1: Big Oh

(a) $f(n) = n^2 + 5n + 20$.
The largest growing term in $f(n)$ is n^2. In other words, the asymptotic behavior of $f(n)$ is n^2. Therefore, the function is $O(n^2)$.

(b) $f(n) = 2 \log n + 4$.
The function is $O(\log n)$

(c) $f(n) = n^{-2} + 2n^{-1}$.
The function is $O(n^{-1})$

(d) $f(n) = 1 + n^{-2}$.
The function, when $n \to \infty$, will be a constant. Therefore, the function is $O(1)$.

(e) $f(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$.
The bound for $f(n)$ is between $\ln n$ and $\ln n + 1$. Therefore, the function is $O(\log n)$.

2: Removing Duplicates

Algorithm: : Initially, sort the array A and store the resultant array as P. Initialize an empty array B. Iterate over all the elements in P and check if the current element is equal to the previous one (skip this step for first element of the array). If the element is not equal to the previous value or if it's the first element, add it to the array B, or else move to the next element.

Algorithm 1 Removing Duplicates

```
procedure $remove\_duplicates(A)$
    $P = merge\_sort(A)$
    Initialize B as an empty array
    for $i = 1$ to $N$ do
        if $i \neq 1$ then
            if $P[i] \neq P[i-1]$ then
                $B.add(P[i])$
            end if
        else
            $B.add(P[i])$
        end if
    end for
    return B
end procedure
```

Correctness: : The array P is sorted version of the array A and contains all the elements of A. Also, the array P contains all the equal elements placed adjacent to each other. The array B is constructed from P by picking up either the element itself if it has no duplicate copy or exactly one element from a group of same elements.

Running time: The *merge_sort* routine in the algorithm takes $O(n \log n)$ time. The *for* loop iterates over the size N. Therefore, the running time is $O(n \log n + n) = O(n \log n)$.

3: Square vs Multiply

Let a and b are the two n- digit numbers that we need to multiply. Let *square(x)* is the procedure that squares a n-digit number, x, in $O(n \log n)$ time. We need to find an algorithm, *multiply(a,b)*, that can multiply a and b in $O(n \log n)$ time.
Consider, $ab = \frac{a^2+b^2-(a-b)^2}{2}$. Based on this result, we can devise our algorithm *multiply(a,b)*.

Algorithm 2 Multiply

```
1: procedure multiply(a, b)
2:     x = square(a)
3:     y = square(b)
4:     z = square(a − b)
5:     d = x + y − z
6:     P = d/2
7:     return P
8: end procedure
```

Running time: The run time for the operations in steps 2, 3, and 4 is $O(n \log n)$. The division operation can be performed by a shift right operation which is a constant time operation. Therefore, the run time for the procedure *multiply(a,b)* is $O(n \log n)$.

4: Basic Probability

(a) Total number of possiblities for the experiment is 2^k. The number of outcomes with exactly one H can be k (only first toss is H, only second toss is H, ..., only k^{th} toss is H). Therefore, the required probabilty is $\frac{k}{2^k}$

(b) Total number of possible colorings of the boxes can be k^k. The reason is that the first box can be colored with any of $\binom{k}{1}$ colors, the second box can also be colored with any of $\binom{k}{1}$ colors. In this way, the total number of possibilites are $\binom{k}{1} \cdot \binom{k}{1} \cdot \ldots k \; times \cdot \ldots \binom{k}{1} = k^k$.

Now for boxes to be colored uniquely, let's say the first box is colored with any of $\binom{k}{1}$ colors, then second box can be colored with any $\binom{k-1}{1}$ colors, and finally the last can only be colored with one remaining color, i.e. $\binom{1}{1}$. Total possibilities for boxes to be colored uniquely is $\binom{k}{1} \cdot \binom{k-1}{1} \cdot \ldots k \; times \cdot \ldots \binom{1}{1} = k!$. Hence, the required probability is $\frac{k!}{k^k}$

5: Array Sums

Let us describe an $O(n^2 \log n)$ time algorithm.

Algorithm: : Initially, we sort the array A to get the array P. Now we iterate over all the possible pairs of elements in the array P. During these iterations we look for a third element which is equal to the difference of the pair. If such an element exists, we have found our required triplet.

Algorithm 3 search

```
1: procedure sum_search(A)
2:     P = merge_sort(A)
3:     for $i = 1$ to $N$ do
4:         for $j = 1$ to $N$ do
5:             $k = binary\_search(P, P[i] - P[j])$
6:             if search succesful then
7:                 print(i,j,k)
8:             end if
9:         end for
10:    end for
11: end procedure
```

Correctness: : P is a sorted version of A and contains all the elements of A. After fixing i and j from the array P, we search for k such that $P[i] - P[j] = P[k]$. If such a k exists, the binary search is successful and we have found a triplet, otherwise there does not exist a k for the chosen i and j such that the condition holds.

Running time: : The merge sort takes $O(n \log n)$ time. The double for loop along with the binary search in each iteration will take $O(n^2 \log n)$ time (as binary search takes $O(\log n)$ time). The asymptotic behavior for this algorithm is, therefore, $O(n^2 \log n)$.