1. (4 points)
1a) Use pseudocode to specify a brute-force algorithm that determines when given as input a sequence of n positive integers whether there are two distinct terms of the sequence that have as sum a third term. The algorithm should loop through all triples of terms of the sequence, checking whether the sum of the first two terms equals the third.

1b) Give a big-$O$ estimate for the complexity of the brute-force algorithm from part (a).

1c) Devise a more efficient algorithm for solving the problem that first sorts the input sequence and then checks for each pair of terms whether their difference is in the sequence.

1d) Give a big-$O$ estimate for the complexity of this algorithm. Is it more efficient than the brute-force algorithm?

2. (1 point) Some sorting algorithms are NOT stable (duplicate keys relative order is preserved after sorting). However if every key in A[i] is changed to A[i]*n + i - 1 (assume 1<=i<=n) then all the new elements are distinct (and therefore stability is no longer a concern). After sorting, what transformation will restore the keys back to their original values? What is the effect on the runtime of any of the sorting algorithms if you add this transformation before executing the sort and un-transformation after the sort?

3. (2 points) Prove, by induction on k, that level k of a binary tree has less than or equal to $2^k$ nodes (root level has k=0).

4. (2 points) Use definition of big O to prove or disprove.
4a) is $2^{(n+1)}$ ?= O($2^n$)

4b) is $2^{(2n)}$ ?= O($2^n$)

5. (3 points) The following routine takes as input a list of n numbers, and returns the first value of i for which L[i] < L[i - 1], or n if no such number exists.

```c
int firstDecrease(int * L, int n) {
    for (int i = 2; i <= n && L[i] >= L[i-1]; i++) {
    }
    return i;
}
```

5a) What is the big-O runtime for the routine, measured as a function of its return value i?

5b) If the numbers are chosen independently at random, then the probability that firstDecrease(L) returns i is (i-1)/i!, except for the special case of i = n+1 for which the probability is 1/n! Use this fact to write an expression
for the expected value returned by the algorithm. (Your answer can be expressed as a sum, it does not have to
be solved in closed form. Do not use O-notation.)

5c) What is the big-O average case running time of the routine? Hint: Simplify the previous summation until
you see a common taylor series.

6. (3 points) Consider the following program and recursive function.

```c
#include <iostream.h>
void Z(int[], int, int);
void swap (int&, int&);

void main() {
    int A[3]={1,2,3};
    int n=3;
    Z(A, n, 0);
}

void swap(int &x, int &y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}

void Z(int A[], int n, int k) {
    if (k == n-1) {
        for (int i=0; i<n; i++)
            cout << A[i] << " ";
        cout << endl;
    } else {
        for (int i=k; i<n; i++) {
            swap(A[i], A[k]);
            Z(A, n, k+1);
            swap(A[i], A[k]);
        }
    }
}
```

6a) Demonstrate the execution, show the output, and explain what the program accomplishes.

6b) Give a recurrence equation describing the worst-case behavior of the program.

6c) Solve the recurrence equation.

7. (5 points) Problem 2-4: Inversions
Let A[1 .. n] be an array of n distinct numbers. If i < j and A[i] > A[j], then the pair (i, j) is called an inversion
of A.
7a) List the five inversions of the array <2, 3, 8, 6, 1>.

7b) What array with elements from the set {1, 2, . . . , n} has the most inversions? How many does it have?

7c) What is the relationship between the running time of insertion sort and the number of inversions in the input
array? Justify your answer.

7d) Give an algorithm that determines the number of inversions in any permutation on n elements in Θ(n lg n)
worst-case time. (Hint: Modify merge sort.)

8. (3 points) Give big-O bounds for T(n) in each of the following recurrences. Use induction, iteration or Master
Theorem.
8a) T(n) = T(n-1) + n

8b) T(n) = 2T(n/4) + n^(1/2)

8c) T(n) = T(n/4) + T(n/2) + n^2
9. (2 points) Communication security is extremely important in computer networks, and one way many network protocols achieve security is to encrypt messages. Typical cryptographic schemes for the secure transmission of messages over such networks are based on the fact that no efficient algorithms are known for factoring large integers. Hence, if we can represent a secret message by a large prime number $p$, we can transmit over the network the number $r = p \times q$, where $q > p$ is another large prime number that acts as the encryption key. An eavesdropper who obtains the transmitted number $r$ on the network would have to factor $r$ in order to figure out the secret message $p$.

Using factoring to figure out a message is very difficult without knowing the encryption key $q$. To understand why, consider the following naive factoring algorithm:

For every integer $p$ such that $1 < p < r$, check if $p$ divides $r$. If so, print "The secret message is $p$!" and stop; if not, continue.

$9a$) Suppose that the eavesdropper uses the above algorithm and has a computer that can carry out in 1 microsecond (1 millionth of a second) a division between two integers of up to 100 bits each. Give an estimate of the time that it will take in the worst case to decipher the secret message if $r$ has 100 bits.

$9b$) What is the worst-case time complexity of the above algorithm? Since the input to the algorithm is just one large number $r$, assume that the input size $n$ is the number of bytes needed to store $r$, that is, $n = \left( \log_2 r \right) / 8$, and that each division takes time $O(n)$. 