1. To show that $1/n$ is $O(1)$ we must find constants $c > 0$ and $n_0 \geq 1$ such that

$$\frac{1}{n} \leq c, \quad \forall n \geq n_0$$

Let us multiply both sides of the first inequality by $n$ (this can be done since $n$ is positive, as $n \geq n_0 \geq 1$), to get

$$1 \leq cn, \quad \forall n \geq n_0$$

Now we can choose $c = 1$ and the above inequality becomes

$$1 \leq n, \quad \forall n \geq n_0$$

Finally, we can choose $n_0 = 1$, as

$$1 \leq n, \quad \forall n \geq 1.$$  

2. To show that $n$ is not $O(\sqrt{n})$ we use a proof by contradiction. Assume that $n$ is $O(\sqrt{n})$ and derive a contradiction from this assumption. If $n$ is $O(\sqrt{n})$, this means that we can find constants $c > 0$ and $n_0 \geq 1$ for which

$$n \leq c \times \sqrt{n}, \quad \forall n \geq n_0.$$  

Dividing both sides by $\sqrt{n}$, (this can be done since $n \geq n_0 \geq 1$, so $\sqrt{n}$ is positive) we get

$$\sqrt{n} \leq c, \quad \forall n \geq n_0,$$

which clearly cannot hold, since $\sqrt{n}$ grows without bound, so $\sqrt{n}$ cannot be upper-bounded by any constant $c$. Therefore, $n$ is not $O(\sqrt{n})$.

3. Note that since $f(n)$ is $O(g(n))$ then we know that constant values $c' > 0$ and $n_0' \geq 1$ exist such that

$$f(n) \leq c'g(n), \quad \forall n \geq n_0'.$$  

To show that $k(f(n) + g(n))$ is $O(g(n))$, we need to find constants $c > 0$ and $n_0 \geq 1$ such that

$$k(f(n) + g(n)) \leq cg(n), \quad \forall n \geq n_0.$$  

Using (1) in the left hand side of the above inequality we get

$$k(f(n) + g(n)) \leq k(c'g(n) + g(n)), \quad \forall n \geq n_0'.$$  

Simplifying the right hand side we get

$$k(f(n) + g(n)) \leq (k(c' + 1))g(n), \quad \forall n \geq n_0'.$$

Hence, if we choose $c = k(c' + 1)$ and $n_0 = n_0'$ inequality (2) holds, and since $c$ and $n_0$ are constants we have shown that $k(f(n) + g(n))$ is $O(g(n))$. 


4 (i) **Algorithm** IsBalanced($A, n$)

**In:** Array $A$ storing $n$ different integer values

**Out:** True if $A$ is balanced; false otherwise

```plaintext
{ 
for $i \leftarrow 0$ to $n - 1$ do {
    inverse_found \leftarrow false
    for $j \leftarrow 0$ to $n - 1$ do {
        if $(i \neq j) \text{ and } (A[j] + A[i] = 0)$ then // Additive inverse for $A[i]$ found
            inverse_found \leftarrow true
    }
    if inverse_found = false then
        return false // $A[i]$ has no additive inverse in $A$ so $A$ is not balanced
}
return true // Every $A[i]$ has its additive inverse in $A$ so $A$ is balanced
}
```

(4a) The outside loop is repeated once for each value of $i$ between 0 and $n - 1$. Hence, the outside loop is repeated $n$ times. The inside loop is repeated $n$ times for each iteration of the outside loop. Therefore, the total number of iterations of the inside loop is at most $n^2$. The algorithm then terminates after a finite amount of time.

(4b) The first loop considers all values $i \in \{0, 1, \ldots, n - 1\}$. For each one of these values, the second loop considers all values $j \in \{0, 1, \ldots, n - 1\}$ looking for a value $j \neq i$ such that $A[j] + A[i] = 0$ (first if statement of the algorithm). This value $A[j]$ is called the additive inverse of $A[i]$. Hence, if the additive inverse of $A[i]$ is in $A$ the algorithm will set the value of inverse_found to true, otherwise the value of that variable will be false.

