Algorithm Homework 1

Hsuan-Yu Kuo

November 4, 2021

1 Environment

All codes in the zip files can run under Ubuntu 20.04 and g++ version 9.3.0. To run the program, run `g++ coin.cpp -o coin` or `g++ coin2.cpp -o coin2` in the command line and `coin`, `coin2` are the ELF executables.

2 Result

2.1 Formal statement

You are given an array of n elements $a_0, a_1, \cdots, a_{n-1}$; however, you don't know anything except the size of the array. All the elements except one of them are equal. You have a compare function *cmp* taking two sets of indices $\{s_{1,1}, s_{1,2}, \cdots, s_{1,|s_1|}\}, \{s_{2,1}, s_{2,2}, \cdots, s_{2,|s_2|}\}$ as arguments, all $|s_1| + |s_2|$ indices have to be distinct ($|s_1|, |s_2|$ are the size of s_1 and s_2, respectively.) The compare function will return a value depending on the relations between $\sum\limits_{i=1}^{|s_1|} a_{s_{1,i}}$ and $\sum\limits_{i=1}^{|s_2|} a_{s_{2,i}}$, with time complexity $\Theta(|s_1| + |s_2|)$ by adding all elements. Find the element that is different from others.

2.2 Algorithm 1 (coin.cpp)

2.2.1 Solution

From now on, we call an element **unique** if it is different from others.

Notice that if a_i is unique, then it is different from its adjacent elements; that is, $a_i \neq a_{(i-1) \bmod n}$ and $a_i \neq a_{(i+1) \bmod n}$ must hold. If $n \geq 3$, The inverse statement is also true: If $a_i \neq a_{(i-1) \bmod n}$ and $a_i \neq a_{(i+1) \bmod n}$ holds, then a_i must be unique. This observation lead to the following simple algorithm.

- Iterate i from 0 to $n-1$.
- For every i, use the compare function to compare $\{i\}, \{(i-1) \bmod n\}$ and $\{i\}, \{(i+1) \bmod n\}$. If both comparison shows that they are not equal, then i is the index of the unique element.

2.2.2 Runtime analysis

- Iterating i from 0 to $n-1$ is $O(n)$ iterations.
- For each iteration, comparing a pair of sets takes at most 3 operations (adding elements takes 2 operations, comparing takes 1 operation), so comparing two pairs takes at most 6 operations. Check the results of two comparisons takes 2 operations. Therefore each iteration takes $O(1)$ time.
- Total time complexity is $O(n)$.

2.3 Algorithm 2 (coin2.cpp)

2.3.1 Solution

Here we use a method similar to divide and conquer algorithm. Every time we update the candidates of the unique element. According to the numbers of candidates left, there are following cases:

- If there are exactly 1 candidates left, then we found the unique element.
- If there are exactly 2 candidates left, select any element x other than those two candidates. It is always possible since initially $n \geq 3$. Notice that x is not the unique element.

  Then compare each candidate to x; if any of them is not equal to x, then we found the unique element.
- In other cases, divide the candidates into three groups with equal size. There may be 1 or 2 elements left. Let sum_1, sum_2, sum_3 be the sum of the candidates in the first, second, third group, respectively.

  Notice that since the sizes are all the same, for $i \neq j$, $sum_i = sum_j$ means that the unique element is not in group i and j; otherwise the unique element must be in group i or group j.

  Comparing sum_1, sum_2 and sum_1, sum_3, we have following cases:

  - If $sum_1 \neq sum_2$ and $sum_1 \neq sum_3$, then the unique element is in the first group.
  - If $sum_1 \neq sum_2$ but $sum_1 = sum_3$, then the unique element is in the second group.
  - If $sum_1 = sum_2$ but $sum_1 \neq sum_3$, then the unique element is in the third group.
  - If $sum_1 = sum_2$ and $sum_1 = sum_3$, it means that the unique elements is not in those three groups. Update the candidates to be those remaining elements.

2.3.2 Runtime analysis

For $n \leq 2$, the running time is $\Theta(1)$.

For $n > 2$, let the running time is $T(n)$. The measuring and comparing uses $\frac{4}{3}\lfloor\frac{n}{3}\rfloor+2$ operations (Adding numbers uses $\frac{4}{3}\lfloor\frac{n}{3}$ operations, comparing takes two operations). Checking the query results used 2 operations. If it falls into the first three cases it uses additional $T(\lfloor\frac{n}{3}\rfloor)$ operations, otherwise it uses additional at most 6 operations. Therefore the running time is $\frac{4}{3}\lfloor\frac{n}{3}\rfloor + 2 + \max(T(\lfloor\frac{n}{3}\rfloor), 6) \leq T(\frac{n}{3}) + 6 + 2 + \frac{4}{3} \cdot \frac{n}{3}$

The time complexity can be simplified as follow:

$$T(n) = \begin{cases} T(\frac{n}{3}) + \frac{4n}{9} + 8 & \text{if } n \geq 2 \\ 1 & \text{otherwise} \end{cases}$$

Notice that the formula matches the third case of Master theorem, so the time complexity is $O(n)$.

2.4 Notes

In this report, since the time complexity of calculating the sum of weights is $\Theta(n)$, there are various ways to achieve $O(n)$ time complexity. I'm convinced that there is no way to achieve lower time complexity for deterministic algorithms.

If each weighing takes $O(1)$ time, the first solution is still $O(n)$, while the second solution is $O(\log n)$ according to the Master theorem. The second solution is much better in this case.