CS 205: Final Exam Question 2 - Big O

16:198:205

1) Formally prove that if $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.

2) True or false - give a mathematical justification:

   a) $n = O(n^2)$.

   b) $n(n+1)(n+2) - n^3 = O(n^3)$.

   c) $n(n+1)(n+2) - n^3 = O(n^2)$.

   d) $n \ln n = O(n^2)$.

   e) $n^2 = O(n \ln n)$.

   f) $1/n = O(1)$.

   g) $1000000n = O(n)$.

   h) $2^n = O(3^n)$.

   i) $3^n = O(2^n)$.

   j) $\sum_{i=1}^{n} i(i+1)(i+2) = O(n^4)$.

Recall the idea of **mergesort**: to sort a list, divide a list in two, sort the two halves, and merge them to form a sorted whole. In class, we gave an argument that the complexity of mergesort on a list of N elements could therefore be described as

$$M(N) = \underbrace{M(N/2)}_{\text{sort the left half}} + \underbrace{M(N/2)}_{\text{sort the right half}} + \underbrace{N}_{\text{to merge the two halves}} = 2M(N/2) + N. \qquad (1)$$

Noting that $M(1) = 1$, since sorting a list of size 1 is easy, this led to an overall complexity of $M(N) = O(N \ln N)$. Your good buddy suggests the following: If mergesort gets such good performance dividing the list into two halves and merging them, imagine how fast a mergesort would be that split the list to sort into *three* parts, sorted them, then merged the result.

3) Like the recursive relation above, give a rough description of the overall worst case complexity of this tri-mergesort.

4) In terms of big-O, which approach has the smaller complexity?

5) In your opinion, is it worth the additional effort and overhead it would take to implement this approach? Justify.