Part (a)

```
void f1(int n)
{
    int i=2;
    while(i < n){
        /* do something that takes O(1) time */
        i = i*i;
    }
}
```

CALL 1: i = 4 2^2 2^{2^1}

CALL 2: i = 16 2^4 2^{2^2} ... $n-1 = 2^{2^k}$

CALL 3: i = 256 2^8 2^{2^3}

log is base 2 by default

$n-1 = 2^{2^k}$

$\log(n) = 2^k$

$\log(\log(n)) = k$

$\theta(\log(\log(n)))$

Part (b)

```
void f2(int n)
{
    for(int i=1; i <= n; i++){
        if( (i % (int)sqrt(n)) == 0){
            for(int k=0; k < pow(i,3); k++) {
                /* do something that takes O(1) time */
            }
        }
    }
}
```

runs when i is divisible by sqrt(n)

n=25 5 10 15 20 25

n=16 4 8 12 16

n=9 3 6 9

RUNS sqrt(n) TIMES!

$i^3 - 1$

$$T(n) = \sum_{i=1}^{n} \left[\theta(1) + O\left(\sum_{k=0}^{i^3-1} \theta(1) \right) \right]$$

Worst case inner loop

Common summation

$$\sum_{j=1}^{\sqrt{n}} \sum_{k=0}^{i^3-1} \theta(1) = \sum_{j=1}^{\sqrt{n}} \theta(i^3)$$

$$= \theta(n^{1/2+3})$$

$$\theta(n^{7/2})$$

Part (c)

```
for(int i=1; i <= n; i++){
  for(int k=1; k <= n; k++){
    if( A[k] == i){
      for(int m=1; m <= n; m=m+m){
        // do something that takes O(1) time
        // Assume the contents of the A[] array are not changed
      }
    }
  }
}
```

A[k = i]

Max passes n times

CASE: 1 2 3 4 5 6

n=5

Runs 1 time, Runs 2 times, Runs 3 times ...

Worst Case!

k	m+m
1	2
2	4
4	8
...	...

$n = 2^k$

$\log(n) = k$ ✗

$$\sum_{i=1}^{n} \sum_{k=1}^{n} \theta(1) + \sum_{m=1}^{\log(n)} \theta(1)$$

$$= \sum_{i=1}^{n} \theta(n) + \sum_{k=1}^{n} O\left(\sum_{m=1}^{\log n} \theta(1) \right)$$

Worst case

$$= \theta(n^2) + \sum_{k=1}^{n} O(\theta(\log k))$$

$$= \theta(n^2) + O(\theta(n \log n))$$

$n^2 > n \log n$

$$= \theta(n^2)$$

Part (d)

Notice that this code is very similar to what will happen if you keep inserting into an ArrayList (e.g. vector). Notice that this is NOT an example of amortized analysis because you are only analyzing 1 call to the function f(). If you have discussed amortized analysis, realize that does NOT apply here since amortized analysis applies to multiple calls to a function. But you may use similar ideas/approaches as amortized analysis to analyze this runtime. If you have NOT discussed amortized analysis, simply ignore it's mention.

```
int f (int n)
{
  int *a = new int [10];
  int size = 10;
  for (int i = 0; i < n; i ++)
     {
        if (i == size)
          {
             int newsize = 3*size/2;
             int *b = new int [newsize];
             for (int j = 0; j < size; j ++) b[j] = a[j];
             delete [] a;
             a = b;
             size = newsize;
          }
        a[i] = i*i;
     }
}
```

i	0	1	2	3	4	...	9	10	11
size	10	10	10	10	10		10	10	15
a[i]	0	1	4	9	16		81	100	121

$10 \to 15 \to 22 \to 33$

$10 \quad 10\times1.5^1 \quad 10\times1.5^2 \quad \dots \quad 10\times1.5^k$

$n = 10 \cdot (3/2)^k$

$\frac{n}{10} = (3/2)^k$

$\log_a x = b \Rightarrow a^b = x$

$\log_{3/2}(n/10) = k$

10 is insignificant in complexity?

$$T(n) = \sum_{i=0}^{n} O\left(\log_{3/2} k\right) = \sum_{i=0}^{\log_{3/2}(k)} \sum_{j=0}^{size} \theta(1) = \sum_{j=0}^{\log_{3/2} k} (3/2)^i$$

Worst case

$\theta(n)$

$k = \log_{(3/2)}(n/10)$

$$= 2\left((3/2)^{k+1} - 2\right) = \theta\left((3/2)^k\right)$$

$$= \theta\left((3/2)^{\log_{3/2}(n/10)}\right) = \theta(n/10) = \theta(n)$$

Rules of logarithm

$b^{\log_b k} = k$