ECE220: Computer Systems & Programming **Fall 2023 ZJUI**
Machine Problem 4 **due: Saturday 28 October, 11:59:59 p.m.**

Choosing Postage Stamps

Your task this week is to solve an optimization problem and print the solution. The optimization problem involves selecting the correct combination of "postage stamps" to affix to a package in order to exceed a required amount. Specifically, given a required postage cost and four different stamp values, write a C function that selects the best combination among 4 types of stamps. The "Best" is defined as a combination of stamps with total value of at least the postage cost, but otherwise minimal. If ties in the total value exist with different combinations of stamps, your function must select the combination with the smallest number of stamps. If ties are still possible, your function must select the combination with the largest number of stamps with the highest value (then maximize the number of stamps with second-highest value, and so forth). Abstractly, this problem can be viewed as an algorithmic problem, an optimization problem, and in some cases a number theory problem. But don't spend too much time worrying about the math: the objective for this week is for you to gain some experience with control structures in C, particularly with loops and conditional statements.

Background

Finding the right combination of stamps is an optimization problem. But why is this problem challenging? In practice, it's usually not. Before electronic commerce, merchants solved this problem thousands of times a day in their heads. To make the problem easier, monetary systems (both coins and bills) use specific combinations of values, always including the basic unit of currency. Similarly, selecting postage stamps has also been fairly straightforward, although the reasons for the simplicity are not as good, at least in the U.S., where postage costs are always multiples of the current large stamp value, and the post office issued numerous small stamps for those with outdated "large" stamps when prices rose.

Let's start with some simple examples, using stamp values typical of monetary systems, such as 10, 5, 2, and 1. With these values, the problem can be solved using a greedy approach, in which we start with the biggest value and work down towards the smallest, choosing the number of stamps of each type. For a cost of 79, for example, we first observe that we can use 7 stamps of value 10, leaving 9 (=79 – 10×7) more to cover. Moving to the stamp of value 5, we decide to use 1, leaving 4 more to cover. Adding 2 stamps of value 2 then brings our total to 79, and we are done, having used 0 stamps of value 1. Using a correct version of our program, we would type the command on the left and see the answer as shown on the right:

`./stamps 79 10 5 2 1` prints `7 1 2 0 -> 79` (and a newline).

Greedy algorithms are easy to write and often produce reasonable results, but they don't always solve the problem precisely. Imagine that our stamp values are now 50, 25, 10, 1, and that we want to produce the value 30. Following a greedy approach, we select 0 stamps of value 50, 1 stamp of value 25, 0 stamps of value 10, and 5 stamps of value 1, for a total of 6 stamps. If we instead use 3 stamps of value 10, however, we cover the cost of 30 using only 3 stamps! A correct version of our program, of course, must produce the correct answer:

`./stamps 30 50 25 10 1` prints `0 0 3 0 -> 30` (and a newline).

Similar problems arise when the smallest stamp value is larger than 1. Consider the stamp values 50, 25, 10, and 3. What if the amount needed is 17? A greedy approach gives 1 stamp of value 10 and 3 stamps

of value 3, totaling 19. In fact, obtaining a value of 17 is not possible with these stamp values, but neither is 19 the best choice: by using 6 stamps of value 3, we reduce the total cost to 18, even though we end up using more stamps. And, again, a correct program must find this answer:

`./stamps 17 50 25 10 3` prints `0 0 0 6 -> 18`
`(exact match not possible)` (+newline)

Note the extra line of text, which is printed by the code given to you based on the return value from your function.

Ties in both cost and number of stamps may seem impossible at first, but they are actually fairly easy to find. Consider, for example, a value of 5 with stamp values 4, 3, 2, and 1. With a correct program, we observe:

`./stamps 5 4 3 2 1` prints `1 0 0 1 -> 5` (and a newline).

For some combinations of stamp values, we can solve the problem using modular arithmetic, making it more of a number theory problem. For example, consider stamp values 30, 15, 10, and 6. If we choose A, B, C, and D of each, respectively, we obtain total value V = 30A + 15B + 10C + 6D. If V is the desired value, we must have (V = B) mod 2, (V = C) mod 3, and (V = D) mod 5. And then A must be maximized by converting multiples of the smaller stamps into those with value 30. For example, if we want a total of 995, we have A = 32, B = 1, C = 2, and D = 0.

Sometimes problems that are easy to solve in practice are tricky to solve in general.

The Task

You must write the C function specified by the following:

```
int32_t print_stamps (int32_t amount, int32_t s1, int32_t s2,
                      int32_t s3, int32_t s4);
```

The first parameter passed to **`print_stamps`** is the amount of postage needed, which will always be positive. The other four parameters are the stamp values in decreasing order. All stamp values are positive, and are strictly decreasing (no two values will be the same).

