1. (10 points)

   (a) Find (with proof) a function f_1 such that $f_1(2n)$ is $O(f_1(n))$.

   (b) Find (with proof) a function f_2 such that $f_2(2n)$ is not $O(f_2(n))$.

(a) $f_1 = x$
$p.f. \quad let\ c = 4, n_0 = 1.\ For\ all\ n \geq 1, we\ have\ T(n) = f_1(2n) = 2n \leq 4n = cf_1(n)$
So, $f_1(2n) = O(f_1(n))$.

(b) $f_2 = 2^x$
$p.f. \quad T(n) = f_2(2n) = 2^{2x}.\ For\ all\ c > 0, when\ x > log_2 c, we\ have\ 2^{2x} > 2^x c,$
So, there doesn't exist such an n_0, that makes all $x > n_0$ satisfied $T(n) < cf_2(n)$,
So, $f_2(2n)$ is not $O(f_2(n))$.

2. (20 points) Use the Master theorem to give tight asymptotic bounds for the following recurrences.

   - $T(n) = 4T(n/2) + n^2$.
   - $T(n) = 8T(n/2) + n^3$.
   - $T(n) = 11T(n/4) + n^2$.
   - $T(n) = 7T(n/3) + n$.

The master theorem is:

$$If\ T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^k)\ for\ some\ constants\ a > 0, b > 1, k \geq 0, then$$

$$T(n) = \begin{cases} O(n^{log_b a}), & if\ a > b^k \\ O(n^k logn), & if\ a = b^k \\ O(n^k), & if\ a < b^k \end{cases}$$

(1) $a = 4, b = 2, k = 2,\ a = 4 = 2^2 = b^k,\ T(n) = O(n^2 logn)$
(2) $a = 8, b = 2, k = 3,\ a = 8 = 2^3 = b^k,\ T(n) = O(n^3 logn)$
(3) $a = 11, b = 4, k = 2,\ a = 11 < 16 = 4^2 = b^k,\ T(n) = O(n^2)$
(4) $a = 7, b = 3, k = 1,\ a = 7 > 3 = 3^1 = b^k,\ T(n) = O(n^{log_3 7})$

3. (20 points) Consider the following recursive algorithm. On input a list of distinct numbers, the algorithm runs in three phases. In the first phase, the first 2/3 elements of the list are sorted (recursively). In the second phase, the last 2/3 elements are sorted (recursively). Finally, in the third phase, the first 2/3 elements are sorted (recursively) again. Derive a recurrence describing the running time of the algorithm and use the recurrence to bound its asymptotic running time. Would you use this algorithm in your next application?

As described above, the algorithm runs in three phases:
1. The first 2/3 elements of the list are sorted (recursively).
2. The last 2/3 elements are sorted (recursively).
3. The first 2/3 elements are sorted (recursively) again.

After that, the list is sorted. As a result, the running time is derived as below:

$$T(n) = T\left(\frac{2}{3}n\right) + T\left(\frac{2}{3}n\right) + T\left(\frac{2}{3}n\right) = 3T\left(\frac{2}{3}n\right)$$

$So, a = 3, b = \frac{3}{2}, k = 0,\ \ a = 3 > 1 = \frac{3}{2}^0 = b^k$

$$T(n) = O\left(n^{\log_{\frac{3}{2}} 3}\right) = O(n^{\frac{1}{1-\log_3 2}})$$

$\log_{\frac{3}{2}} 3$ is a number that bigger than 2, so this is an relatively inefficient algorithms, I won't use it.

4. (30 points) In the table below, indicate the relationship between functions f and g for each pair (f, g) by writing **yes** or **no** in each box. For example, if $f = O(g)$ then write **yes** in the first box.

f	g	O	o	Ω	ω	Θ
$10logn$	log^3n	yes	yes	no	no	no
$nlog(2n)$	$nlogn$	yes	no	yes	no	yes
$\sqrt{logn}$	$loglogn$	no	no	yes	yes	no
$10n^2 + logn$	$n^2 + 11log^3n$	yes	no	yes	no	yes
$\sqrt{n} + logn$	$n^{2/3} + 10$	yes	yes	no	no	no
$n^2 2^n$	3^n	yes	yes	no	no	no
$n^{1/3}$	$(logn)^2$	no	no	yes	yes	no
$nlogn$	$\frac{n^2}{logn}$	yes	yes	no	no	no
$n!$	n^n	yes	yes	no	no	no
$logn!$	$logn^n$	yes	no	yes	no	yes

5. (**OPTIONAL — EXTRA CREDIT**, 30 points)

The Fibonacci numbers are defined by the recurrence

$$F_0 = 0, F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}, n \geq 2$$

(a) Assume that the cost of adding, subtracting, or multiplying two integers is **constant** (that is, $O(1)$), **independent of the size** of the integers.

