Pset4: Blotto

CPSC 223 Spring 2024

Instructors: Alan Weide & Ozan Erat

Due: 11:59 PM NHT Friday Mar 8, 2024

Objectives

- To use a map ADT (Abstract Data Type)
- To implement a map ADT using a hash table

Introduction

Now that our pirate friends can sign up for Hookbook, they need something to do while they're on the app. In this assignment, you will implement a game for your crew to play during the long voyages searching for treasure around the Caribbean.

Blotto is a simultaneous game for two players in which the players distribute military units over a number of battlefields each worth some number of points that represent the battlefields' strategic value. (Note that the linked article discusses a limited version in which each battlefield is worth the same amount; we allow for a different value for each battlefield, and if you prefer a more peaceful context, Blotto can be seen a a model of allocating campaign resources over contested areas, or spreading advertising resources across various media outlets.) Each player has a set number of units to distribute, each unit is equivalent to each other unit, and a unit must be allocated entirely to a single battlefield. For each battlefield, the player who allocates more units to that battlefield wins the points for it (with the points split evenly in case of a tie, including a 0-0 tie), and the winner is the player with the most total points.

For example, suppose there are four battlefields worth 1, 2, 3, and 4 points respectively and each player has 10 units to distribute. If player 1's distribution is $(2, 2, 2, 4)$ and player 2's distribution is $(3, 4, 2, 1)$ then player 1 wins battlefield 4 player 2 wins battlefields 1 and 2, and the players tie on battlefield 3. Player 1's score is then $1.5 + 4 = 5.5$ and player 2's score is $1 + 2 + 1.5 = 4.5$. Player 1 wins.

Overview

Write a program called `Blotto` that reads players' distributions from standard input, along with a list of head-to-head matchups between pairs of those players. Output gives the result of each matchup, where, for each matchup, points are awarded to each player based on who placed more units in each battlefield, and the value of the battlefields given on the command line. For example, if the file `blotto.in` contains

```
P1,2,2,2,3
P2,3,3,2,1
P3,2,2,5,0

P1 P2
P1 P3
P2 P3
```

then running `./Blotto 1 2 3 4 < blotto.in` should result in output

```
P1 5.5 - P2 4.5
P1 5.5 - P3 4.5
P2 7.0 - P3 3.0
```

Be sure you understand exactly how the output is computed from the input based on the rules of the game.

Implementing the `gmap` ADT

You must implement the `gmap` ADT according to the specification in `gmap.h`, which can then be used to store the players' distributions as you read them from standard input for retrieval later when you read the matchups. **You may not modify `gmap.h` at all.** The `gmap` ADT defines a map from generic keys to generic values, with the functions used for hashing, copying, comparing, and freeing the keys passed to the map upon creation.

The `gmap` functions must run in the following time bounds, where n is the number of key/value pairs in the map:

- `gmap_create`: $O(1)$
- `gmap_size`: $O(1)$ (worst-case)
- `gmap_contains_key`: expected $O(1)$, worst-case $O(n)$
- `gmap_get`: expected $O(1)$, worst-case $O(n)$
- `gmap_put`: expected $O(1)$, worst-case $O(n)$
- `gmap_remove`: expected $O(1)$, worst-case $O(n)$
- `gmap_for_each`: $O(n)$ (assuming `f` runs in $O(1)$ time)
- `gmap_keys`: $O(n)$
- `gmap_destroy`: $O(n)$

These bounds assume that the key operations (copy, compare, hash, and free), and `malloc` and `free` run in $O(1)$ time. Expected running times are under the assumption that the distribution of keys and the hash function are such that the hash values of the keys are distributed uniformly across the range of the `size_t` type.

Your ADT will be tested with a series of **unit tests** that test individual functions. An incomplete suite of unit tests is implemented in `gmap_unit.c` (with supporting modules `gmap_test_functions` and `string_key`) which will be replaced with a complete test suite for the private test script. Your makefile must include a target `GmapUnit` that is built from those files and your implementation of the `gmap` module. All unit tests must pass with no errors reported from Valgrind. No public or private tests will violate the preconditions, and we will not test out-of-memory conditions for this part or the `Blotto` executable described below.

Note that although you may not use all of the functions provided by the `gmap` module when writing the `Blotto` program, you must implement all of them for this assignment.

Your `Blotto` program must run in expected $O(n + m)$ time for a fixed number of battlefields and total number of units, where n is the number of entries and m is the number of matchups, under the assumption that `malloc` and `free` run in $O(1)$ time.

Details

Command-line Arguments

- There will be at least one command-line argument
- Each command-line argument will be a positive integer in a format readable by `atoi`
- The number of battlefields is determined by the number of command-line arguments

Input

- Standard input will consist of two parts separated by a blank line
- The first part lists the players' distributions as follows
  - There will be one line per player
  - There may be no players, in which case standard input will begin with a blank line
  - Each line starts with a unique non-empty entry id for the player
  - The id will contain at most 31 characters and no commas or whitespace
  - The rest of the line will contain the number of units the player distributes to each battlefield, with each number being a non-negative integer in a format readable by `scanf` with the `%d` format

  specifier
  - Each player's distribution will have a value for each battlefield
  - All parts of the data for each player are separated by commas
  - There is no whitespace on each line except for a newline at the end of each line
  - The total number of units for each player will be non-zero and equal to the number of units for all other players
- The second part lists the matchups as follows
  - There is one matchup per line
  - There may be no matchups, in which case standard input ends with the blank line that follows the players' distributions
  - Each matchup is specified by giving the id of two different players that were given in the first part of the input, separated by a single space
  - There is no other whitespace on each line aside from the space that delimits the ids and the newline at the end of the line

Output

- There is exactly one line of output per matchup, written to standard output in the order the matchups appeared in standard input
- If there are duplicate matchups in standard input, then the corresponding output will appear repeated in the corresponding locations in the output
- Each line contains the id of the winner of the matchup (or the id of the first player given in the matchup, in the case of a draw), a single space, the score for that player given with one digit after the decimal point, a hyphen (`'-'`) with a single space on either side as a separator, and then the id and score of the other player in the same format
- Each line ends with a newline character
- There is no other output to either standard output or standard error

If any of the input to the `Blotto` program (standard input or the command-line arguments) is invalid for any reason, then the output can be anything but your program must not crash or hang. (Segmentation fault is a crash. Detecting invalid input and exiting with status code `1` is not a crash.)

