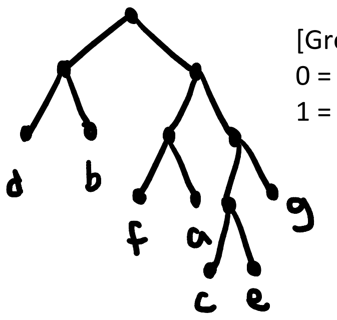
CS341 S2017 Midterm

1. a) By the Master Theorem, we have that 2 = 2^1. Hence we have

T(n) in **O(nlogn)**

b) By the Master Theorem, we have that 4 = 2^2. Hence we have

T(n) in **O(n^2logn)**

2. 

[Greedy problem]
0 = left
1 = right

3. [Greedy problem]
Prefix sum: We create a prefixSum array, where prefixSum[i] is the sum of all the elements from 0 to i. This can be done in linear time by iterating through the array once, and updating the running sum.

```
function calculatePrefixSum(array) {
    for int i from 1 to array.length - 1 {
        array[i] = array[i-1] + array[i]
    }
    return array
}
```

Then to answer the sum, we just return prefixSum[j] - prefixSum[i].

4. [PQ1 problem]
We use an extended version of Karatsuba's algorithm. We split the polynomials into upper and lower halves, denoted by Pu(x), Pl(x), Qu(x), Ql(x). We have that

$$P(x) = P_u(x)\, x^{n/2} + P_l(x)$$
$$Q(x) = Q_u(x)\, x^{n/2} + Q_l(x)$$

We use a divide and conquer algorithm as follows.

```
function polynomialMult(P, Q) {
    initialize Pu, Pl, Qu, Ql
    lower = polynomialMult(Pl, Ql)
    upper = polynomialMult(Pu, Qu)
    middle = polynomialMult(Pl + Pu, Ql + Qu)
    return upper * x^n + (middle - lower - upper) * x^n/2 + lower
```

}

Runtime: We have 3 recursive calls each with a subproblem of size n/2. Our remaining operations take O(n) time, since we are adding and subtracting. Hence we have that the recurrence relation is T(n) = 3T(n/2) + O(n), which resolves to O(n^log_2(3)).

5. [A2 problem]
We have that a graph G has an odd directed cycle if and only if one of its strongly connected components is non-bipartite (as an undirected graph). Hence our algorithm works as follows.

We first run DFS to find its strongly connected components. For each strongly connected component, we check if its undirected version is bipartite, by converting it to an undirected graph then running BFS to check for bipartite-ness. If one of the SCCs is non-bipartite, the graph contains an odd directed cycle and we return true. Otherwise we return false.

6. [similar to Tutorial 3 problem]
Let A be the adjacency matrix of the graph. We have that A^k (i, j) is the number of k-length walks from i to j.

We prove this statement by induction on the walk length, k.

Base case: k=1. We have that the A(i,j) = 1 if there is an edge from i to j, and 0 otherwise. Hence A(i,j) represents the number of length 1 walks from i to j.

Inductive step. Assume A^k (i, j) represents the number of k-length walks from i to j. A walk of length n+1 from i to j can be expressed as an n-length walk from i to some k plus an edge from k to j. Hence we have that the number of n+1-length walks from i to j is the sum of all walks from i to k times the number of ways to walk in one step from k to j. Formally, this is

$$\sum_{k=1}^{n} A(k,j)\, A^{n}(i,k)$$

which is the formula for multiplying the matrix A^n by A, which is A^(n+1). Hence we have proved the statement.

A triangle is a path of length 3 from a vertex to itself. To check if there is a triangle, we can check if there are any non-zero entries in A^3 (i, i), for all vertices i. If there is such a vertex, we can check all pairs of edges from it to find the triangle.

7. [A2 problem]
We use a modified BFS algorithm to return the number of shortest s-t paths. We initialize an array numpaths, where numpaths[v] is the number of shortest paths from s to v. We initialize this array to 0, except numpaths[s] = 1. We also initialize an array mindist, where mindist[v] is the minimum path length from v to s. In each iteration of the BFS, we update numpaths as follows. Let v be the current vertex we are traversing in the BFS. For each of its neighbours u, we:

 If u has not been visited, we set numpaths[u] = numpaths[v], and mindist[u] = mindist[v] + 1.

Otherwise, if mindist[u] = mindist[v] + 1, we update numpaths[u] to numpaths[u] + numpaths[v].

At the end of the BFS, we return numpaths[t].

