A1 S2021

1. a) [A1 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)).

b) Our algorithm works as follows. For each integer a_k, we would like to find 2 other integers a_i and a_j that sum to it. We do this by forming the polynomial

$$P(x) = x^{a_1} + \cdots + x^{a_{k-1}}$$

We represent this as an array of coefficients of length n, with a 1 at each a_i from i=1 to k-1, and 0 otherwise. We do not include the integers for i >= k, since we cannot form a sum to a_k with two positive numbers greater than it.

Then we compute P(x)^2. The coefficient of x^a_k is the number of pairs of integers that sum to a_k. We divide this by 2, since we double count. By running this for every a_k, we find the total number of triples such that i < j < k and a_i + a_j = a_k.

Note that all a_i <= n.

```
function counting(a1…an) {
    result = 0
    for k=1 to n
        P = array of length n, 0-initialized
        for i=1 to k-1
            P[a_i] = 1
        P2 = polynomialMultiply(P, P)
        result += P2[a_k]/2
    return result
}
```

Runtime: We iterate through each element in the array once. In each iteration, we create the polynomial, which takes O(n) time. Polynomial multiplication takes O(n^log_2(3)) time. Hence our algorithm runs in O(n*n^n^log_2(3)) time.

2. [A1 problem]
A) We have that T (n) = T (2n/3) + T (n/3) + n^2 ≥ n^2 for all n ≥ 0. Hence we have that T(n) ∈ Ω(n^2).

We show that T (n) ∈ O(n^2). Consider the recurrence S(n) = T(2n/3)+T (2n/3)+n^2 = 2T(2n/3)+n^2. Using the Master Theorem, we have a = 2, b = 3/2 , and c = 2. We have that log_b(a) = log_3/2(2) ≈ 1.71 < c. Hence by the Master Theorem, S(n) ∈ Θ(n^2) and thus S(n) ∈ O(n2). Since for all n ≥ 0 we have that T(n) ≤ S(n), we have that T(n) ∈ O(S(n)). Hence we have that T(n) ∈ O(n^2).

Since we have T(n) ∈ Ω(n^2) and T(n) ∈ O(n^2), we have that T(n) ∈ Θ(n^2).

B) At level k of the recursion tree, there are sqrt(n) * n^1/4 *…*n^1/2k = n^(1-2^-k) nodes with each node having n^(1/2k) work. Hence the total work on level k is n.

When k = loglogn, the subproblem size is n^1/2k = 2. Hence the total work done is at most n*k = nloglogn, so we have that T(n) ∈ $O(n\log\log n)$

3. We use a divide and conquer algorithm for this problem. We divide the array of segments in half and calculate the intervals for each half. Then we merge them together.

To merge the two halves, we iterate through both arrays of intervals at once, as follows. In each iteration, we pick the interval with the smaller x-coordinate. The height for this interval is the maximum of the current heights of each half.

```
// Returns array of tuples (intervalStart, height)
// ie the end of an interval is the intervalStart of the next interval
function visibleSegments(segments) {
    if (segments.length == 0) return []
    if (segments.length == 1) return [(segments[0].first, segments[0].height), (segments[0].last, 0)]

    mid = segments.length / 2
    left = visibleSegments(segments[0 to mid])
    right = visibleSegments(segments[mid+1 to end])

    result = []
    leftH, rightH = 0

    while (!left.empty() or !right.empty())
        if left.empty() add remaining from right to result and break
        if right.empty() add remaining from left to result and break

        init interval
        if (left.top().first < right.top().first)
            interval = left.pop()
            leftH = interval.height
        else if (left.top().first > right.top().first)
            interval = right.pop()
```

```
                rightH = interval.height
            else
                interval = left.pop()
                leftH = interval.height
                rightH = right.pop().height

            height = max(leftH, rightH)
            if (result.isEmpty() or height != result.lastitem.height)
                result.push((interval.start, height))
        return result
}
```

Runtime: Our merge operation takes O(n) time, since we iterate through each element of the left and right halves once. Hence the recurrence relation for our algorithm is T(n) = 2T(n/2) + O(n), which resolves to O(nlogn).

4. [In progress]