1.

a. To show that $\frac{3n+4}{n^2+2}$, we need to find constants c>0 and n_0 ≥1 such that $\frac{3n+4}{n^2+2} \le \frac{c}{n}$, $\forall n \ \geq \ n_0$
b. When n is sufficiently large, the n^2 term in the denominator dominates the constant term, so: n^2+2≈n^2. Thus for a large n, $\frac{3n+4}{n^2+2} \approx \frac{3n+4}{n^2}$.
c. Now the fraction can be split, $\frac{3n+4}{n^2} = \frac{3n}{n^2} + \frac{4}{n^2} = \frac{3}{n} + \frac{4}{n^2}$.
d. For large n, $\frac{4}{n^2}$ will be very small and ignored compared to $\frac{3}{n}$. Thus, $\frac{3}{n} + \frac{4}{n^2} \le \frac{3}{n} + \frac{4}{n} = \ \frac{7}{n}$
e. Thus for large n, $\frac{3n+4}{n^2+2} \le \ \frac{7}{n}$
f. Therefore, c = 7 and n_0 = 1 to satisfy the Big O definition $\frac{3n+4}{n^2+2} \le \ \frac{7}{n}$ for all $n \ \geq \ n_0$.

2.

a. F1(n) is O(g1(n)) which means there exist positive constants c1 and n1 such that for all n≥n1, $f1(n) \ \le \ c1 \ * \ g1(n)$
b. F2(n) is O(g2(n)) which means there exist positive constants c2 and n2 such that for all n≥n2, $f2(n) \ \le \ c2 \ * \ g2(n)$
c. For n≥max (n1, n2), both inequalities hold
d. Multiply both inequalities together to get: $f1(n) \ * \ f2(n) \ \le \ (c1 * c2) \ * \ (g1(n)) \ * (g2(n))$
e. Let c = c1*c2, then $f1(n) \ * \ f2(n) \ \le \ (c) \ * \ (g1(n)) \ * (g2(n))$ n≥max (n1, n2)
f. Thus, by big O definition, $f1(n) \ * \ f2(n) \ is \ O((g1(n)) \ * (g2(n)))$

3.

a. Assume that 4^n is O(2^n). This means there exist constants c>0 and n_0 ≥1 such that $\forall n \ \geq \ n_0$.
b. From our assumption, we have: $4^n \le c \ * \ 2^n$
c. This = $(2^2)^n \ \le c \ * \ 2^n$, and divide both sides by 2^n.
d. This = $2^n \le \ c$. This must hold true $\forall n \ \geq \ n_0$. However, when n grows larger, 2^n grows exponentially. Thus for constant c, there is always an n large enough that $2^n > c$.
e. This leads to a contradiction, thus our initial assumption is false.

4.

a. PART A) I will prove that the algorithms loops are finite and decrease to a stopping condition. Specifically:
   i. For i=0,, j runs from 0 to n−1, making n iterations.
   ii. For i=1, j runs from 1 to n−1 , making n−1 iterations.
   iii. This pattern continues until i=n−1, where j runs from n−1 to n−1, making 1 iteration.
   iv. Thus, the number of iterations for j in total is finite. Since the function isPalindrome() is constant and there are no infinite loops, the algorithm will terminate after processing all possible substrings of T
b. PART B) I will prove that for any input string T of length n ≥1, the algorithm correctly counts the palindrome substrings.
   i. The variable C is initialized to 0.
   ii. The outer loop iterates over all possible starting positions i of substrings.
   iii. The inner loop iterates over all possible ending positions j of substrings that start from i.
   iv. For each pair (i,j) the substring T[i:j+1] is extracted. The function isPalindrome checks if this substring is a palindrome and If it is, c is incremented by 1.
   v. Since the algo checks all possible substrings T[i:j+1] and counts palindromes and isPalindrome() checks all palindromes, the algorithm works correctly.
c. PART C)
   i. Outer loop runs n times, inner loop, for each i, runs n-i times
   ii. Iterations in the inner loop can be counted as $\frac{n(n+1)}{2}$, which is O(n^2).
   iii. Extracting a substring T[i:j+1] takes O(n) time in worst case, and isPalindrome is O(1)
   iv. O(n^2) * O(n) = O(n^3)
   v. Thus the time complexity in worst case is O(n^3).

5.
   a. PART A)
      i. In the best case the loop iterates n times, making i increment 1 each time, as shown here:
      ii. for i = 0, step is set to 1, For each i > 0, if A[i] <= 0, i is simply incremented by step (which is 1). If no element A[i] is greater than 0, the condition A[i] > 0 is never true, so step remains 1.
      iii. So the best case time complexity is O(n)
   b. PART B)
      i. For i = 0, step is set to 1. Whenever A[i] > 0, A[i] is set to -A[i] and step is set to -1. This causes i to decrement in the next iteration. In the worst case, every element in the array could be positive, causing the algorithm to frequently move backwards.
      ii. Each positive element is negated at most once. Therefore, every element will be processed in max two passes (one increment and one decrement)

iii. In the worst case, the number of operations may be up to twice the size of the array due to back and forth movements, thus time complexity would be $O(2n) = O(n)$
iv. Thus the worst case time complexity is $O(n)$.