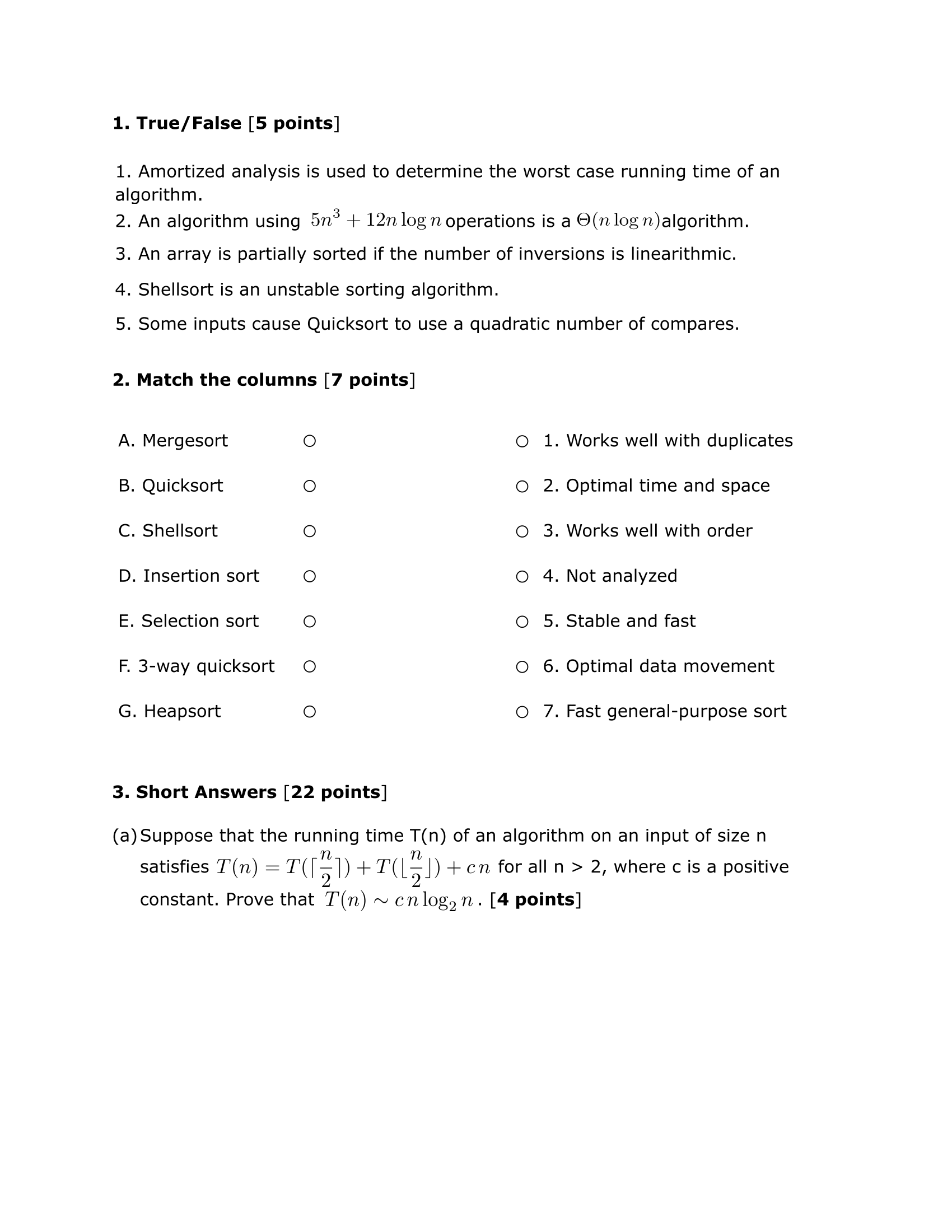


T



Therefore…

c \* () = c \* \* (k+1)

