a)

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

The time complexity of part a is log(log(n)). The local variable in the while-loop is being multiplied by itself after each iteration. After k iterations, the value of the iterator variable i is a maximum of 2^{2^k}. Therefore, to calculate the time complexity, we can create the equation $2^{2^k} <= n < 2^{2^{(k+1)}}$. This can be put in the equation $2^{2^k} <= c * n$, where c is an arbitrary constant and n is the input size.
$2^{2^k} <= c * n$
$\log_2(2^{2^k}) <= \log_2(c * n)$
$2^k <= \log_2(c * n)$
$\log_2(2^k) = \log_2(\log_2(c * n))$
$k <= \log_2(\log_2(c * n))$
$= O(\log(\log(n))$
Therefore, the time complexity for this program is 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 */
            }
        }
    }
}
```

The time complexity of part b is $O(n^{7/2})$. The if-statement within the for-loop checks if i is a multiple of the square root of n. Since the outer for-loop only occurs n times, the maximum number of times the if-statement is triggered is sqrt(n) times. The inner for-loop iterates from 0 to i^3, and performs an O(1) operation on each iteration. In the worst case scenario, this loop will occur $sqrt(n)^3 + (2*sqrt(n))^3 + \ldots + n^3$, which is a runtime complexity of $O(n^3)$ since n^3 is the biggest term. Therefore, the total time complexity will be $sqrt(n) * n^3$, which equals $O(n^{7/2})$.

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 time complexity of part c is O(n log(n)). The worst case scenario for this function is if the array contains the same values which are all equal to an i value such that 1 <= i <= n. This is the worst case scenario because the contents of the array cannot be changed. If this scenario occurs, then the statement: `if( A[k] == i)` will be true for all k that 1<= k <=n for a singular i value. Therefore, since this only occurs for a singular i value in the outer for loop and n times on the inner loop, the time complexity for the if statement is O(n). However, within the if-statement, there is another for-loop which iterates from 1 to n that performs an operation with O(1). The local variable in the for-loop is added to itself for each iteration, which is also represented by the iterator value being multiplied by 2 for each iteration. The number of iterations will be $2^k <= n < 2^{k+1}$, where k is the number of iterations. This can be put into an equation $2^k <= c * n$, where k is the number of iterations, c is an arbitrary constant, and n is the input size.
$\log_2(2^k) <= \log_2(c * n)$
$k <= \log_2(c * n)$
$k <= \log_2(c) + \log_2(n)$
$k <= O(\log_2(c) + \log_2(n))$
$= O(\log(n))$
Therefore, the total time complexity for the program is n * log(n), which equals O(n 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;
     }
}
```

The time complexity of part d is O(n). The for-loop iterates n times, and there are two operations in the for loop: `a[i] = i*i` and the if-statement: `if (i == size)` . The first operation takes O(1) time and occurs n times, so the total time complexity is O(n) + the time complexity of the if-statement. Within the if statement, there is only one operation that does not have an O(1) time complexity: it is the for-loop: `for (int j = 0; j < size; j ++) b[j] = a[j];` . The operation within the for-loop has an O(1) time complexity. The size starts out at 10, but increases by 1.5x its amount every time the outer iterator equals the size. In the worst case scenario, the size will be 1.5 * n. This means the inner for-loop will have a time complexity of O(n). Therefore, the total time complexity for the program is O(n + n), which equals O(n).