive call, and there is a max of 1 call to function "count" that iterates through a list in linear time. Via Master's Theorem, we get that this algorithm is $O(nlogn)$.

3.

1. Overview:
   Given an input array "arr" of n elements and an integer t, we encode the n numbers of "arr" as the exponents of a length n polynomial of the form $x^{arr[n-1]}+...+x^{arr[0]}$. We then cube this polynomial using FFT and check if t is in the power coefficients of the resultant polynomial returned as an array named "poly".

2. Pseudocode:
   function tripleSum(arr, t):
   poly = FFT(FFT(arr, arr), arr)
   for int i in poly:
   if i == t:
   return true
   return false

3. Proof of correctness:
   Multiplying polynomials cause the exponents in each of their terms to add. Cubing a polynomial results in a polynomial that contains all terms with all combinations of summing any three of the original polynomial's powers. Thus, "poly", an array of the exponential powers of the result of cubing the polynomial represented by "arr", will contain all combinations of summing any three elements of "arr." Thus, searching "poly" for t is correct.

4. Runtime analysis:
   FFT allows us to do polynomial multiplication in O(nlog n) time. Scanning an array for a particular item takes linear time. Thus, our algorithm runs in O(nlog n) time.

4.

(c) We replace all the 0s from both the RAM and the virus with -1's. We then reverse the virus's string and convert both the RAM and the virus into polynomials. RAM is then represented by $r(x) = r_0 + r_1x + \ + r_{m-1}x^{m-1}$ with $r_i = s_2[i], i \in \{0, 1, ..., m1\}$ The virus is represented by $v(x) = v_0 + v_1x + \ + v_{n-1}x^{n-1}$ with $v_i = s_1[n - i - 1], i \in \{0, 1, ..., n1\}$. We multiply these polynomials via FFT. There is a startling result: in the resultant polynomial, the coefficient c_{n+i-1} of any $x^{n+i-1}, i \in \{0, ..., m - n\}$ is the dot product of the virus and the substring in the RAM starting with index i. If this dot product is at least $n - 2k$ where k is the number of allowable errors, then it is a match.

The procedure is $m \log m$ time as it is dominated by calculating the FFT and the inverse FFT from multiplying the virus polynomial by the RAM polynomial. The dot products and scanning of the RAM take linear time. Thus, the algorithm runs in $O(m \log m)$.

5.

(a) A(1, 2), B(2, 11), D(3, 6), E(4, 5), G(7, 10), F(8, 9), C(13, 18), H(14, 17), I(15, 16)
AB(T), BD(T), DE(T), ED(B), BG(T), GD(C), GF(T), CH(T), AE(F), HI(T), CI(F)

(b) There are 8 strongly connected components

(c) Given n vertices, the graph can have $\binom{n}{2} = n(n-1)/2$ edges at most. This is $O(n^2)$.

(d) All edges belong to 2 vertices. Thus, the number of edges is the total degrees of a graph divided by 2. Because the number of edges is a whole number, the total degrees has to be even.

(e) We first prove the that if a graph G is bipartite then it has no odd cycles. Let $v_0, ..., v_k, v_0$ be a cycle in G. Assuming that G is bipartite, assume WLOG that v_0 is on the left side. Then v_1 has to be on the right side. Thus, all v_i where i is even is on the left side and all where i is odd is on the right side. The cycle that we proposed ended on v_0 which is on the left side. That we means we previously landed on an odd i on the right side. Thus, there are an equal number of odd and even vertices, meaning that cycle is even.
We now prove that if a cycle is odd it cannot be bipartite. Suppose G has no cycles of odd length. Pick a vertex $a \in V$. For every vertex $v \in V$, let p_v be any path from a to v, and let d_v be the length of this path. partition all $v \in V$ into L and R, where L contains all vertices with even length paths and R contains all vertices with odd length paths. L and R are a trivially a complete partitioning of G. If this is not a bipartite split, then there is some $u, v \in E$ where E is the set of edges such that both $u, v \in L$ or $u, v \in R$. In either case, there is now a closed walk in G given by $p_u, \{u, v\}, p_v$ with total length is $d_u + d_v + 1$, which is odd. Since G has a closed walk of odd length, then G also has a cycle of odd length which is a contradiction.

(f) We proceed via a constructive proof using induction.
Base case: 1 vertex. There are 0 edges in this graph so it is semiconnected and contains a direct path.
Inductive Hypothesis: For a DAG of size n, it is semiconnected iff there is a directed path that visits all vertices of G
Inductive Step: For a DAG of size $n + 1$, it is semiconnected iff there is a directed path that visits all vertices of G
Case 1: Some node v_n in the size n graph now points to the added v_{n+1}. If v_n was the last node in G, v_{n+1} is now the last node and it cleanly extends the properties of semiconnectness and having a direct path. If v_n is not the last vertex, we add an edge from $v_n + 1$ to the child of v_n to ensure that it is semiconnected to the descendants of all the latter vertices in the graph. In this way, the directed path is preserved as well via a detour through v_{n+1}.
Case 2: v_{n+1} points to some v_n in our original graph. If v_n was the first node in G, v_{n+1} now supplants it as the first node and the properties of semiconnectedness and having a direct path still hold. If v_n was not the first node, we draw an edge from v_n's parent to v_{n+1}. Without this, the ancestors of v_n would not be semiconnected to v_{n+1} and could be without creating a cycle. The new edge is still in the same direction and does not alter the fact that there exists a directed path through G.

Thus, we have proven that any construction of G will inevitably yield both characteristics.