Observe that if for every value $A[i]$ the algorithm finds its additive inverse in $A$ then the array is balanced and in that case the condition of the second if statement of the algorithm is always false causing the algorithm to return the value true as required. However, if for some value $A[i]$ its additive inverse is not in $A$, then the second loop of the algorithm will never set inverse_found to true and so the condition of the second if will be true causing the algorithm to return false as in that case $A$ is not balanced.

4 (ii) The worst case for the above algorithm is when the array is balanced as in this case the condition of the second if statement is always false causing the first for loop to perform the maximum number of iterations.

We analyze the inner-most for loop first. In every iteration of this loop a constant number $c'$ of operations is performed and the loop is repeated $n$ times. Thus, the total number of operations performed by this loop is $c'(n)$.

As for the first for loop, in every iteration it performs a constant number $c''$ of operations to set inverse_found to false and to check the condition of the second if statement plus the $cn$ operations of the second for. Since the first loop is repeated $n$ times in the worst case, the total number of operations performed by the algorithm is $n(c' + cn) = c'n + cn^2$. Outside the first loop a constant number $c'''$ of operations are performed, so the total number of operations performed by the algorithm is $c'' + c'n + cn^2$ is $O(n^2)$. 

2
Another Solution

The following algorithm avoids iterating over the entire array once the additive inverse of a value $A[i]$ has been found.

**Algorithm IsBalanced($A, n$)**
*In: Array $A$ storing $n$ different integer values*
*Out: True if $A$ is balanced; false otherwise*

```plaintext
{ 
  for $i \leftarrow 0$ to $n - 1$ do { 
    inverse_found $\leftarrow$ false 
    $j \leftarrow 0$ 
    while ($j \leq n - 1$) and (inverse_found = false) do { 
      if ($i \neq j$) and ($A[j] + A[i] = 0$) then // Additive inverse for $A[i]$ found 
        inverse_found $\leftarrow$ true 
      else $j \leftarrow j + 1$ 
    } 
    if inverse_found = false then 
      return false // $A[i]$ has no additive inverse in $A$ so $A$ is not balanced 
  } 
  return true // Every $A[i]$ has its additive inverse in $A$ so $A$ is balanced
}
```

A similar proof of correctness as for the first version of the algorithm can be used for this new algorithm. However, the time complexity of this second algorithm is a bit more complicated to compute. The worst case is as before, when the array $A$ is balanced, as then the condition of the second if statement is always false so the algorithm does not terminate before the first loop scans the entire array. As for what values to store in $A$ and in which positions to store them, it does not matter as long as $A$ is balanced.

Assume that $A$ is balanced. For each value $A[i]$, let $p(i)$ be the position of its additive inverse. Since all values in $A$ are different then all values $p(i)$ are also different, i.e. the values of $p(i)$ are $0, 1, \ldots, n - 1$; this will be important later in the analysis.

To compute the time complexity of the algorithm in the worst case we analyze the second loop first. Each iteration of the loop performs a constant number $c$ of operations. For value $A[i]$ the second loop performs $p(i)$ iterations, so the total number of operations that the second loop performs for value $A[i]$ is $cp(i)$.

During its $i$-th iteration, the first loop performs a constant number of operations $c'$ outside the second loop plus $cp(i)$ operations in the second loop; thus the total number of operations is $c' + cp(i)$. As the first loop performs one iteration for each $i \in \{0, 1, \ldots, n - 1\}$, the total number of operations that this loop performs is

$$\sum_{i=0}^{n-1} (c' + cp(i)) = \sum_{i=0}^{n-1} (c' + ci) = c'n + cn(n - 1)/2$$

the first equality holds because the values of $p(i)$ as stated above are $0, 1, \ldots, n - 1$. Finally, a constant number $c''$ of operations are performed outside the first loop. Thus, the total number of operations performed by the algorithm is $c'' + c'n + cn(n - 1)/2$ is $O(n^2)$. 

3