Your C function must determine the combination of stamps that

- has total value no less than the required **`amount`**,
- but otherwise has minimum total value.

If multiple combinations exist with the minimum total value, your function must choose the combination that uses the minimum total number of stamps. If ties in both value and number are possible, your function must choose the combination with the largest number of the most valuable stamp (**`s1`**). If ties are still possible, maximize the number of the second most valuable stamp (**`s2`**), and so forth.

For the chosen combination, your function must print the four numbers of stamps chosen in decimal, separated by spaces, followed by " **`->`** " (four characters, including a leading and trailing space), the total value of the chosen combination in decimal, and a newline ("**`\n`**"). Printing should be easy with **`printf`**, but your format must match ours exactly to receive credit.

Finally, your function must return 1 if the value of the chosen combination matches the required **`amount`** exactly, or 0 otherwise.

The "**`(exact match not possible)`**" line is not printed by your function, but rather by the code provided to you based on the return value from your function. **Do not print that line in your code!**

The intent of this assignment is not for you to invent clever algorithmic approaches. Simply use loops (iterations) to consider every reasonable combination of stamps, and use conditional statements to identify the best choice. If you design your loops in the right way, your code will break some ties naturally, and your conditionals will be simpler. If you design your loops more simply, your conditionals must check all of the tie-breaking conditions. Either approach is acceptable: again, we just want you to learn how to write loops and conditionals and to use variables to solve a problem. Computers are fast, so making them do a little more work to save humans some thinking time is usually the right answer.

Pieces

Your program will consist of a total of three files:

`mp4.h` This header file provides function declarations and a brief description of the function that you must write for this assignment.

`mp4.c` The main source file for your code (you must write it yourself). Include the **`mp4.h`** header file and be sure that your function matches the one in the header.

A second file is also provided to you:

`main.c` A source file that interprets commands and calls your function.

You need not read this file, although you are welcome to do so.

We have also provided you with a correct version of the executable, called **`gold`**. You can use this program to generate additional test cases as well as exact output against which to compare your own program.

Specifics

You should read the description of the function in the header file before you begin coding.

- Your code must be written in C and must be submitted as a file named **`mp4.c`**. in the **`mp/mp4`** subdirectory of your repository. We will NOT grade any other files. **Changes made to any other files WILL BE IGNORED during grading.** If your code does not work properly without such changes, you are likely to receive 0 credit.
- You must implement the **`print_stamps`** function correctly.
  - You may assume that the required amount is positive.
  - You may assume that all stamp values are positive.
  - You may assume that stamp values are unique (not equal) and given in decreasing order: **`s1 > s2 > s3 > s4 > 0`**.
- Your routine's return values and outputs must be correct.
- Your code must be well-commented. Follow the commenting style of the code examples provided in class and in the textbook.

Compiling and Executing Your Program

Once you have created the **`mp4.c`** file and written the **`print_stamps`** function, you can compile your code by typing:

```
gcc -g -Wall main.c mp4.c -o stamps
```

The "**-g**" argument tells the compiler to include debugging information so that you can use **gdb** to find your bugs (you will have some).

The "**-Wall**" argument tells the compiler to give you warning messages for any code that it thinks likely to be a bug. Track down and fix all such issues, as they are usually bugs. Also note that if your code generates any warnings, you will lose points.

The "**-o stamps**" argument tells the compiler to name the resulting program "**stamps**". If compilation succeeds, you can then execute the program as specified in the examples given earlier in this document.

The **stamps** program takes five command line arguments:

```
./stamps <amount> <s1> <s2> <s3> <s4>
```

The first argument is the amount of postage needed, and the other four arguments are the region sizes. If the arguments given are not valid (according to the assumptions given in the "Specifics" section), the code in **main.c** responds without calling your function.

Grading Rubric

We put a fair amount of emphasis on style and clarity in this class, as reflected in the rubric below.

Functionality (70%)

- 5% - Function returns 1 when amount can be covered exactly, and 0 when it cannot.
- 10% - Output matches specified output exactly.
- 20% - Function chooses correct combination when no ties in value are possible.
- 20% - Function chooses correct combination when no ties in both value and number of coins are possible.
- 15% - Function always chooses correct combination.

Style (15%)

- 15% - Loops consider only reasonable combinations of stamps (note, for example, that you need at most $\lceil \mathbf{amount} / \mathbf{s1} \rceil$ of the first stamp, where the brackets indicate rounding up to the next integer).

Comments, Clarity, and Write-up (15%)

- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 10% - code is clear and well-commented, and compilation generates no warnings (note: any warning generated during compilation means 0 points here)

Note that some categories in the rubric may depend on other categories and/or criteria. For example, if you code does not compile, you will receive no functionality points. Your function must be able to be called many times and produce the correct results, so we suggest that you avoid using any static storage (or you may lose most/all of your functionality points).