Hints and Other Considerations

- You can use the `entry` module to read entries, but we do not guarantee that the `entry_read` function in that module will handle all invalid inputs without errors.
- Can you test your `Blotto` implementation before you complete your `gmap` implementation? (Hint: there is an incomplete, inefficient implementation of `gmap` using an unsorted array to hold the keys and values in `gmap_array.c`.)
- Can you test parts of `Blotto` before you've written code to read the matchups file?
- What functions might be helpful to divide your `Blotto` implementation into?
- Are there any functions from the unsorted array implementation of `gmap` that you can use unchanged in your hash table implementation?
- What functionality of `gmap` is not strictly necessary to test other functionality?
- What test cases are missing from the public tests?
- Why is it necessary that `gmap` *copies* keys rather than storing pointers to the ones passed to `gmap_put`? Why does it not need to do the same thing for the values?
- In addition to the unsorted array implementation of `gmap` mentioned above, the partial implementation of a chained hash table from the in-class examples and the implementation of the `Dict` ADT in Prof. Aspnes's notes may be helpful as starting points.
- It's time to take the training wheels off just a little bit…for this assignment, you're required to create your own `makefile` from scratch. Here are the requirements for your `makefile`:
  - It must be named "`makefile`" or "`Makefile`"
  - It needs at least four targets, in the following order:
    - `all`, which builds both `Blotto` and `GmapUnit`

    * `Blotto`, which compiles an executable named `Blotto` that runs the `Blotto` game
    * `GmapUnit`, which compiles an executable named `GmapUnit` that runs the main function from `gmap_unit.c`
    * `clean`, which removes all files created by building any of the other targets
  - Both executables must compile with the following flags: `-std=c17 -Wall -pedantic -g`

Testing Script

Included with your starter code is a directory named "`Tests`". In that directory is an executable file named `test.Blotto` (among other files). Running this script *from the directory containing your `makefile`* will perform the following actions: 1. Builds `Blotto` and `GmapUnit` using your `makefile` 2. Runs a variety of unit tests using your `GmapUnit` executable 3. Runs a variety of input/output tests using your `Blotto` executable 4. Runs valgrind with those same tests 5. Runs an automated performance analysis of the various functions you have implemented 6. Runs your program on a variety of "invalid" input under which your program must crash or hang 7. Reports how many test cases your program passed

Suppose your project structure looks like this on the Zoo:

```
~/cpsc-223/hw4-blotto/          *-- Current working directory
    blotto.c
    libblotto.h
    libblotto.c
    LOG.md
    makefile
    README.md
    Tests/
        bin/                    # Helper scripts
            big_oh.py
            Head
            makewarn
            run
        t001                    # Test case script
        t001.out                # Test case answer
        t002
        t002.out
        ...
        test.Blotto             # Main testing script
```

Then, from the terminal in the `hw4-blotto` directory, you could run this test script with the following command (and you would see the following output, and more):

```
[netID@viper:~/cpsc223/hw4-blotto]
$ Tests/test.Blotto

Public test script for Blotto (10/22/2023)

***** Checking for warning messages *****
Making -B Blotto GmapUnit
...
```

> **Note**: This test script has *only* been confirmed to work on the Zoo. There is no express or implied warranty for its correctness or stability on any other platform.
>
> **Note**: This is not an exhaustive test script, and there is a nontrivial cohort of test cases that we will run when grading that are not included here. Put another way, even if your program passes all 55 test cases in this script, it is not guaranteed to be a correct implementation. (But it's probably decent, at worst.)

Note: In the zip/upload/download process to distribute the starter package, executable permissions have been stripped from the testing scripts. Run the command `chmod -R u+x Tests/` to restore them. The `-R` flag is important because it indicates "recursive". (This command is a bit overkill in that the `*.in` and `*.out` files do not need executable permissions, but you can undo those afterward with `chmod u-x Tests/*.{in,out}`.)

Starter File Modification Restrictions

Most of the starter files for Pset 4 (Blotto) have comments explaining exactly what you may and may not change. There are two important ones to note:

- `gmap.h`: **DO NOT MODIFY** this file. Private test cases rely on this file existing as is and **if your submission is not compatible with the starter code version of `gmap.h`, you will receive minimal or no credit for this assignment**
- `string_key.h`, `string_key.c`: **DO NOT MODIFY** these files. Private test cases rely on these files existing as is and **if your submission is not compatible with the starter code version of `string_key`, you will receive minimal or no credit for this assignment**

Everything else may be modified at will. The private grading script will replace `gmap_unit.c` and `gmap_test_functions.{c,h}` with its *own version* of those files and compile its own `GmapUnit` executable. The private grading script does not (directly) use `entry.{c,h}` or `gmap_array.c`.

Submissions

Submit your project to Gradescope by uploading all files needed to compile and run both executables required by this assignment (`Blotto` and `GmapUnit`). Do not submit any files in the `Tests` directory. It is due **11:59 PM NHT on Friday, March 8, 2024**.

Refer to the Syllabus for information on collaboration and late-assignment policy.