- Write pseudocode for an algorithm that computes F_n based on the recursive definition above. Develop a recurrence for the running time of your algorithm and give an asymptotic lower bound for it, using the fact that $F_n \geq 2^{n/2}$, for $n \geq 6$.
- Write pseudocode for a non-recursive algorithm that asymptotically performs fewer additions than the recursive algorithm. Discuss the running time of the new algorithm.
- Show how to compute F_n in $O(\log n)$ time using only integer additions and multiplications. Provide an upper bound for the running time of this algorithm. (Hint: Express F_n in matrix notation as follows

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}.$$

Consider how you may use powers of the matrix $\left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ in the equation above to compute F_n.)

(b) Now assume that adding two m-bit integers requires $\Theta(m)$ time and that multiplying two m-bit integers requires $\Theta(m^2)$ time. What is the running time of the three algorithms under this more reasonable cost measure for the elementary arithmetic operations?

(a) 1. The recursive algorithm:

```
Function Fibonacci_rec(N)
        If N == 0 then return 0
        Else if N == 1 then return 1
        Else return (Fibonacci_rec(N-1) + Fibonacci_rec(N-2))
        Endif
End
```

The recurrence: $T(n) = T(n-1) + T(n-2) + c$

$$\mathrm{T(n)} = F_n \mathrm{T}(1) + F_{n-1}\mathrm{T}(0) + \mathrm{c} = \Omega(F_n + F_{n-1})$$

Because $F_n \geq 2^{\frac{n}{2}},\ for\ n \geq 6$, so $F_n + F_{n-1} \geq 2^{\frac{n}{2}} + 2^{\frac{n-1}{2}}\ for\ n \geq 6,\ F_n + F_{n-1} = \Omega(2^n)$

So, the lower bound for the running of the recursive algorithm is $\mathrm{T(n)} = \Omega(2^n)$

2. The non-recursive algorithms :

```
Function Fibonacci_nrec(N)
        F(0) = 0 ;
        F(1) = 1 ;
        if N >= 2
                for i = 2: N
                        F(i) = F(i-1) + F(i-2);
                Endfor
        Endif
        Return(F(N));
End
```

The running time in this algorithm is $\mathrm{O(n)}$, and this is lower than the lower bound for the running time of the recursive algorithm.

3. $\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, now the problem is to compute $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$.

We can solve it recursively, and this will take $\mathrm{O(logn)}$ time.

The pseudocode is shown as below:

```
Function Fibonacci_mat(N)
        [a11, a12, a21, a22] = Matrix_power(N-1);
         Return a11;
End
Function [a11, a12, a21, a22] = Matrix_power(M)
        If M = 1
                a11 = 1;
                a12 = 1;
                a21 = 1;
                a22 = 0;
        else
                [b11, b12, b21, b22] = Matrix_power(M/2);
                a11 = b11*b11 + B12*21;
                a12 = b11*b12 + b12*b22;
                a21 = b21*b11 + b22*b21;
                a22 = b21*b12 + b22*b22;
        endif
    end
```

The upper bound of this algorithm is $\mathrm{O}(\mathrm{logn})$.

(b) The digit of the Nth Fibonacci number is $\log_{10}(F_n) = \Omega\left(log_{10}\left(2^{\frac{n}{2}}\right)\right) = \Omega(\mathrm{n})$.

On the other side, it is obvious that the digit of the Nth Fibonacci number is $\mathrm{O}(\mathrm{n})$, so the digit of the Nth Fibonacci number is $\Theta(n)$.

For the first algorithm, the running time satisfies

$$T(n) = T(n-1) + T(n-2) + \Theta(n) = \Omega(2^n) + \Theta(n) = \Omega(2^n)$$

For the second algorithm, the running time is $\Theta(\sum_{i=1}^{n} i) = \Theta(\sum_{i=1}^{n} i) = \Theta\left(\frac{\mathrm{n}(\mathrm{n}-1)}{2}\right) = \Theta(n^2)$

For the third algorithm, there is

$$\mathrm{T}(\mathrm{n}) = T\left(\frac{n}{2}\right) + 8\theta\left(\left(\frac{n}{2}\right)^2\right) + 4\theta(n) + \mathrm{c} = \mathrm{T}\left(\frac{n}{2}\right) + \theta(n^2)$$

Use master theorem, a = 1, b = 2, k = 2. $a < b^k$, so $\mathrm{T}(\mathrm{n}) = O(n^2)$.