Data Structure and Algorithm, Spring 2021

Homework 3

Purple Correction Date: 05/27/2021 12:00

Orange Correction Date: 05/25/2021 18:00

Red Correction Date: 05/24/2021 20:00

Due: 23:59:00, Saturday, June 19, 2021

TA E-mail: dsa_ta@csie.ntu.edu.tw

Rules and Instructions

- Any form of cheating, lying, or plagiarism will not be tolerated. Students can get zero scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.
- In Homework 3, the problem set contains 6 problems and is divided into two parts, the non-programming part (Problems 1, 2, 3) and the programming part (Problems 4, 5, 6). The total score of these six problems are 500 points. **Note that due to the current pandemic situation, the score of HW3 is capped at 400 points.** That is, your-final-score $= \min\{400, \text{your-raw-score}\}$. The implication is that you may choose to work on only 80% of HW3 without any significant impact on your score. Please take advantage of this special policy to lower your own work load in the next few weeks.
- For problems in the non-programming part, you should combine your solutions in ONE PDF file. Your file should generally be legible with a white/light background—using white/light texts on a dark/black background is prohibited. Your solution must be as simple as possible. At the TAs' discretion, solutions which are too complicated will be regarded as incorrect. Moreover, if you would like to use any theorem which is not mentioned in the classes, please include its proof in your solution.
- The PDF file for the non-programming part should be submitted to Gradescope as instructed, and you should use Gradescope to tag the pages that correspond to each subproblem to facilitate the TAs' grading. Failure to tagging the correct pages of the subproblem can cause losing part or all of the scores on the subproblem.
- For the programming part, you should have visited the *DSA Judge* (https://dsa-2021.csie.org) and familiarized yourself with how to submit your code via the judge system in Homework 0.1126.
- For problems in the programming part, you should write your code in C programming

language, and then submit the code via the judge system. You can submit up to 5 times per day for each problem. The judge system will compile your code with

```
gcc main.c -static -O2 -std=c11
```

- Discussions on course materials and homework solutions are encouraged. But you should write the final solutions alone and understand them fully. Books, notes, and Internet resources can be consulted, but not copied from. For *each non-programming problem*, you have to specify the references (the Internet URL you consulted with or the people you discussed with) on the first page of your solution for that problem; for *each programming problem*, you have to specify the references on the first lines (comments) of your code (`main.c`) for that problem.

- Since everyone needs to write the final solutions alone, there is absolutely no need to lend your homework solutions and/or source codes to your classmates at any time. In order to maximize the level of fairness in this class, lending and borrowing homework solutions are both regarded as dishonest behaviors and will be punished according to the honesty policy.

- The score of the part that is submitted after the deadline will get some penalties according to the following rule:

$$\text{LateScore} = \max\left(0, \frac{86400 \cdot 5 - \text{DelayTime (sec.)}}{86400 \cdot 5}\right) \times \text{OriginalScore}$$

- If you have questions about HW3, please go to the course forum and discuss (*strongly preferred*, which will provide everyone a better learning experience). If you really need an email answer, please follow the rules outlined below to get a fast response:

  - The subject should contain two tags, `"[HW3]"` and `"[Px]"`, specifying the problem where you have questions. For example, `"[HW3][P1] Can swap in subproblem 7 be done in constant time?"`. Adding these tags allows the TAs to track the status of each email and to provide faster responses to you.
  - If you want to provide your code segments to us as part of your question, please upload it to Gist or similar platforms and provide the link. Screenshots or code segments directly included in the email is discouraged and may not be reviewed.

Non-Programming Part

Problem 1 - Hash (60 pts)

1. (10 pts) Suppose Arvin stores n keys in a hash table of size n^2 using a **uniform** hash function $h(k)$. What is the probability that there is any collision ?

2. (10 pts) Assume that we have a **uniform** hash function $h(k)$ which maps input k into hash space P. Arvin wants to generate a set of magic passwords using the hash function $h(k)$. What is the expected number of times to query $h(k)$, in order to get $\frac{|P|}{4}$ unique password ?

