[TB14131] (IMAC-U) - 情報処理演習

C4TL1205 - LANDY Lucas (COLABS Student) -
lucas.landy.t3@dc.tohoku.ac.jp

Table of Contents

Assignment: Practice of Information Processing

This assignment consists of two main programs: `J_score.c` and `bisection.c`. Each program fulfills specific computational requirements based on provided specifications. Below is a detailed overview of each program, including functionality, structure, algorithms used, and usage instructions.

1. Japanese Football League Ranking Program (`J_score.c`)

Overview:

`J_score.c` and `J_score2.c` is a program that processes and ranks teams in the Japanese football league based on statistical data provided in `J_dataset.csv`. `J_score.c` is using the `J_score_template1.c` and `J_score.c` is using the `J_score_template2.c`. The program reads team statistics, calculates scores and goal differences, sorts the teams according to specified criteria, and outputs the ranking in `J_score.txt`.

Specifications

1. **CSV File Format**:

- Each line in `J_dataset.csv` represents a team's performance in the following order:

```
Team Name, Wins, Draws, Losses, Goals For (GF), Goals Against (GA)
```

2. **Ranking Criteria**:
   - **Win Points**: Calculated as `3 * wins + 1 * draws`.
   - **Goal Difference (GD)**: Calculated as `GF - GA`.
   - **Tiebreakers**:
     - Higher win points rank higher.
     - If win points are the same, higher GD ranks higher.
     - If GD is also the same, higher goals scored (GF) ranks higher.
3. **Output Requirements**:
   - In addition to the fields in the CSV, the output should include **Ranking**, **Win Points**, and **Goal Difference**.
   - The output should be sorted by ranking.

Program Structure

1. **Dynamic Memory Allocation**:
   - The program dynamically counts the number of teams in the CSV and allocates memory accordingly, making it adaptable to any number of teams.
2. **Functions**:
   - `read_data`: Reads and parses CSV data into an array of team structures.
   - `calc_score`: Computes win points and goal difference for each team.
   - `rank_score`: Sorts the teams using an efficient heap sort algorithm.
   - `write_data`: Outputs the sorted data to `J_score.txt` in a readable format.
   - `swap_SC`: Function to swap two teams in the table array
   - `count_teams`: Counts rows in the CSV to determine the number of teams.
   - `heapify`: Heapifies a subtree rooted with node i in an array of SC structures, ensuring the max-heap property
3. **Heap Sort for Efficient Ranking**:
   - We use heap sort to rank teams based on the given criteria. Heap sort is efficient with $O(n \log n)$ time complexity, which is suitable for handling larger data sets.

How to Run

1. **Compile**:

```
gcc -o J_score J_score.c
```

or

```
gcc -o J_score2 J_score.c
```

2. **Execute**:

```
./J_score
```

or

```
./J_score2
```

3. **Output**: The rankings are saved in J_score.txt (or J_score2.txt) in a table format.

```
Ranking   Team            Wins  Draws  Losses  GF    GA    Points  GD
-------------------------------------------------------------------
1         Yokohama FC      7     8     18     30    56      29   -26
2         Kashiwa Reysol   6    14     13     32    46      32   -14
3         Gamba Osaka      9     7     17     38    60      34   -22
4         Shonan Bellmare  8    10     15     40    55      34   -15
5         Kyoto Sanga FC  11     4     18     37    44      37    -7
6         Sagan Tosu       9    11     13     43    46      38    -3
7         FC Tokyo        11     7     15     41    46      40    -5
8         Consadole Sapporo   10    10    13    56    59      40    -3
9         Albirex Niigata  10    12     11     35    40      42    -5
10        Kawasaki Frontale   13     8    12    50    45      47     5
11        Cerezo Osaka    15     4     14     39    33      49     6
12        Kashima Antlers  13    10     10     41    33      49     8
13        Avispa Fukuoka   15     6     12     37    42      51    -5
14        Nagoya Grampus   14     9     10     40    35      51     5
15        Urawa Red Diamonds    14    12     7    40    27      54    13
16        Sanfrecce Hiroshima    16     7    10    41    28      55    13
17        Yokohama F. Marinos    19     7     7    62    37      64    25
18        Vissel Kobe     20     8     5     59    29      68    30
```

Conclusion

The `J_score.c` program efficiently ranks teams in the Japanese football league by dynamically reading and processing team statistics from a CSV file. With the implementation of heap sort, the program handles larger datasets with high performance, sorting teams based on win points, goal difference, and goals scored. This flexible approach, including dynamic memory allocation, allows for future expansion with more teams and enhances usability, making `J_score.c` a robust solution for sports ranking calculations.