Correctness: We prove correctness of this algorithm by induction on k, the length of the shortest path from s to any vertex t.

Base case: k=0. Then s=t, and there is one shortest path from s to itself. We initialize numpaths[s] = 1, so we return the correct solution.

Inductive step: Assume the algorithm computes the correct number of shortest paths for all paths of length k. Every path of length k+1 from s to t must pass through a neighbor of t with distance k from s. By the inductive hypothesis, our algorithm computes the correct number of k-length shortest paths. Hence our algorithm computes the correct number of k+1-length shortest paths.

8. [Greedy problem]
Our algorithm works as follows. We place the first tower a distance of k from x1 (x1 + k). We then iterate through the rest of the houses, placing a tower at (xi + k) if house xi does not have service.

```
function cellStations(x1… xn, , k) {
    lastStation = x1 + k
    stations = [ lastStation ]
    for i = 2 to n
        if (abs(xi - lastStation) > k)
            lastStation = xi + k
            stations.push(lastStation)
    return stations
}
```

Correctness: Suppose there is a different optimal algorithm that returns a solution y = [y1,…yk]. Let s = [s1,…sj] be the solution returned by our algorithm. Let c be the first index where yc and sc differ. Since our algorithm always places stations k miles after the next unserviced house and both solutions are the same before this station, neither solution has a station in range of the house at c. Hence the tower at yc and sc must be in range of the house. Since sc = house + k, we must have that yc < sc, otherwise it would not cover this house and y would not be optimal. We can then exchange yc with sc and y would still be optimal, as it would have all houses covered with the same number of stations. We can repeat this with all differences and since after every exchange y is still optimal, we must have y = s.

9. [A1 problem]
We use a divide and conquer algorithm to find the maximum rectangular area. If there is only one building, we return the height of the building. For a range of buildings, the maximum area must be either
(1) Contained in the left half of the buildings.
(2) Contained in the right half of the buildings.
(3) The maximum area spans the 2 middle buildings.
Hence at each recursive step, we recurse in the left half of the buildings, then recurse in the

right half of the buildings, then find the maximum area containing the 2 middle buildings. Then we return the maximum of the 3.

To find the maximum area containing the 2 middle buildings, we start with the maximum area of a poster contained in these 2 buildings. At each iteration, we either add the taller building from the left or the right side. The current area then becomes the height of the currently shortest building multiplied by the width of the range of buildings. We iterate until our range now spans all the buildings, keeping track of the maximum area as we iterate. Since this process iterates through each building exactly once, it takes Θ(n) time, where n is the number of buildings.

Runtime: Recursing in the left and right halves take T(n/2) time each (sloppy). Hence the recurrence relation for this algorithm is T(n) = 2T(n/2) + Θ(n). Using the Master Theorem, we have a = 2, b = 2, and c = 1. We have that logb(a) = log2(2) = 1 = 1 c. Hence by the Master Theorem,

T(n) ∈ Θ(n log n)

Correctness: There are only 3 possibilities for where the largest rectangle is: we compute the maximum of all 3. We prove correctness for the algorithm to compute the area of the largest rectangle spanning the middle two buildings. The height of the rectangle has to be less than or equal to the height of the two middle buildings. The algorithm calculates the maximum area of the rectangle with height less than the height of the midpoint. For every height h(i)≤h(mid) the algorithm finds the smallest index to the right of the midpoint with a height less than h(i) and the largest index to the left of the midpoint with a height less than h(i).

Bonus
We find the maximum area as follows. For every building k, we calculate the maximum area where some i is the shortest building. Then we take the maximum of all of these. The maximum area must have the height of the shortest building in its range. Hence this method calculates the maximum area.

For each building k, we must know the index of the first shorter building to its left, i, and the index of the first shorter building to its right, j. Then the maximum area with building k as the shortest building must be the height of k times the width j − i.

To do this, iterate through the buildings from left to right, maintaining an increasing stack of buildings (the height of buildings in the stack is monotonically increasing). At each iteration, we pop off the stack any buildings that are taller than the current one. For each popped item k, we calculate the maximum area where k is the shortest building. The first shorter building to its left must be the current top of the stack, since we maintained our stack to be increasing. The first shorter building to its right must be the current building. We can calculate the area using the difference between indices for the width and the height of the popped building. Then we push the current building to the stack. We iterate until we have reached all the buildings.

Runtime: Every building is pushed to the stack once, and every building is popped from the stack at most once. Hence the runtime of this algorithm is T(n) ∈ O(n).