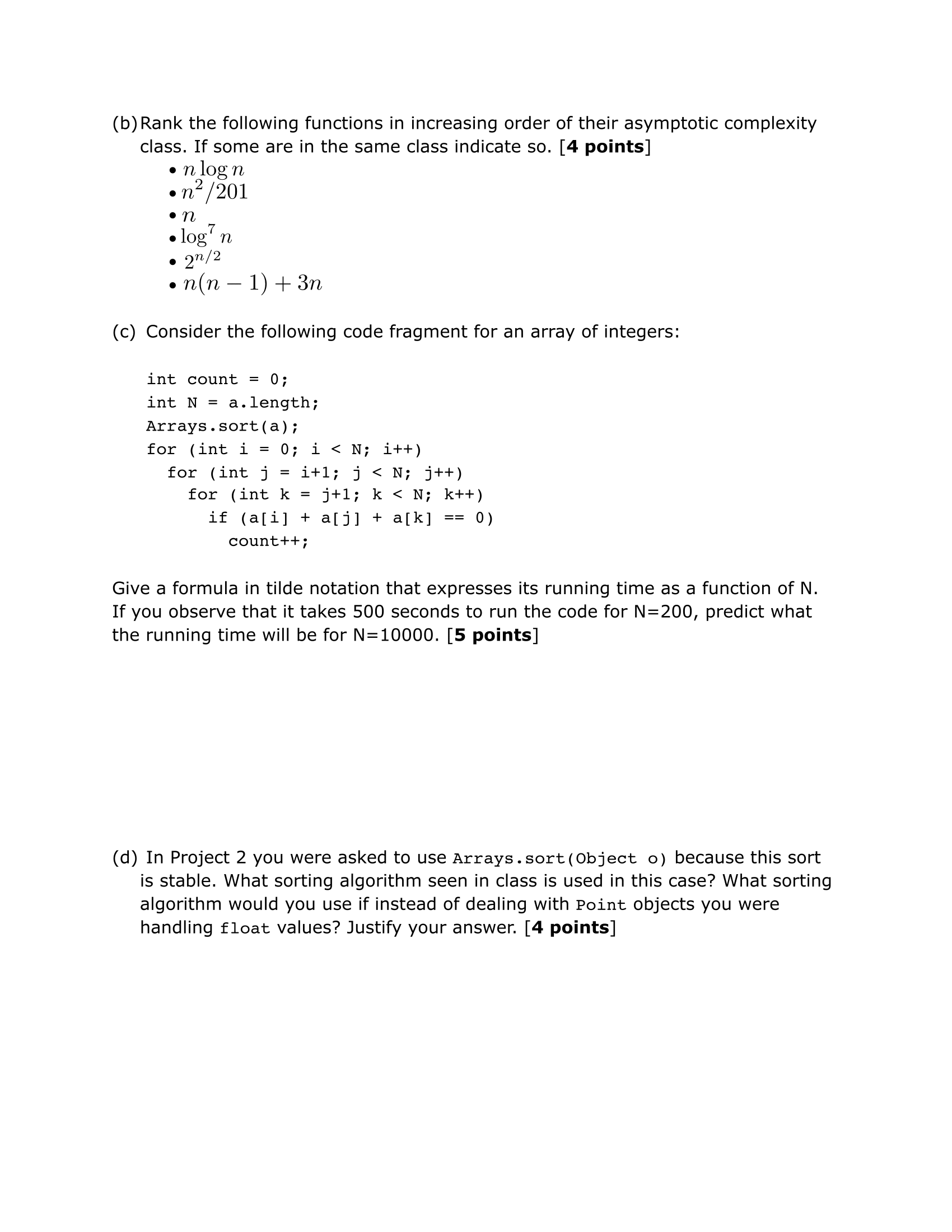
Proving…

F

T

T

F



The tilde approximation for the inner loop is:

