Shrey Joshi

Net ID: SXJ210081

1) n_0 here denotes the intersection of $T(N)$ and $f(N)$. It's important to include $N >= n_0$ because when dealing with big-O notation, we care about the asymptotic behavior as $x \longrightarrow \infty$. $T(N)$ might be greater than $f(N)$ locally but beyond some $N = n_0$, $f(N)$ is the upper bound.

2) They are both $O(N)$ since they differ only by a constant. In the definition $T(N) <= cf(N)$, the constant c would be 2 for $2n$ and 3 for $3n$.

3) When the input to an $O(N)$ algorithm is doubled, running time is doubled. When the input to an $O(N * N) = O(N^2)$ algorithm is doubled, running time is quadrupled.

4) By measuring the Big-O notation for certain functions and segments of code in a program, one can A) find the total running time complexity of the program, which tells you how well it scales, and B) identify the segments of code that contribute the most to running time complexity so one can prioritize optimizing those. Ex. in an $O(N^2)$ program it's better to reduce the only function that is $O(N^2)$ to $O(N)$ rather than reduce another function that is $O(N)$ to $O(1)$

5) $n!$ grows faster than 2^n. Consider $n = 4$.

$2^n = 2 \cdot 2 \cdot 2 \cdot 2$

$n! = 4 \cdot 3 \cdot 2 \cdot 1.$

$2^n = 2 \cdot 2 \cdot 2 \cdot 2 < 2 \cdot 2 \cdot 3 \cdot 2 = n!$

Looking at the two expressions element-wise, we see that for any $n > 4$, the LHS is scaled by 2 and the RHS is scaled by n for every increment of n. Thus $n!$ grows faster than 2^n.

6)

a) $4n^5 + 3n^2 - 2 = O(n^2)$

b) $5^n - n^2 + 19 = O(2^n)$

c) $(3/5) * n = O(n)$

d) $3n * log(n) + 11 = O(n)$

e) $[n(n+1)/2 + n]/2 = O(n^2)$

7) There is one loop that iterates n times, so it is $O(n)$

8) There are two nested loops, each running n times, so it is $O(n^2)$

9) There are two loops, each with time complexity $O(n)$, so it is $O(n^2)$

10) In the worst case, num < numItems, where the loop that iterates $O(n)$ times run. Thus the overall time complexity is $O(n)$.

11) Here we are repeatedly dividing n by 2 until we get 0. This is inherently a logarithmic process, so the time complexity is $O(\log(n))$

12) This is a recursive function that repeatedly divides n by 2 until we get 0, same as (11) except recursive instead of iterative. Again, this is a logarithmic process so the time complexity is $O(\log(n))$.