Algorithmic Problem Solving

Practice Problem Solving Session/Seminar

Herman Appelgren
Dept of Computer and Information Science
Linköping University

Solving a Problem

1. Analyze the problem
   - What data do you receive? What data should you produce?
   - How large is the input? What is a feasible time complexity?
2. Find a solution
   - Can you solve it by hand/in your head? What steps do you perform?
   - Could you alter the problem somehow to make it easier?
   - When you have a solution in mind, are there any corner cases?
   - How can you store the data efficiently? Is your time complexity feasible?
3. Implementation
   - Start with pseudo code (code comments) if your solution is complex.
   - Use good variable names! Invest a few seconds writing descriptive names, and in return you'll save a lot of debugging time.
   - Are there large numbers? Are 32-bit integers sufficient?
   - Is there a lot of input/output? See Kattis' help section for advice.
   - Learn tools for debugging (valgrind, gdb, jdb, pdb, ...)

Problem A - Babelfish

- We receive a dictionary of n (0 ≤ n ≤ 100,000) word pairs $< a_i, b_i >$ followed by m (0 ≤ m ≤ 100,000) words c_i.
- **Problem:** For each c_i, if $c_i = b_i$ for some i, print a_i. Otherwise, print "eh".
- **Solution:** Straight-forward lookup of c_i, but make sure to use an appropriate data structure.
  - Linear search takes O(n) per lookup => too slow.
  - Use a map (TreeMap/std::map in Java/C++) with lookup in O(log(n)).
  - Use a hashmap (HashMap/std::unordered_map in Java/C++ or Python dictionary) with lookup in O(1).
- **Time complexity:** Assuming $n \geq m$, the time is bounded by parsing the input. $O(n\ log(n))$ with map or $O(n)$ with hashmap.

C++ example (0.09 sec)

babelfish.cpp

```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <unordered_map>

using namespace std;

int main() {
    // Improves IO speed without having to use the cstdio routines,
    // which are particularly cumbersome for string data. Note call
    // order, this is important!
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    unordered_map<string, string> dict;
    while (true) {
        string line, a, b;
        getline(cin, line);
        if (line.size() == 0) break;
        istringstream iss(line);
        iss >> a >> b;
        dict[b] = a;
    }

    string c;
    while (cin >> c) {
        if (dict.find(c) == dict.end()) {
            cout << "eh" << "\n";
        } else {
            cout << dict[c] << "\n";
        }
    }
    // Avoid std::endl. It flushes the buffer, which reduces performance
    // when the output is large. Prefer "\n" for line breaks and do a
    // single std::flush at the end.
    cout << flush;

    return 0;
}
```

Java example (0.57 sec)

Babelfish.java

```java
import java.util.HashMap;
import java.io.*;

public class Babelfish {
    public static void main(String[] args) throws IOException{
        // Buffered IO to improve speed. For numeric input, use
        // the Kattio class available on the Kattis help pages.
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter output = new PrintWriter(new BufferedOutputStream(System.out));

        HashMap<String, String> dict = new HashMap<>();
        while (true) {
            String[] line = input.readLine().split(" ");
            if (line.length < 2) break;
            dict.put(line[1], line[0]);
        }
        while (true) {
            String c = input.readLine();
            if (c == null) break;
            if (dict.containsKey(c)) {
                output.println(dict.get(c));
            } else {
                output.println("eh");
            }
        }
        output.flush(); // Crucial when using buffered output!
    }
}
```

Python example (0.62 sec)

babelfish.py

```python
dict = {}
line = input()
while line:
    a, b = line.split()
    dict[b] = a
    line = input()
while True:
    try:
        c = input()
        print('eh' if c not in dict else dict[c])
    except:
        break

```

Notes on Input/Output

- Fast I/O: See Kattis help section.
- The IO specification describes the content of ONE test file, i.e. what your program must process in a single run. Your program might be executed several times with different input files.
- Make sure to read the IO specification carefully! In particular:
  - How are sections of the input data separated?
  - How do you know how many datapoints there are?
  - How do you know that there is no more input? Common variants: An integer n at the beginning, blank lines, special input tokens (such as "0 0" before EOF in CD), or simply EOF.
  - Are there multiple test cases in the same input?
- What are the limits of the problem? Is it e.g. allowed to have 0 datapoints?
- Note that the sample input/output doesn't always cover all possible input/output formats!