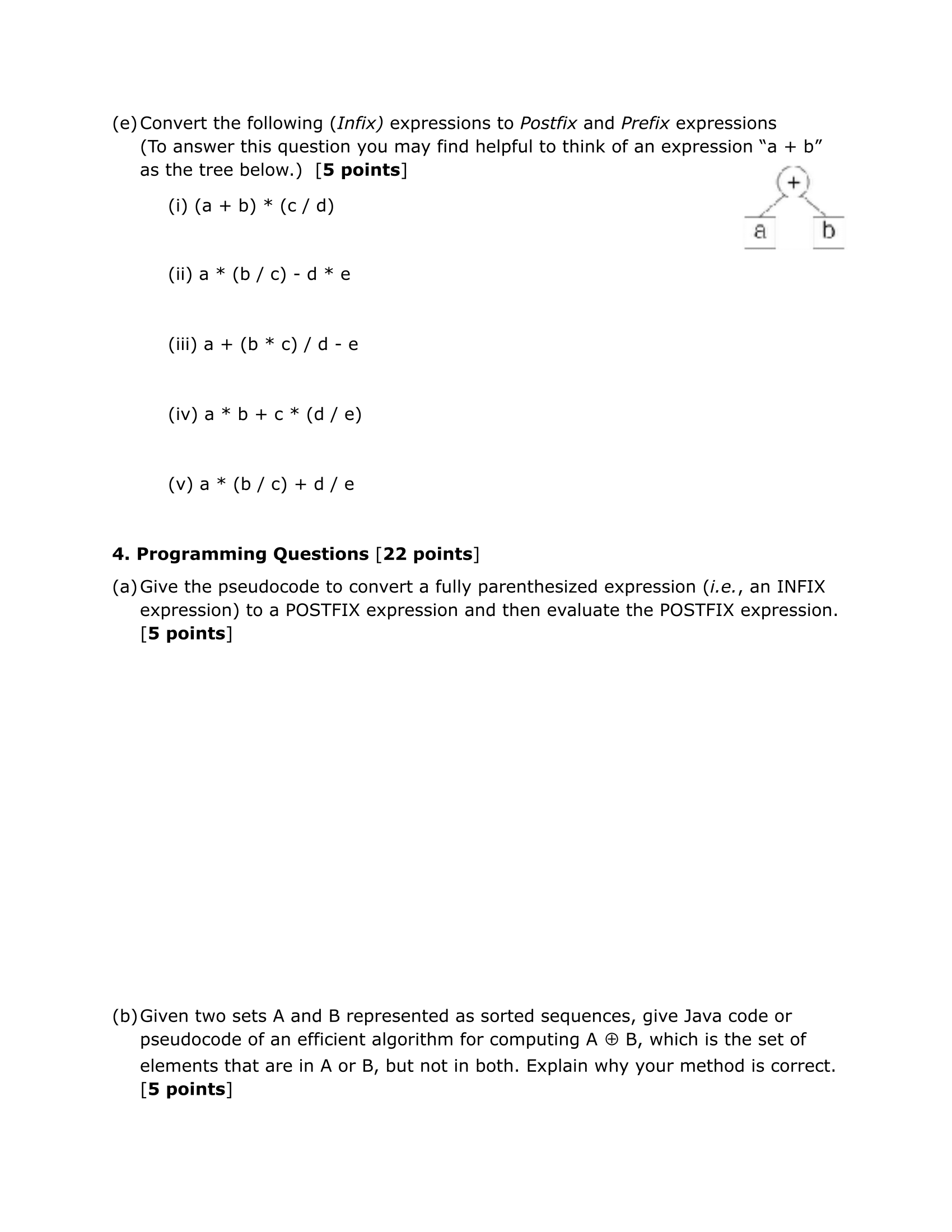
To determine the time interval, let’s set a variable ‘t’ such that…

using the given information we get

This gives us t = 3/8000. Plugging in we now can solve for N = 10,000:



In the case of the project, Arrays.sort utilized *merge sort* as it was sorting an array of objects. If it were to sort an array of primitive types (such as an array of floats, in the case of this example) Arrays.sort would instead utilize quicksort. The reason merge sort is used for an array of objects is because it is stable as well as faster in the case of dealing with objects (since there is a possibility of duplicate object references in a ‘to be sorted’ array). Quick Sort can be used for primitive types because they have no identity and thus their relative position in the order does not matter. In an object array the relative position might also matter, making merge sort viable over quick sort for Objects. Overall-- merge sort is faster and more accurate for objects while quick sort is simply more practical for primitive data types.



