CS3300: Compiler Design **Odd Sem**

Lab #11

Register Allocation Deadline: October 21, 2018, 23:55

Important Instructions

- Program should read from stdin and write to stdout.
- Follow submission guidelines carefully. Violating the guidelines will incur penalty.

Task

Write a C/C++ program (no need of lex/yacc) for a variant of register allocation. Assume that the registers are numbered $R_1, R_2, ...$ The purpose is to assign the minimum number of registers to variables such that no two simultaneously *live* variables are assigned the same register. However, since a variable may be live across multiple live-ranges, we may have to *spill* a register to memory, to free the register for some other variable currently live. The same register may be assigned to multiple variables (across non-overlapping live-ranges). Similarly, the same variable may be assigned to different registers at different live-ranges. Of course, the same variable would not be simultaneously assigned to more than one registers. If two variables get live at the same time, they should be assigned registers in the order they appear in the input file (sorted increasing). For spilling, use the smallest numbered register whose live-range does not conflict with the current variable for which the register is required. If all the live-ranges in registers overlap with that of the required variable, choose the smallest numbered register that is not live at *this line* (try proving that such a register always exists).
Note that the above method of spilling is not optimal, but is done to ease your life. In reality, we will have to balance the spill cost (memory access) and the benefit of holding a variable in register due to a future use.

Input

Tabular information on live-ranges for variables: `var-name live-start live-end`, one per line, sorted-increasing on `var-name`. Each line denotes that variable `var-name` is *live* from `live-start` line number in some program's IR through `live-end` line number (including `live-end`). The line numbers are positive (>0 and <100) integers with `live-start` $\leq$ `live-end`. `var-name` would only be a..z in small-case (strlen(`var-name`) == 1). You do not need to make any checks on the input format.

Output

Print two numbers: `num-reg num-spill`
`num-reg` is the minimum number of registers required for the input program, computed as max-clique in the graph of live-ranges. `num-spill` is the exact number of memory spills required to free registers.

Examples

Example 1

Input

```
a 1 2
a 5 6
b 7 8
b 3 4
```

Output

```
1 3
```

Explanation: The live-range graph contains two nodes (for `a` and `b`), but no edges, because no two live-ranges overlap. The max-clique size is thus 1. Hence we require a single register, say R_1. Thus, `num-reg` = 1.
Variable `a` can be loaded in R_1 at line 1. At line 3 of the original source program (line 4 of the input), we need variable `b`. Since we have a single register, `a` in R_1 needs to be spilled to memory; hence `num-spill` = 1.
At line 5, we again require variable `a`, hence we spill `b` to memory; hence `num-spill` = 2.
At line 7, we require `b` again, hence we spill `a` to memory; hence `num-spill` = 3. We print this.

Example 2

Input Program
Input

```
a 1 1
a 2 5
b 2 2
c 5 5
c 7 8
```

Output

```
2 1
```

Explanation: The live-range graph contains three nodes (for `a`, `b`, `c`), and two edges: `a--b` and `a--c`. This is because live-range of `a` 2..5 overlaps with that of `b` 2..2 as well as with that of `c` 5..5.
Since max-clique size is 2, `num-reg` = 2 (say, R_1 and R_2).
`a` gets R_1.
`b` gets R_2 at line 2.
At line 5, we need to load `c` in register. We have two choices: spill R_1 or spill R_2. However, R_1 contains `a` whose live-range overlaps with that of `c`. R_2 contains `b` whose live-range does not overlap with that of `c` Hence, we choose to spill R_2, and `c` gets R_2 for the rest of the program. Hence, `num-spill` = 1.

Example 3

Input Program
Input

```
a 1 1
a 5 5
a 8 8
b 1 1
b 9 9
c 5 6
c 7 7
d 8 8
e 5 5
e 6 6
e 9 9
```

Output

```
3 4
```

Explanation: The live-range graph contains edges `a--b, a--c, a--d, a--e, b--e, c--e` with max-clique of size 3. Thus, `num-reg` = 3 (say, R_1, R_2 and R_3).

At line 1, `a` gets R_1 and `b` gets R_2.
At line 5, `c` gets R_3.
At the same line, `e` is also live. However, in the live-range graph, `e` is connected to all of `a, b, c`. In such a case, we need to find the variable that is not live *at this line* (line 5). Such a variable is `b`, hence R_2 gets spilled to memory, and `e` is loaded in R_2. `num-spill = 1`.
At line 6, `c` and `e` are live, and both are already in registers.
At line 7, `c` is live and present in register R_3, hence no spilling happens.
At line 8, `a` and `d` are live. `a` is already present in R_1. Thus, `d` needs to spill R_2 or R_3 corresponding to `e` or `c`. None of the two are co-live with `d`, hence we choose the smallest of the two registers, and `d` spills `e` to memory and occupies R_2. `num-spill` = 2.
At line 9, `b` and `e` get live.
`b` is connected to `a` and `c` in the live-range graph but not to `d`. Hence it spills R_2 to memory. `num-spill` = 3.
`e` is connected in the graph to all of `a, b, c` currently loaded in registers. Thus, we look for the variable that is not live at line 9. There are two of them: `a` and `c`, loaded in R_1 and R_3. `e` chooses the smaller of the two, spilling `a` to memory. `num-spill` = 4.

Submission

Submit a tar.gz file with filename as <ROLLNO>.tar.gz (eg. CS12B043.tar.gz) containing the following structure:

- CS12B043/
  - source files
  - Makefile

The Makefile should generate an executable **a.out**.