**HOMEWORK-3**
**Total Points: 65**

1. [10 Points] Prove that $2^{20n}$ is not $O(2^n)$.

   if $2^{20n}$ is $O(2^n)$
   $2^{20n} <= c * 2^n$
   $2^{19n} <= c$
   Clearly, $2^{19n}$ is not always less than a constant; therefore, $2^{20n}$ is not $O(2^n)$.
   **Full marks if the justification is correct; otherwise, give zero.**

2. [5 Points] Prove that $2^{20+n}$ is $O(2^n)$.

   if $2^{20+n}$ is $O(2^n)$
   $2^{20+n} <= c * 2^n$
   $2^{20} * 2^n <= c * 2^n$
   $2^{20} <= c$
   Because $2^{20}$ is a constant; therefore, $2^{20+n}$ is $O(2^n)$.
   **Full marks if the justification is correct; otherwise, give zero.**

3. [10 Points] What is the time complexity of the following algorithm? Justify your answer.

```
long long foo(long long n) {
  long long i = 0;
  while (i < n) {
     if (i % 5 == 0) {
       i = i + 6;
     }
     else if (i % 5 == 1) {
       i = i + 1;
     }
     else if (i % 5 == 2) {
       i = i + 16;
     }
     else if (i % 5 == 3) {
       i = i + 11;
     }
     else if (i % 5 == 4) {
       i = i + 21;
     }
  }
  return i;
}
```

   This algorithm increments the value of i by 55 after every 5th iteration.
   The value of i after 5 iterations = 55
   The value of i after 10 iterations = 55*2
   The value of i after 15 iterations = 55*3
   The value of i after 5 * k iterations = 55*k

   The algorithm stops after 5 * k iterations, if 55 * k = n

The total number of iterations = n/11
Because this algorithm does a constant number of operations in every iteration, the time complexity is O(n).

**Give full marks if the justification is correct; otherwise, give zero marks.**

4. [15 Points] What is the time complexity of the following algorithm? Justify your answer.

```
long long foo(long long n) {
  long long i = 0;
  while (i < n) {
     if (i % 5 == 0) {
       i = i +  6;
     }
     else if (i % 5 == 1) {
       i = i + 1;
     }
     else if (i % 5 == 2) {
       i = i + 16;
     }
     else if (i % 5 == 3) {
       i = i + 11;
     }
     else if (i % 5 == 4) {
       i = i * 5;
     }
  }
  return i;
}
```

The algorithm works as follows:
The value of i after 5 iterations = 34 * 5
The value of i after 10 iterations = $(34*5 + 34)*5 = 34*5^2 + 34*5$
The value of i after 15 iterations = $(34*5^2 + 34*5 + 34)*5 = 34*5^3 + 34*5^2 + 34*5$
The value of i after 5 * k iterations = $34 * (5^k + 5^{k-1} + 5^{k-2} + \ldots + 5)$
$= 34 * 5 * (1 + 5 + 5^2 + \ldots + 5^{k-1})$
$= 34 * 5 * (5^k - 1)/4$

The algorithm stops after 5 * k iterations, if $34 * 5 * (5^k - 1)/4 = n$
The total number of iterations = $5 * \log_5((4*n / 170) + 1)$
Because this algorithm does a constant number of operations in every iteration, the time complexity is O(log(n)).
**Give full marks if the justification is correct; otherwise, give zero marks.**

5. [15 Points] Compute the lower bound and upper bound on the number of operations performed by the recursive algorithm for Fibonacci. Use the expansion method discussed in class.
The recurrence relation for the lower bound is:
T(0) = T(1) = 2
T(n) >= 2T(n-2) + c
The recurrence relation for the upper bound is:

$T(0) = T(1) = 2$
$T(n) <= 2T(n-1) + c$

**Lower bound [7.5 points]:**

$T(n) >= 2T(n-2) + c$
$>= 2(2T(n-4) + c) = 2^2T(n-2*2) + c(1 + 2)$
$>= 2^2(2T(n-2*2 -2) + c) + c(1+2) = 2^3T(n-2*3) + c(1+2+2^2)$
…
$>= 2^kT(n-2k) + c(1+2+2^2 + \ldots + 2^{k-1})$
$>= 2^kT(n-2k) + c(2^k - 1)$ **// 5 points if up to this part is correct**

If n is even, substituting n-2k = 0
$T(n) >= 2^{n/2}\, T(0) + c(2^{n/2} - 1) = O(2^{n/2})$
If n is odd, substituting n-2k = 1
$T(n) >= 2^{(n-1)/2}\, T(1) + c(2^{(n-1)/2} - 1) = O(2^{(n-1)/2})$ **// 2.5 points if complexity is correct**

**Upper bound [7.5 points]:**

$T(n) <= 2T(n-1) + c$
$<= 2(2T(n-2) + c) = 2^2T(n-2) + c(1 + 2)$
$<= 2^2(2T(n-3) + c) + c(1+2) = 2^3T(n-3) + c(1+2+2^2)$
…
$<= 2^kT(n-k) + c(1+2+2^2 + \ldots + 2^{k-1})$
$<= 2^kT(n-k) + c(2^k - 1)$ **// 5 points if up to this part is correct**

Substituting, n-k = 1
$T(n) >= 2^{n-1}\, T(1) + c(2^{n-1} - 1) = O(2^n)$ **// 2.5 points if complexity is correct**

6. [10 Points] Write a recursive algorithm to check whether a number “n” is prime. The time-complexity of your algorithm should be less than O(n). Discuss why your algorithm is correct. What is the time complexity of your algorithm? Justify your answer.

```
// This program returns 1 if n is prime; otherwise, returns 0
// The initial value of i is 2
// This function checks if n is divisible by any integer between 2 to √n
int CheckPrime(int n, int i) {
  if (n % i == 0)
    return 0;
  else if (i * i > n)
    return 1;
  else
    return CheckPrime(n, i+1);
}
```

**// 5 points if the algorithm is recursive and correctly computes the prime number.**

The recurrence relation for the time complexity is

$T(m) = T(m-1) + c$ // where m is the problem size

$= T(m-2) + 2c$

$= T(m-3) + 3c$

$= \ldots$

$= T(m-k) + kc$

Substituting $m-k = 1$

$T(m) = T(1) + (m-1)*c = O(m)$

Because $m = \sqrt{n}$

The time complexity is $O(\sqrt{n})$.

**// 5 points if the time complexity is less O(n) and the justification is also correct.**