**String infixToPostfix(String toTok) {**

**StringTokenizer e = new StringTokenizer(toTok);**

**String toRet = “”;**

**Stack<String> stack;**

**while (e.hasMoreTokens()) {**

**String token = e.nextToken();**

**If (token is operand) {**

**toRet = toRet + token + “ “;**

**} else if (token == “)”) {**

**While (stack.top() != ‘(‘)**

**toRet = toRet + stack.pop() + “ “;**

**stack.pop();**

**} else {**

**While (ISP(stack.top() >= ICP(token))**

**toRet = toRet + stack.pop() + “ “;**

**stack.push(token);**

**}**

**}**

**While (!stack.empty())**

**toRet = toRet + stack.pop() + “ “;**

**Return toRet;**

**}**

Postfix: a b \* c d e / \* +

Prefix: + \* a b \* c / d e

Postfix: a b c / \* d e / +

Prefix: + \* a / b c / d e

Postfix: a b c \* d / + e -

Prefix: - + a / \* b c d e

Postfix: a b c / \* d e \* -

Prefix: - \* a / b c \* d e

Postfix: a b + c d / \*

Prefix: \* + a b / c d

**int calcPostfix(String toTok) {**

**StringTokenizer e = new StringTokenizer(toTok);**

**Stack<Integer> stack;**

**While (e.hasMoreTokens()) {**

**String token = e.nextToken();**

**If (token is operand) {**

**stack.push(Integer.parseInt(token));**

**} else {**

**Remove operands related to operator ‘token’ from the stack**

**Perform the mathematical operation ‘token’**

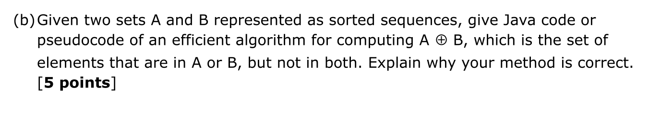
**Push the result to the stack**

**}**

**}**

**return stack.pop();**

**}**



public static ArrayList<Integer> findIntersection(int[] A, int[] B) {

ArrayList<Integer> intersection = new ArrayList<Integer>();

int n1 = A.length;

int n2 = B.length;

int i = 0, j = 0;

while (i < n1 && j < n2) {

if (A[i] > B[j]) {

j++;

} else if (B[j] > A[i]) {

i++;

} else {

intersection.add(A[i]);

i++;

j++;

}

}

return intersection;

}

ANSWER TO (b)

**JUSTIFICATION:**

For the above requested algorithm, I believe I have coded it in the most efficient way possible. The above method results in a O(n) runtime since it iterates two separate counters and compares the array indexes at these counters. This method is very efficient because instead of having to do a binary search there is instead just simple iteration from 0 to n (n being the size of the array). This results in less of a runtime for various sizes of n and thus making this algorithm the most efficient, especially as opposed to a binary search (where it can be up to O(nlogn) ) This method only works because the arrays are sorted **beforehand** so we take advantage of that and benefit our runtime (as seen demonstrated in the code above).

**import java.util.Arrays;**

**public class countMergeAndSort {**

**public static long merge(int[] arr, int[] left, int[] right) {**

**int i = 0;**

**int j = 0;**

**int count = 0;**

**while (i < left.length || j < right.length) {**

**if (i == left.length) {**

**arr[i+j] = right[j];**

**j++;**

**} else if (j == right.length) {**

**arr[i+j] = left[i];**

**i++;**

**} else if (left[i] <= right[j]) {**

**arr[i+j] = left[i];**

**i++;**

**} else {**

**arr[i+j] = right[j];**

**count += left.length - i;**

**j++;**

**}**

**}**

**return count;**

**}**

**Public static long sortAndCount(int[] arr) {**

**if (arr.length < 2)**

**return 0;**

**int m = (arr.length + 1) / 2;**

**int left[] = Arrays.copyOfRange(arr, 0, m);**

**int right[] = Arrays.copyOfRange(arr, m, arr.length);**

**return sortAndCount(left) + sortAndCount(right) + merge(arr, left, right);**

**}**

**}**

