Problem 1: Runtime Analysis:

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

I is squared in every single iteration and the loop will keep iterating until i is greater than n. The loop stops when $2^{2^{(k-1)}} \geq n$. When we take log of both sides it becomes $\log(2^{k-1}) \geq \log(n)$ and when we take log again on both sides, $k \geq \log(\log(n)) + 1$. This will simplify to $O(\log(\log(n)))$.

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 */
            }
        }
    }
}
```

In the outer loop, the loop iterates from i which starts at 1 until n. Thus, we can assume that it runs n times. There is an if statement that will only run if i is a multiple of the square root of n. We know that the inner loop can execute at sqrt(n), 2sqrt(n), 3sqrt(n), and so forth. When $\sum_{k=1}^{n} = (k * \sqrt{n})^3$, that can ultimately be simplified to $n^{3/2}$ for n and when n is replaced when solving this equation, the n with the highest dominance would be $n^{5/2}$ which can ultimately be determined as the runtime for this function. But we have to factor in the outside loop as well which makes it $O(n^{7/2})$

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
      }
    }
  }
}
```

The outer two loops runs to n as the outermost loop runs from 1 to n times while the second outer loop runs from k to n so when the two loops combined would be at least n^2. If we assume if the if statement executes for most A[k]. The innermost loop iterates as m doubles until it reaches n. Thus the innermost loop iterates at log(n) times. Thus when considering the overall runtime, it can be identified as $O(n^2 \log n)$.

d.
```
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;
     }
}
```

An array is allocated and then the loop runs at around n times. Whenever i is equal to size, the array will then be resized to 1.5 times its old length. Afterwards, the array will be initialized with the square of i. The runtime is most affected by the resizing process whenever i == size which runs at Big Theta(n) as that is the time complexity copying takes.

Problem 2: Linked List Recursion Tracing

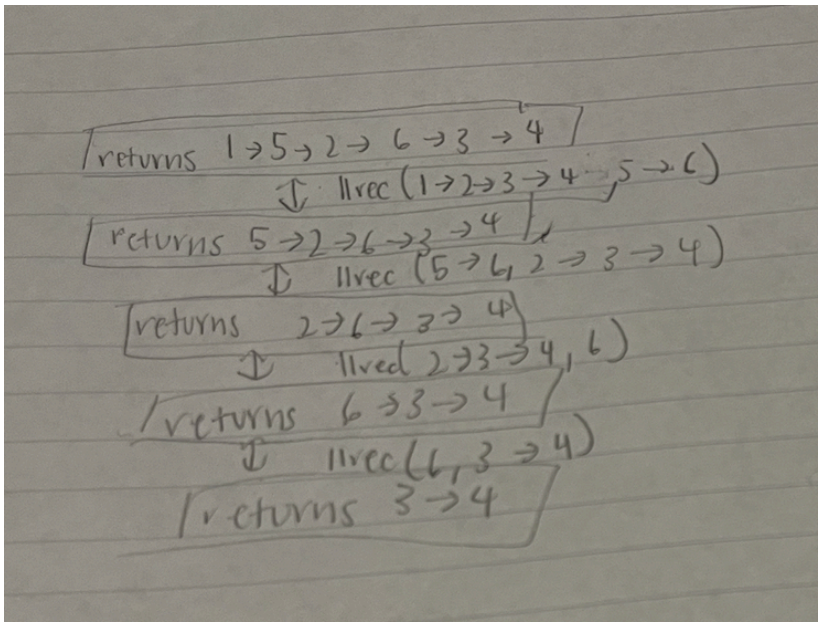
```
struct Node {
    int val;
    Node*  next;
};

Node* llrec(Node* in1, Node* in2)
{
    if(in1 == nullptr) {
        return in2;
    }
    else if(in2 == nullptr) {
        return in1;
    }
    else {
        in1->next = llrec(in2, in1->next);
        return in1;
    }
```

}

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

The function will keep iterating recursively until one of the list reaches null which is when the recursion will start to unwind. In this case, it's when in1 is 3->4, in which the recursion will unroll where the final merged list will be 1->5->2->6->3->4.



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

As in1 is nullptr, the function would return in2 immediately resulting in the linkedlist 2.