CSE 5311 Lab Assignment 4

Due August 6, 2003

Goals:

1. Review of ‘‘merging’’.

2. Understanding of approximation for an NP-hard problem.

Requirements:

1. Write (and test) a program that approximates a *subset sum* (CLRS, sections 34.5.5 and 35.5) for a set of positive integers and a range `s . . . t` of acceptable values. The first line of the input (read from `stdin` or `cin`) will contain the number `n` of positive integers in the set and the lower and upper bounds for the range. The remaining `n` lines will contain the elements of the set without ordering and possibly with repeated values. `n` will not exceed 20.

   The output will be a subset of the input set whose sum is in the range `s . . . t`, along with the actual sum achieved. In addition, your program should indicate the amount of space and time used.

   Your program must compile and execute on OMEGA. There should be a comment near the beginning of your code that indicates how to compile on OMEGA. Your debugging trace should be disabled in the version you submit. If $n \leq 5$, then all intermediate lists should be output.

2. Email your code to `yxb4544@omega.uta.edu` before 3:00 pm on August 6. The subject should include your name as recorded by the University.

Getting Started:

1. You should sort the input.

2. Even though you will not submit it, it is useful to first implement `EXACT-SUBSET-SUM` on page 1045. This handles the case where `s` = `t`. Your version should differ from the book’s in several ways:

   a. You will not need a separate function to `MERGE-LISTS`. By having two pointers to the list from the previous round, you should be able to generate the new list while assuring that duplicates are removed and stop (the present round) when a value larger than `t` is generated.
   b. Along with each list element you should include an indication of the last value ‘‘added in’’ to achieve that value.
   c. You will have code to actually output the subset.

3. Your approximation code will be similar to the preceding exact version, except that the list elements will be *intervals* rather than distinct values. In particular, instead of using 0 in the initial list you will use `0 . . . t - s`. This initialization allows your code to stop when an interval that includes `t` has been generated. The most significant change is to handle overlapping intervals. If two intervals overlap and have the same last value ‘‘added in’’, they must be combined. Each list may have touching intervals, but none that overlap. No interval should have values larger than `t`.