3. (20 pts) We have learned open addressing from the lecture video. Define two hash functions $h_1(k) = k \mod m$ and $h_2(k) = 1 + (k \mod (m-1))$. Please insert the keys $\{18, 34, 9, 37, 40, 32, 89\}$ in the given order into a hash table of length $m = 11$. Assume the hash table is empty initially. Please specify in a step-by-step manner how the keys are inserted into the hash table using open addressing with (1) linear probing with hash function $h_1(k)$, (2) double hashing with primary hash function $h_1(k)$ and secondary hash function $h_2(k)$.

   Please fill in the table below, showing the content of the hash table after each insertion.

   - Open addressing with linear probing

keys to be inserted \ index	0	1	2	3	4	5	6	7	8	9	10
18								18			
34											
9											
37											
40											
32											
89											

   - Open addressing with double hashing

keys to be inserted \ index	0	1	2	3	4	5	6	7	8	9	10
18								18			
34											
9											
37											
40											
32											
89											

4. (20 pts) Cuckoo hashing is a hashing technique which guarantees O(1) worst-case query time. It uses two tables T_1 and T_2 of the same size, and two hash functions $h_1(k)$ and $h_2(k)$. To place an element x into the hash table, start by inserting x into T_1 at position $h_1(x)$. If a collision happens, the element x originally stored at position $h_1(x)$ in T_1 is moved to T_2 at position $h_2(y)$. In case of another collision, the element z stored at position $h_2(y)$ in T_2 is again moved to position $h_1(z)$ in T_1. Repeat these steps until the moved element can be placed in a position without collision.
   However, in practice, it is possible for this insertion process fails by entering an infinite loop. If this happens, all elements would be removed from the two tables and rehashed with two new hash function $h_1'(k)$ and $h_2'(k)$. However, in this question, the inserted sequence does not introduce this condition, and thus you **DO NOT** need to consider this,

   Given two tables of size 7 each and two hash functions $h_1(k) = k \bmod 7$ and $h_2(k) = \lfloor \frac{k}{7} \rfloor \bmod 7$. Insert elements $[6, 31, 2, 41, 30, 45, 44]$ into the hash table using cuckoo hashing in the given order. Draw the content of the two hash tables after each insertion in a step-by-step manner by filling in the table below.

   - Table T_1, using $h_1(k)$

keys to be inserted \ index	0	1	2	3	4	5	6
6							
31							
2							
41							
30							
45							
44							

   - Table T_2, using $h_2(k)$

keys to be inserted \ index	0	1	2	3	4	5	6
6							
31							
2							
41							
30							
45							
44							

Problem 2 - String matching (60 pts)

In the following subproblems, if the question asks for an algorithm, your answer should include:

(a) Your algorithm in the form of pseudo-code and a brief explanation of its correctness.

(b) Analysis of time complexity and space complexity of your algorithm. Better complexity would result in the higher points, but correct algorithm would give you base points.

Consider a string $S[1..N]$ of length N. You are given Q queries ($Q \approx N$), where there are three numbers l_1, l_2, n. The query is to determine if string $S[l_1..l_1 + n - 1]$ equals to string $S[l_2..l_2 + n - 1]$ (i.e., the respective substrings starting at l_1 and l_2 and of length n match each other).

1. (10 pts) Design an algorithm to respond to these queries. Note that when analyzing the complexity, you should take the time of pre-processing and responding to queries (if any) into consideration.

Now you are given another string $S[1..N]$ of length N. Please compute a function $x(i) = \max\{p \mid S[1..p] == S[i..i + p - 1]\}$ for $1 \leq i \leq N$, and store these values $x(i)$ in an array X.

2. (10 pts) Given a string $S =$ "$bcdabcde$", compute $x(i)$ for $1 \leq i \leq 8$. 先從1 開始

3. (20 pts) Design an algorithm that calculate the content of array X, given the input string S. 要 O(n)

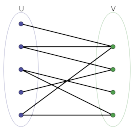
4. (20 pts) Now that you have finished the problems above, try to solve of a slightly different version of the string matching problem taught in class: find the number of occurrences of pattern p in string t. Utilize the array X created from the previous subproblems to construct an efficient algorithm to solve the problem. 和直接用rabin 有什麼不同?

