1. (4 points) Consider sorting $n$ numbers stored in array $A$ by first finding the smallest element of $A$ and exchanging it with the element in $A[1]$. Then find the second smallest element of $A$, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of $A$. Write iterative pseudocode for this algorithm, which is known as selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all $n$ elements? Give the best-case and worst-case running times of selection sort in $\Theta$-notation.

2. (3 points) 2a. What does the following recursive algorithm do? Demonstrate on a small input array $A$.

```java
void foo(int A[], int n) { //initial call foo(A, n)
    if (n <= 1) return;
    if (A[0] > A[n-1]) swap A[0] and A[n-1];
    foo(A+1, n-2); // A+1 is pointer to 2nd index position of array
    if (A[0] > A[1]) swap A[0] and A[1];
    foo(A+1, n-1);
}
```

2b. Write a recurrence describing the number of times the algorithm compares two members of array $A$, measured as a function of the array length $n$. You do not need to solve the recurrence relation exactly, but state if the solution is linear growth, polynomial growth or exponential growth.

3. (3 points) Analyze the time complexity of genBitStrings(int n) which generates all bit strings of length $n$.

```java
public class BitString {
    int length;
    boolean[] b;
    public BitString(int n) {
        length = n;
        b = new boolean[n];
    }
    public void genBitStrings(int n) {
        if (n == 0) {
            for (int i = 0; i < length; i++) System.out.print(b[i]+" " );
            System.out.println();
        }
        else {
            genBitStrings(n-1);
            b[n-1] = !b[n-1]; // complement (0 --> 1, 1 --> 0) the n-th bit of b
            genBitStrings(n-1);
        }
    }
}
```

4. (3 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 $k=0$).
5. (3 points) Throughout this course, we assume that parameter passing during procedure calls takes constant time, even if an \( N \)-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time = \( \Theta(1) \).
2. An array is passed by copying. Time = \( \Theta(N) \), where \( N \) is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time = \( \Theta(q - p + 1) \) if the subarray \( A[p .. q] \) is passed.

Consider the recursive binary search algorithm for finding a number in a sorted array. Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let \( N \) be the size of the original problem and \( n \) be the size of a subproblem.

6. (6 points) Give big-O bounds for \( T(n) \) in each of the following recurrences. Use induction, iteration or Master Theorem.

6a. \( T(n) = 2T(n/2) + n^{1/2} \)

6b. \( T(n) = T(n - 2) + n \)

6c. \( T(n) = 7T(n/3) + n^2 \)

7. (3 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 \( r = pq \), 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.

a. 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.

b. 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, \( n = (\log_2 r)/8 \), and that each division takes time \( O(n) \).