CD

- **Problem:** Given two sets A and B of integers ($|A| = n, |B| = m \leq$ 1,000,000), find the size of the intersection $A \cap B$.
- **Solution:** Again, the problem can be solved using lookup if we choose an appropriate data structure. There are several options:
  - Use vectors/arrays, sort them (already given in increasing order) and lookup with binary search.
  - Use set/hashset similar to map/hashmap in the previous problem.
- **Time complexity:** Assuming n in O(m).
  - Using vectors/arrays: input in O(n), sort in O(nlogn), lookup in O(nlogn) => O(nlogn)
  - Using set/hashset: both input and lookup in O(nlogn)/O(n).
- **Warning:** The input size is quite large (up to a few MB), so make sure to use techniques for fast IO.

Closest Sums

- Given is a set A = $\{a_1, \dots, a_n\}$ of n (2 ≤ n ≤ 1,000) integers and m (1 ≤ m ≤ 24) query integers b_i.
- **Problem:** For each b_i, output c_i such that $c_i = a_j + a_k$ where $j \neq k$ and $|b_i - c_i|$ is minimized.
- **Subproblem:** Find all possible sums of pairs of integers in A.
  - Iterate over A with two nested loop. O(n^2) is fine since n is small.
- **Solution:** Calculate all possible sums of pairs, store them in an efficient data structure and use lookup for all queries.
  - Here, a hash map doesn't work, since there might not be an exact match.
  - Use a sorted vector/array or a map, which allows lookup of closest match in O(log(n)).
- **Time complexity:** $O(n^2 \log(n^2)) = O(n^2 \log(n))$ to calculate all sums to an appropriate data structure. $O(m \log(n^2)) = O(m \log(n))$ for lookups.

Mathemagicians

- *n* magicians ($3 \leq n \leq 100{,}000$) magicians stand in a circle, each wearing either a red or a blue hat.
- The magicians can (one at a time) change the color of their hats to match one of their neighbors'.
- **Problem:** Is it possible to reach a given target configuration from a given original configuration?
- **Note:** The state space is huge (2^100,000 ≈ 10^30,000), so conventional search techniques (to be covered later in the course) are not feasible.

Mathemagicians

- Think in terms of contiguous "fields" of magicians wearing the same color hat.
- Operations on the borders extend/contract the size of adjacent fields.
- **Key conclusion**: We can only move or remove fields, not create new ones!
- **Solution:** Count the number of fields a in the original and b in the target configuration. Bar special cases, the target is reachable if $b \leq a$.
  - Special case 1: $a = 1$ and all magicians wear the wrong color.
  - Special case 2: The magicians wear alternating colors and the target is the opposite configuration of alternating colors.
- **Time complexity:** Count the number of fields in $O(n)$.

Problem set

- On the live sessions later during the semester, there will be six problem instead of today's four.
- Today's problems mainly involved basic data structures, but many problems on live sessions are based on more advanced algorithms and data structures implemented in the labs.
- According to the ranking at open.kattis.com, today's problems ranged from 2.1 to 4.8 in difficulty on a scale from 1-10. The difficulty of the live problems are usually in the range 2.5-7.5.
- Today the problems were arranged approx. in increasing order of difficulty, but this will not be the case in general.

Conclusion

- Most of today's problems were about basic data structures. If you found them difficult, consider recapitulating theory from your data structures and algorithms course.
  - OpenDSA is a good free online resource: https://www.ida.liu.se/opendsa/Books/TDDD86F20/html/
  - Make sure you are familiar with how they are implemented in your chosen programming language.
- Be familiar with how Kattis handles input and output.
  - Consult the documentation for your language.
  - Know how to ensure fast IO for large datasets.