A1 S2017

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 a value greater than or equal to it. We do this by forming the polynomial

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

We represent this as an array of coefficients of length n, with a 1 at each a_i from i=1 to n, excluding a_k, and 0 otherwise.

Then we compute P(x)^2. The coefficient of x^a_i is the number of pairs of integers that sum to a_t. Hence we iterate through all numbers from a_k+1 to n, and sum their coefficients. We divide this value by 2, since we double count each pair. 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 the sum of 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)
		for i=a_k+1 to n
			result += P2[i]/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. Iterating from a_k+1 to n takes O(n) 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(n2).

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. [similar to Tutorial 2 problem]
We first derive an algorithm to find the indices of the maximum subarray in a 1D array. We use Kadane's algorithm, which works as follows. We define a variable currentSum which stores the maximum sum ending at our current location. We also keep track of a global maximum sum. We iterate through the array, and add the value of the current element to currentSum. If this value is less than zero, we reset currentSum to 0. This allows us to discard negative subarrays.

```
function maxSum1D(A) {
    maxSum = INT_MIN
    currentSum = 0

    // Indices for maxSum and currentSum ranges
    maxSumStart = 0
    maxSumEnd = 0
    currentSumStart = 0

    for i=0 to n
        currentSum += A[i]
        if (maxSum < currentSum)
            maxSum = currentSum
            maxSumStart = currentSumStart
            maxSumEnd = i
        if (currentSum < 0):
            currentSum = 0
            currentSumStart = i + 1
    return maxSum, maxSumStart, maxSumEnd
}
```

Since we iterate through the array once, this algorithm runs in O(n) time.

Using our maxSum1D algorithm, we now derive an algorithm for the maximum sum rectangle problem. We iterate through all possible upper and lower boundaries for a rectangle in A. For each pair (u, d), we create an array B, where B[i] = sum(A[u…d][i]. Then we use the maxSum1D algorithm above to calculate the left and right indices of the maximum rectangle with upper and lower bounds (u, d).

```
function maxRectangle(A) {
        maxRect = 0
        indices = (0,0,0,0)
        for u=1 to n
                B = [0]*n

                for d=u to n
                        for i=1 to n: B[i] += A[d][i]
                        maxRect = max(maxRect, max1DSum(B))
                        update indices
        return maxRect, indices
}
```

Runtime: Iterating through all pairs (u, d) takes O(n^2) time. For each pair (u, d), we call create the array B and max1DSum, which takes O(n) time. Hence our algorithm takes O(n^3) time.

4. We know that preorder[0] is always the root of the tree. We can find preorder[0] in inorder, at some index i. inorder[i] is traversed after its left subtree, so inorder[start to i-1] is the left subtree, and inorder[i+1 to end] is the right subtree. Hence we can build the tree by recursively building its children.

```
function buildTree(preorder, inorder) {
        if (inorder)
                root = preorder.popfront()
                i = index of root in inorder

                treeNode = new TreeNode(root)
                treeNode.left = buildTree(preorder, inorder[start to i-1]
                treeNode.right = buildTree(preorder, inorder[i+1 to end]

                return treeNode
}
```

Runtime: We iterate through each element of preorder once. For each iteration, it takes O(n) time to find the index of preorder[0] in inorder. Hence our solution runs in O(n^2) [can optimize by using a hashmap for inorder].

Correctness: Prove by induction.

Bonus
This algorithm is the same idea as the one above but we do it iteratively. It works as follows. We keep pushing nodes from preorder into a stack. For each node we push onto the stack, we create a TreeNode for it and push it onto a treeNode stack. We push nodes from preorder until the top of the stack matches the first element of inorder. We then pop elements from the stack

until it no longer matches inorder.

```
function buildTree(preorder, inorder) {
	init stack, treeNodeStack
	createRightChild = false

	root = preorder.pop()
	treeRoot = new TreeNode(root)
	curTreeNode = treeRoot

	while (!preorder.empty())
		if (!stack.empty() and stack.top == inorder.top())
			treeRoot = treeNodeStack.pop()
			stack.pop()
			createRightChild = true
			inorder.pop()
		else
			node = preorder.pop()
			stack.push(node)
			treeNode = new TreeNode(node)
			if (createRightChild)
				createRightChild = false
				curTreeNode.right = treeNode
				curTreeNode = treeRoot.right
			else
				curTreeNode.left = treeNode
				curTreeNode = treeRoot.left

			treeNodeStack.push(treeNode)
	return treeRoot
}
```

Runtime: We pop every element from preorder and inorder once each. Hence this algorithm runs in O(n) time.