**CS 201 Section 1 Homework 2 Suggested Solutions**

1. (7.1) Programming time for (a) is 100 man months, for (b) is 1000 man months, and for (c) is $100 + 10 \times (1\% \times 100) = 110$ man months. If execution time for (a) is $x$, then time for (b) is $x/4$ and time for (c) is $x/2 + (x/2)/4 = 5x/8$.

2. (7.6) This was a hard problem, even if you figured what the question was really asking!

The idea here is that you are an employee of Dr. Scrooge, forced to implement his decision about how to represent symbol. By "symbol" the problem means "symbol name" rather than "symbol value". The question is: if symbol names consist only of letters (excluding Q) and they must be made to fit in 48 bits, how many letters long can they be, and how will these letters be represented and packed into the 48 bits?

To simplify things, let's assume that we have only upper-case letters, so that we're trying to represent 25 different characters, and that all symbols will be the same (maximum) length, so that we don't need to worry about marking the end of a symbol. (You were free to make other assumptions as long as you stated them.)

The most obvious approach is to use a standard ASCII encoding, requiring 8 bits per character, which allows symbols of length $48/8 = 6$, with the usual packing scheme. But we can do much better.

Suppose first that we want to use a bit-based scheme for packing and unpacking characters. That is, each character value should represented by a number that fits in a fixed number of bits. The obvious scheme is to represent `A` by 0, `B` by 1, and so on. Since we have 25 different values in the range $[0, 24]$, we will evidently need 5 bits, since $2^4 < 25 \leq 2^5$. The encoding scheme is simply to concatenate all the bits together. With this approach, we can store $\lfloor 48/5 \rfloor = 9$ values in a word, leaving 3 bits unused. This is an acceptable solution.

But note that it leaves $32 - 25 = 7$ values of each character position unused. We can do better still by abandoning bit-based packing schemes and using *arithmetic coding*. The basic idea, outlined in class lecture, is to encode values in base 25. That is, the sequence "ADB...," corresponding to values "143..." will be represented by the unique number

$1 \times 25^0 + 4 \times 25^1 + 3 \times 25^2 + \ldots$

Obviously quite a lot of arithmetic will be needed to pack and unpack characters!

How many characters can we fit in a word? Each character will require $\log_2 25 \approx 4.6$ bits, so we can accommodate $\lfloor 48/4.6 \rfloor = 10$ characters in all. This is the best we can do without knowing more about the probability distribution of characters.

Why all the business about omitting `Q`? I honestly don't know, since $\log_2 26 \approx 4.701$, so an (unsigned) word could also hold 10 characters from the full 26 character alphabet. (In fact, 27 character alphabets would also work.) Perhaps Tanenbaum got his arithmetic wrong.

3.(7.8) The instruction location counter is a static counter that indicates where the next instruction should be placed in memory. The program counter is a dynamic pointer that indicates which instruction should be executed next. To see that these can be quite different, consider what happens after a jump instruction: the ILC points to the next address, whereas the PC points to the target of the jump.

4. (7.19) The point here is that a library procedure can itself call library procedures that also need to be linked in to the final program. Naively, we might try simply adding any such procedures to our list of names, so that they too will be selected from the library when encountered in the linear scan. But this may fail because we might have already passed (and skipped) some of these procedures.

Concretely, suppose that the library is sorted in alphabetical order, and that library procedure B calls library procedure A (which is not itself called directly by the user program). The linear scan of the library will first encounter A, and skip it because it is not on the search list for the user program. Then it will encounter B, include it, and add B's externals – including A – to the search list. But as we scan the remainder of the library, A will not be encountered again to consider for inclusion.

A partial solution is to order the library so that callers always proceed callees; e.g., B must be placed before A. Then linear scanning works fine unless there is a *cycle* in the call graph, e.g., B also calls A, and the user program calls A or B, but not both. To handle this case, we must be prepared to repeat the linear scan if symbols remain unresolved after one scan; many systems do this, since a single scan often Another alternative is to use a non-sequential access method to fetch procedures out of the library.

7.23. The answer depends on the starting address for programs. If we assume this is 100, as in the text, the the relocation constants are: 100,300,1100,1700,2200.