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T.length
 2  m = P.length
 3  h = d^(m-1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                    // preprocessing
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n - m                // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n - m
14          t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

Problem 3 - Having Fun with Disjoint Sets (80 pts)

In subproblem 1 and 2, you can use a disjoint set data structure with linked list representation and **union by size** technique. You can use the following functions without any detailed explanation:

- `MAKE-SET(x)`: Create a set with one element `x`.
- `UNION(x, y)`: Merge the set that contains `x` and the set that contains `y` into a new set, and delete the original two sets that contains `x` and `y`, respectively.
- `FIND-SET(x)`: Find out the ID of the set that contains `x`.



1. (15 pts) Giver loves graph theory. Now, he's interested in **bipartite graph**. A bipartite graph is a simple, undirected, connected graph $G = (V, E)$ whose set of vertices V can be split into **two disjoint and independent** vertices set A, B, such that every edge in E connects a vertex in A and a vertex in B. Now, Giver wants you to implement three functions:

做法不一樣?
https://stackoverflow.com/questions/53246453/detect-if-a-graph-is-bipartite-using-union-find-aka-disjoint-sets

   - `INIT(N)`: Initialize the graph with N vertices. Vertices are numbered from 0 to $N-1$.
   - `ADD-EDGE(x, y)`: Add an edge which connects vertex x and vertex y. You can assume that $0 \leq x, y \leq N-1, x \neq y$.
   - `IS-BIPARTITE()`: Return `TRUE` is the graph is a bipartite graph, return `FALSE` otherwise

   Giver will perform `INIT(N)` first, then perform the other two operations M times. Please use disjoint set data structure to solve this problem (implement the above 3 functions) in $O(N + M \log N)$ time. Note that Giver can call the other two operations **in any order**. The following is an example that may help you solve this problem:

   - `INIT(5)`: Initialize the graph with 5 vertices.
   - `IS-BIPARTITE()`: You should return `TRUE`. This is a special case where there is no edge in the graph, but still considered bipartite.
   - `ADD-EDGE(2, 3)`: Add an edge which connects vertices $2, 3$.
   - `ADD-EDGE(3, 4)`: Add an edge which connects vertices $3, 4$.

需要紀錄 edge 嗎 O(N)?

   - `IS-BIPARTITE()`: You should return `TRUE`, since the graph so far is bipartite. This is because V can be split into $\{2, 4\}$ and $\{0, 1, 3\}$, and then any of the edges connect a vertex in the first vertex set and a vertex in the other.

Why?
2-3-4
0, 5

Let's consider a graph $G(V, E)$. The graph G is a bipartite graph if:

- The vertex set V of G can be partitioned into two disjoint and independent sets V_1 and V_2
- All the edges from the edge set E **have one endpoint vertex from the set V_1 and another endpoint vertex from the set V_2**

- `ADD-EDGE(2, 4)`: Add an edge which connects vertices $2, 4$.
- `IS-BIPARTITE()`: You should return `FALSE`, since the graph is not bipartite anymore. This means that no split among the vertices can be found, such that any of the edges connect 3 a vertex in the first vertex set and a vertex in the other.

(Hint: Bipartite graph is 2-colorable. That is to say, you can color the vertices in `black` and `white`, such that any edge connects a black vertex and a white vertex.)

2. (15 pts) Now, Giver finds out that the disjoint set data structure has more applications! Giver observes that there are N people playing `Paper, Scissor, Stone`. Giver also finds out that, each of the N people will always have the same hand (either paper, scissor, or stone) and never changes. In addition, Giver's friend, Robert, reports some relations between those N people to Giver. But Giver finds out that Robert might give him some contradicting relations. Now, Giver wants you to implement the following four functions:

   - `INIT(N)`: Giver sees N people. They are numbered from 0 to $N-1$.
   - `WIN(a, b)`: Person a defeats person b. (Paper defeats stone, scissor defeats paper, and stone defeats scissor).
   - `TIE(a, b)`: Person a and person b have the same hand.
   - `IS-CONTRADICT()`: Return `TRUE` is there's a contradict relation; return `FALSE` otherwise.

