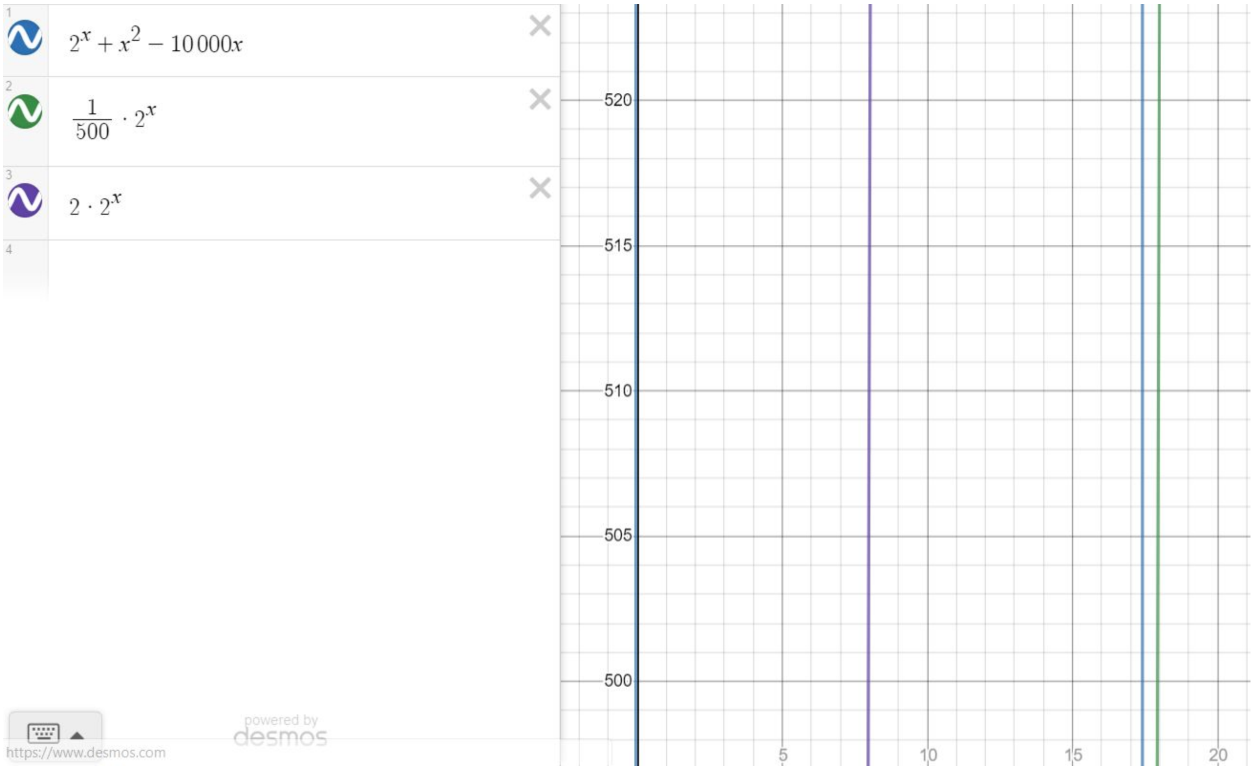
Sarah Yurick
sey13

1. See each below.
    a. The Big-O time for breadth-first search using an adjacency matrix is $O(V^2)$, where V is the number of vertices. This is because since an adjacency matrix is essentially an V x V matrix of ones and zeros depending on which vertices have edges between them, in order to find the neighbors of each vertex it must go through the entire V x V, which is V^2, matrix. In addition, processing each node has a Big-O time of O(V), so the total Big-O time is $O(V^2 + V)$, which is simply $O(V^2)$.
    b. The Big-O time for breadth-first search using an adjacency list is O(E + V), where E is the number of edges and V is the number of vertices. This is because since an adjacency list uses links for each edge, finding the neighbors of a vertex has a time complexity of O(E). In addition, we assume that each vertex (or at least the majority of the vertices) will be processed, which has a Big-O time of O(V).
    c. The Big-O time for depth-first search using an adjacency matrix is $O(V^2)$, where V is the number of vertices, for the same reason as explained in part a.
    d. The Big-O time for depth-first search using an adjacency list is O(E + V), where E is the number of edges and V is the number of vertices, for the same reason as explained in part b.

2. See each below.
    a. The Big-O time for Dijkstra's shortest path algorithm is $O(V^2)$, where V is the number of vertices. This is because Dijkstra's algorithm uses nested for loops, which each repeat V-1 as it searches for the nearest neighbor and V times for each vertex, yielding a total complexity of $O(V^2)$.
    b. Yes, the answer does depend on whether we are using an adjacency matrix or list. The answer given in part a is for an adjacency matrix. Using an adjacency list, however, would have a different Big-O time because the inner loop of traversing the graph would be more similar to breadth-first search, which has a Big-O time of O(E + V) as explained in 1b. You could use a minimum heap to have the neighbors already sorted by the nearest, but heap sort due to the heapifying process has a time of O(logV). Thus, the complexity of the inner loop of O(E + V), while removing from the heap has complexity O(logV), so the total Big-O time is O((E + V) * log V), which simplifies to O(E logV).

3. See each below.
    a. By trial and error, I found that the values can be c1 = 1/500, c2 = 2, with n0 = 17.5. Thus, there are constants c1 and c2 which, when multiplied by g(n), are upper and lower bounds of f(n). Therefore, the Θ time is f(n) = Θ(g(n)).

b.



4. Quicksort is O(N²) for the worst case and Ω(N logN) for the best case. Thus, there is not a Θ time for quicksort because the worst case and best case scenarios, and the upper and lower bounds, are not the same.