Problem 1

a)

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

Ω(n)	Θ(n)	O(n)
if n < 2 loop iterates once	$i_0 = 2 < n$ $i_1 = i_0^2 = 2^2 = 4 < n$ $i_2 = i_1^2 = (i_0^2)^2 = 2^{2*2} = 16 < n$ $i_3 = i_2^2 = (i_1^2)^2 = ((i_0^2)^2)^2 = 2^{2*2*2}$ $= 256 < n$ (...) $i_k = i_{k-1}^2 = 2^{2\wedge k} < n$ terminates at $i_k \geq n$ $2^{2\wedge k} \geq n$ $2^k \geq \log_2 n$ $k \geq \log_2 \log_2 n$	all values of n will follow the pattern in Θ(n) including a sufficiently large enough n to indicate the worst case scenario. ∴ O(n) = Θ(n)
∴ Ω(1)	∴ Θ(lg lg n)	∴ O(lg lg n)

b)

```C/C++
1. void f2(int n)
2. {
3.    for(int i=1; i <= n; i++){                  // O(n)
4.        if( (i % (int)sqrt(n)) == 0){           // early term
5.            for(int k=0; k < pow(i,3); k++) {   // O(i^3)
6.              /* do something that takes O(1) time */ // O(1)
7.            }
```

```
8.        }
9.    }
10.}
```

Ω(n)	Θ(n)	O(n)
if n = 1 outer loop iterates once (1) no early loop term inner loop iterates once (1) ∴ Ω(2)	The if statement in Line 3 within the outer loop acts like an early terminator, allowing us to only consider the outer loops for the lower bound Θ(n) and the upper bound calculated in the next column $O(n^{3.5})$ ∴ $\Theta(n \rightarrow n^{3.5})$	The if statement in Line 3 only runs if the outer loop's counter is a multiple of sqrt(n) Which results on average n / sqrt(n) runs through the inner loop ∴ sqrt(n) ~~The innermost loop in Line 5 runs the outer loop counter to the power of 3 which is sqrt(n) resulting in sqrt(n) * n^3 runtime~~ ~~∴ $n^{3.5}$~~ //edit just realized it $O(i^3)$ i = k sqrt(n) $O(i^3) = O(k^3n^{3/2})$ $\sum_{k=1}^{sqrt(n)} O(k^3n^{3/2})$ $O(n^{3/2})\sum_{k=1}^{sqrt(n)} k^3$ $O(n^{3/2})O(n^{4/2})$ $O(n^{7/2})$ // ended up at the same answer
∴ Ω(1)	∴ $\Theta(n \rightarrow n^{3.5})$	∴ $O(n^{3.5})$

c)

C/C++

```
1. for(int i=1; i <= n; i++){           // O(n)
2.   for(int k=1; k <= n; k++){         // O(n)
3.     if( A[k] == i){                  // early term
4.       for(int m=1; m <= n; m=m+m){   // O(n/2)
5.         // do something that takes O(1) time
```

```
6.          // Assume the contents of the A[] array are not
changed
7.        }
8.      }
9.    }
10.}
```

Ω(n)	Θ(n)	O(n)
If n=1 && A[1] != 1 The program only cycles once ∴ Ω(1)	Line 1 : will run n times Line 2 : will run n times, times outer loop Line 3 : early term, we will take this as the lower bound as $\Theta(n^2)$ and $O(n^3)$ calculated in the next column as our upper bound ∴ $\Theta(n^2 \rightarrow n^3)$	cont. from Θ(n) Assuming the worst case scenario where every value attached to A[k] == 1. The innermost loop will run every time Line 4: will run n/2 times, times n times, times n times ∴ $O(n^3/2)$
∴ Ω(1)	∴ $\Theta(n^2 \rightarrow n^3)$	∴ $O(n^3)$

d)

C/C++

```
1. int f (int n)
2. {
3.   int *a = new int [10];                  // i dont remember if
we are supposed to consider variable initialization in runtime
4.   int size = 10;
5.   for (int i = 0; i < n; i ++)            // O(n)
6.      {
7.        if (i == size)                     // early term
8.          {
9.             int newsize = 3*size/2;
10.            int *b = new int [newsize];
11.            for (int j = 0; j < size; j ++) {b[j] = a[j]}; //
O(3*size/2) added braces for legibility
```

```
12.             delete [] a;
13.             a = b;
14.             size = newsize;
15.           }
16.         a[i] = i*i;
17.      }
18.}
```

Ω(n)	Θ(n)	O(n)
Best case n = 0 where only the initialized variables get run ∴ Ω(1)	Line 5 : will run n times Line 7-15: early term so lower bound n	Line 5 : will run n times Line 7-15: If i == size, then size increases by a factor of 3/2 with a runtime of O(size) size = 10 initially $10\ (3/2)^k \geq n$ $\log_2(3/2)^k \geq \log_2(n/10)$ $k\ \log_2(3/2 \geq \log_2(n/10)$ $k \geq \log_2(n/10) / \log_2(3/2$ $k = \log_2(⅗\ n)$ $k = O\ (\lg n)$ ∴ O(n lg n)
∴ Ω(1)	∴ Θ(n → n lg n)	∴ O(n lg n)

Problem 2

```
C/C++
1. struct Node {
2.     int val;
3.     Node*  next;
4. };
5.
6. Node* llrec(Node* in1, Node* in2) //a.
7. {
8.     if(in1 == nullptr) {
9.         return in2;
10.    }
```

```
11.     else if(in2 == nullptr) {
12.         return in1;
13.     }
14.     else {
15.         in1->next = llrec(in2, in1->next);
16.         return in1;
17.     }
18.}
```

a) What linked list is returned if llrec is called with the input linked lists in1 = 1,2,3,4 and in2 = 5,6?

Answer: in1 = 1, 5, 2, 3, 4 is returned

llrec(in1, in2)	in1 = 1 → 2 → 3 → 4 → nullptr in2 = 5 → 6 → nullptr
Line 8 llrec(in1, in2)	in1 = 1 → 2 → 3 → 4 → nullptr in2 = 5 → 6 → nullptr in1 != nullptr in2 != nullptr in1 → next = llrec(in2, in1→next)
Line 8 llrec(in2, in1→next)	in1 = 1 → 2 → 3 → 4 → nullptr in2 = 5 → 6 → nullptr in2 != nullptr in1→next != nullptr In2 → next = llrec(in1→next, in2→next)
Line 8 llrec(in1→next, in2→next)	in1 = 1 → 2 → 3 → 4 → nullptr in2 = 5 → 6 → nullptr in1→next != nullptr in2→next == nullptr return in1→next
Line 15 llrec(in2, in1→next)	in1 = 1 → 2 → 3 → 4 → nullptr in2 = 5 → 6 → nullptr in2 → next = in1→next

	return in2
Line 15 llrec(in1, in2)	in1 = 1 → 2 → 3 → 4 → nullptr in2 = 5 → ^ in1 → next = in2 return in1
end	in1 = 1 → v in2 = 5 → 2 → 3 → 4 → nullptr in1 = 1 → 5 → 2 → 3 → 4 → nullptr is returned

b) What linked list is return if llrec is called with the input linked lists in1 = nullptr and in2 = 2?

Answer: in2 = 2 is returned

llrec(in1, in2)	in1 = nullptr in2 = 2 → nullptr
Line 8 llrec(in1, in2)	in1 = nullptr in2 = 2 → nullptr in1 == nullptr return in2
end	in1 = nullptr in2 = 2 → nullptr in2 = 2 → nullptr is returned