   Giver will perform `INIT(N)` first, then call the other three functions M times. Please use the disjoint set data structure to solve this problem (implement the above 4 functions) in $O(N + M \log N)$ time. Note that Giver can call the other three functions **in any order**. The following is an example that may help you solve this problem:

   - `INIT(5)`: Giver sees five people.
   - `WIN(1, 2)`: Person 1 defeats person 2.
   - `WIN(2, 3)`: Person 2 defeats person 3.
   - `IS-CONTRADICT()`: You should return `FALSE`, as there exist possible assignments of hands to people satisfying previous relations. For example, Person 1 can have `Paper`, person 2 has `Stone`, and person 3 has `Scissor`.
   - `TIE(2, 4)`: Person 2 has the same hand as person 4.
   - `WIN(4, 1)`: Person 4 defeats person 1.
   - `IS-CONTRADICT()`: You should return `TRUE`, since it is impossible for person 4 to defeat person 1 if person 1 defeats person 2.

Now, Giver wants to have fun with disjoint sets data structure using simple undirected graph again. He wants to implement the following functions:

- `INIT(N)`: Initialize the graph with N vertices. Vertices are numbered from 0 to $N-1$. Currently, there are no edges in the graph yet.
- `ADD-EDGE(x, y)`: Add an edge that connects vertex x and vertex y. Assume there is no edge connecting vertex x and y.
- `SHOW-CC()`: return the number of the connected components in this graph.
- `UNDO()`: Undo the last `ADD-EDGE()` operation. This operation is equivalent to removing the last added edge. When this operation is executed, assume there is at least one edge in the graph.

After spending 11.26 hours, Giver finally finished the implementation of the above four functions. He uses a **`stack`** to store the changes of the variables. When performing `UNDO()` operation, he pops all the changed variables, and restore the variables into their original values. You can see his original implementation here: https://www.csie.ntu.edu.tw/~b07902132/djs.c. He implemented it with linked-list representation but without any optimization techniques.

If there are N vertices in the graph, one **`INIT`** operation and followed by M other operations are performed in total, the time complexity would be $O(N + MN)$, which is too slow.

3. (15 pts) Now, Giver learns that he can apply **`path compression`** technique on his disjoint set implementation. You can see his implementation here: https://www.csie.ntu.edu.tw/~b07902132/djs_path_compression.c.

   Prove or disprove that, the complexity of the above implementation is $O(N + M \log N)$.

4. (15 pts) Now, Giver learns that he can apply **`union by size`** technique on his disjoint set implementation (without **`path compression`**). You can see his implementation here: https://www.csie.ntu.edu.tw/~b07902132/djs_union_by_size.c.

   Prove or disprove that, the complexity of the above implementation is $O(N + M \log N)$.

   (Hint: You can use everything that is proved in lecture, slide, or in the textbook. To disprove the time complexity, you can simply show a counter-example)

Finally, Giver thinks that you must observe **the beauty of disjoint set**. Here's the last challenge that Giver wants to give you.

He would like you to design a disjoint set implementation with the following functions:

- `MAKE-SET(x)`: Create a set with one element **`x`**.

- `UNION(x, y)`: Merge the set that contains `x` and the set that contains `y` into a new set, and delete the original set(s) that contains `x, y`.
- `SAME-SET(x, y)`: Return `TRUE` if element x and element y belong to the same set. Return `FALSE` otherwise.
- `ISOLATE(k)`: Element k will be isolated from the set it belongs to. That is to say, remove k from that set and form a set by itself.

Giver will perform M operations in total. You can assume that $0 \leq x, y, k \leq M - 1$.
Here's an example that may help you solve this subproblem.

- `MAKE-SET(1)`
- `MAKE-SET(2)`
- `SAME-SET(1, 2)`: You should return `FALSE`.
- `UNION(1, 2)`
- `SAME-SET(1, 2)`: You should return `TRUE`.
- `MAKE-SET(3)`
- `UNION(1, 3)`
- `SAME-SET(2, 3)`: You should return `TRUE`
- `ISOLATE(2)`
- `SAME-SET(1, 3)`: You should return `TRUE`
- `SAME-SET(2, 3)`: You should return `FALSE`

5. (20 pts) Design a disjoint set implementation that can support the above functions. If we perform M operations in total, its complexity should be $O(M\alpha(M))$.

   Note that if your time complexity is not $O(M\alpha(M))$, you can still get some partial points.

Congratulations, you have finished all the disjoint set challenges set by Giver!