1. R-4.8, p. 182

Order the following functions by asymptotic growth rate.

$4n\log n + 2n$, 2^{10}, $2^{\log n}$,
$3n + 100\log n$, $4n$, 2^n,
$n^2 + 10n$, n^3, $n\log n$

2^{10} - is O(1), constant runtime.
$2^{\log n}$ - is n, due to the rules of logarithms. n is O(n), which is comparatively slower than O(1)
4n - is O(n), but the constant 4 makes it slightly slower than the above
3n + 100logn - is also O(n), but with the addition of logn
nlogn - is O(nlogn), comparatively slower than O(n)
4nlogn + 2n - is also O(nlogn), but with the addition of 2n
$n^2 + 10n$ - is $O(n^2)$, comparatively slower than O(nlogn)
n^3 - is $O(n^3)$, comparatively slower than $O(n^2)$
2^n - is $O(2^n)$, comparatively slower than $O(n^3)$

2. R-4.12, p. 182

Give a big-Oh characterization, in terms of n, of the running time of the example4 method shown in Code Fragment 4.12.

O(n). The loop runs through each element once and only once, without breaking before the end.

3. R-4.18, p. 184

Show that if d(n) is O(f(n)) and e(n) is O(g(n)), then d(n) − e(n) is not necessarily O(f(n) − g(n)).

Let f(n) = 2n + nlogn
d(n) = O(f(n)) = n
Let g(n) = 2n
e(n) = O(g(n)) = n
Therefore, d(n) - e(n) = 0
However, O(f(n) - g(n)) = O(2n + nlogn - 2n) = O(nlogn) != 0
Hence, d(n) - e(n) is not always O(f(n) - g(n))

4. Consider $f(n) = 5n^2+ 4n-2$, mathematically show that f(n) is $O(n^2)$, $\Omega(n^2)$, and $\Theta(n^2)$.

$5n^2+4n-2 <= cn^2$ for c = 11, when $n >= n_0 = 1$, thus f(n) is $O(n^2)$
$5n^2+4n-2 >= cn^2$ for c = 1, when $n >= n_0 = 1$, thus f(n) is $\Omega(n^2)$
$5n^2+4n-2$ is $O(n^2)$ and $\Omega(n^2)$, and so is $\Theta(n^2)$.

5. For finding an item in a sorted array, consider "tertiary search," which is similar to binary search. It compares array elements at two locations and elminiates 2/3 of the array. To analyze the number of comparisons, the recurrence equations are T(n) = 2 + T(n/3), T(2) = 2, and T(1) = 1, where n is the size of the array. Explain why the equations characterize "tertiary search" and solve for T(n).

kk

6. To analyze the time complexity of the "brute-force" algorithm in the programming part of this assignment, we would like to count the number of all possible strings.

(a) Explain the number of all possible strings in terms of n(maximum length of a string).

The maximum number of possible strings will be the max combinations for each length up to the maximum. In pseudocode:

```
for (i = 0; i < n; i++)
    max = (max * 26) + max
```

If n is 1, there are 26 combinations. If n is 2, there are 26 + (2 * 26) combinations, and so on.

(b) Consider a computer that can process 1 billion strings per second and n is 100, explain the number of years needed to process all possible strings.

if n is 100, then the maximum number of strings would be 8,116,567,392,432,202,710. At 1 billion (1,000,000,000) strings per second, it would take

~8,116,567,392,432 seconds = 2,254,602,053 hours
~= 93,941,752 days
~= 257,374 years, not counting leap years.

In that many days, there would have been hundreds of thousands of leap years, which would add thousands more years to the total.

(c) If we don't want the computer to spend more than 1 minute, explain the largest n the computer can process.

The largest n the computer would be able to process would be 7, at 835,308,258 strings. This would take less than a second, but adding one more to n would give 217,180,147,158 strings, which would take almost 4 minutes at 1bil words/second.