---

2. Root Finding Program (`bisection.c`)

Overview

`bisection.c` is a program that finds the roots (zeros) of a continuous function using either the bisection method or Newton's method. It calculates the roots of the function $ f(x) = \frac{x}{12} + \sin(x) $ by iteratively halving the interval or using tangent slopes until the desired precision is reached.

Specifications

1. **Bisection Method**:
    - The program divides an interval `[a, b]` into halves, evaluating the function at the midpoint `c = (a + b) / 2`.
    - If `f(a)` and `f(c)` have opposite signs, there is a root in `[a, c]`.
    - If `f(c)` and `f(b)` have opposite signs, there is a root in `[c, b]`.
    - This process repeats until the error is smaller than 10^{-8} or the maximum iteration limit is reached.
2. **Newton's Method (Optional)**:
    - Newton's method is available as an alternative to bisection for faster convergence.
    - Starting from an initial guess $ x_0 $, Newton's method iteratively improves the guess based on the formula $ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} $.
    - This method requires the derivative $ f'(x) = \frac{1}{12} + \cos(x) $.
3. **Function**:
    - The function to be solved is $ f(x) = \frac{x}{12} + \sin(x) $.
    - Both methods iterate until the error is below 10^{-8} or the maximum iterations are exceeded.

Program Structure

1. **Finding All Roots in a Specified Interval**:
    - The `find_all_roots` function iterates through subintervals within a given range (e.g., `[-10, 10]`) and finds roots within each subinterval.
    - This allows for finding multiple roots in a larger range, ensuring all solutions are identified.
2. **Recursive Bisection Method**:
    - The `bisection` function is implemented recursively, updating interval bounds until a root is found to the specified tolerance.
    - If no root is found within the tolerance and maximum iteration limit, the method will exit.
3. **Counting and Outputting the Number of Iterations**:
    - Both the bisection and Newton's method implementations include a counter for the number of iterations.

- The number of iterations for each root is printed alongside the root value to provide insight into the method's convergence efficiency.

4. **Functions**:
   - `f`: Function to evaluate $f(x) = x/12 + sin(x)$.
   - `f_prime`: Function to evaluate $f'(x)$ for Newton's method, derivative of $f(x)$.
   - `bisection`: Recursive bisection method to find a root in [a, b] to specified tolerance.
   - `newton_method`: Newton's method for finding a root near $x=0$.
   - `find_all_roots`: Function to find all roots in the interval [start, end] using the bisection method.

5. **Optional Enhancements**:
   - The program allows the user to choose between bisection and Newton's method for finding roots.
   - If Newton's method is chosen, it may fail to converge if the derivative is close to zero, in which case the program will alert the user.

How to Run

1. **Compile**:

```
gcc -o bisection bisection.c -lm
# Note: The -lm flag links the math library, which is required for functions
like sin and cos.
```

2. **Execute**:

```
./bisection
```

3. **Execution Instructions**:
   - The program will prompt the user to choose a method:
     - Enter 1 for the bisection method.
     - Enter 2 for Newton's method.
   - After selecting a method, the program finds all roots within the interval [-10, 10] and prints each root, the interval in which it was found, and the number of iterations required to reach the solution.

Example Output:

```
# Note: Using busection method:
Choose method: 1 for Bisection, 2 for Newtons Method: 1
Finding roots in the interval [-10.0, 10.0] with step 1.0:
Root 1: -5.7805750817 found in [-6.000000, -5.000000] with 26 iterations
```

```
Root 2: -3.4316084683 found in [-4.000000, -3.000000] with 25 iterations
Root 3: -0.0000000075 found in [-1.000000, 0.000000] with 27 iterations
Root 4: 3.4316084683 found in [3.000000, 4.000000] with 25 iterations
Root 5: 5.7805750817 found in [5.000000, 6.000000] with 26 iterations

# Note: Using Netwton's method:
Choose method: 1 for Bisection, 2 for Newtons Method: 2
Finding roots in the interval [-10.0, 10.0] with step 1.0:
Root 1: -5.7805750827 found in [-6.000000, -5.000000] with 4 iterations
Root 2: -3.4316084602 found in [-4.000000, -3.000000] with 3 iterations
Root 3: -0.0000000000 found in [-1.000000, 0.000000] with 3 iterations
Root 4: 0.0000000000 found in [0.000000, 1.000000] with 3 iterations
Root 5: 3.4316084602 found in [3.000000, 4.000000] with 3 iterations
Root 6: 5.7805750827 found in [5.000000, 6.000000] with 4 iterations
```

This example output shows multiple roots found within the interval and the iterations neededfor each root.

Conclusion

The bisection.c program offers flexible, robust root-finding capabilities using both the bisection and Newton's methods. Its ability to find multiple roots in a specified range, along with the choice of algorithms, makes it adaptable for a wide range of functions and intervals.