Chapter 13 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Let F represent the set of all students enrolled in the Fall 2017 semester of EECS 281, let W represent the set of all students enrolled in the Winter 2018 semester of EECS 281, and let S represent the set of all students enrolled in the Spring 2018 semester of EECS 281. Which of the following student would **NOT** be included in the set $(F \cap W) \cup S$?
   A) A student who took EECS 281 only once and passed during the Winter 2018 semester
   B) A student who took EECS 281 only once and passed during the Spring 2018 semester
   C) A student who took but did not pass EECS 281 during the Fall 2017 semester and retook it during the Winter 2018 semester
   D) A student who took but did not pass EECS 281 during the Winter 2018 semester and retook it during the Spring 2018 semester
   E) More than one of the above

2. You are given two **sorted** vectors of integers, both of size n, and you want to devise an algorithm that can identify all items in one vector but not the other. Assuming that you are not allowed to create any additional containers, what is the worst-case time complexity of the most efficient algorithm that is attainable for solving this problem?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

3. You are given two **unsorted** vectors of integers, both of size n, and you want to devise an algorithm that can identify all items are either the first or second vector, but not both. Assuming that you are not allowed to create any additional containers, what is the worst-case time complexity of the most efficient algorithm that is attainable for solving this problem?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

4. The STL's set algorithms of `std::set_union()`, `std::set_intersection()`, etc., rely on which of the following assumptions about their input to produce the desired behavior?
   I. The two input ranges passed in must already have their values sorted.
   II. The two input ranges passed in must be stored in containers that support random access.
   III. The two input ranges passed in must contain the same number of elements.

   A) I only
   B) III only
   C) I and II only
   D) I and III only
   E) I, II, and III

5. Which of the following could be a practical application of the union-find data structure?
   A) Implementing a image editing program that identifies and updates regions of an image with a similar color
   B) Implementing a networking program that identifies groups of servers that can communicate with each other
   C) Implementing an algorithm that can be used to identify people in a social network with similar interests
   D) Implementing an algorithm that counts the number of connected components in a graph
   E) More than one of the above

6. Which of the following statements is **TRUE** about an implementation of the union-find container that forces every element in the data structure to keep track of the *ultimate* representative of its disjoint set, if no path compression is used?
   A) The find operation may take up to $\Theta(n)$ time, given n elements in the union-find container
   B) The union operation may take up to $\Theta(n)$ time, given n elements in the union-find container
   C) The amortized time complexities of find and union are both $\Theta(n)$, given n elements in the union-find container
   D) The amortized time complexities of find and union are both $\Theta(\alpha(n))$, given n elements in the union-find container, where $\alpha(n)$ denotes the inverse Ackermann function
   E) More than one of the above

7. Consider a union-find container with n elements, each with its representative set to itself. Assuming path compression is used, and only one call to `union_set()` is invoked with two arbitrary elements in this container (with no other union or find operations completed before or after), what is the largest possible number of elements that could have their representatives modified after this function call?
   A) 0
   B) 1
   C) 2
   D) $n-1$
   E) n

8. Consider a union-find container where, in addition to remembering one of its representatives, each element also keeps track of the *size* of the disjoint set that it belongs to. When elements from two disjoint sets are unioned together, the representative of the smaller set is always updated to the representative of the larger set (with ties broken arbitrarily). Under this implementation, what is the worst-case time complexity of a single call to `find_set()` on this union-find container? Note that $\alpha(n)$ represents the inverse Ackermann function.
   A) $\Theta(1)$
   B) $\Theta(\alpha(n))$
   C) $\Theta(\log(n))$
   D) $\Theta(n)$
   E) $\Theta(n\alpha(n))$

9. The following represents the state of a union-find container:

Item	0	1	2	3	4	5	6	7	8	9
Representative	5	6	7	2	1	3	6	9	8	8

   Among these 10 elements, how many disjoint sets are there?
   A) 0
   B) 1
   C) 2
   D) 3
   E) 4

10. Consider the same union-find container as in question 9. Suppose you wanted to merge all of the disjoint sets among these elements into a single set. What is the *minimum* number of union operations necessary to accomplish this?
   A) 0
   B) 1
   C) 2
   D) 3
   E) 4

11. Consider the same union-find container as in question 9. Suppose you used the path compression approach when you implemented union-find. If you were to call the find function on item 0, which of the following statements would be **TRUE**?
   A) 0's representative would become 3
   B) 5's representative would become 7
   C) 4's representative would become 6
   D) 2's representative would become 8
   E) More than one of the above

12. Consider the same union-find container as in question 9. Suppose you used the path compression approach when you implemented union-find. Which of the following operations would **NOT** change any of the representatives given in the above table?
   A) Calling find on item 7
   B) Calling find on item 4
   C) Calling find on item 3
   D) Calling find on item 1
   E) None of the above

13. You are given a union-find data structure, which implements path compression. There are 281 elements in the container, and 203 of these elements have themselves as their ultimate representative. What is the largest possible size that a disjoint set within this container can have?
   A) 77
   B) 78
   C) 79
   D) 202
   E) 203

14. You are given a union-find data structure, which implements path compression (but no other optimizations). There are 370 elements in the container, and 280 of these elements have themselves as their ultimate representative. You perform one call to `find_set()` on this container with an arbitrary element (with no other operations completed). What is the largest possible number of elements that could potentially have its representative changed with this call to `find_set()`?
   A) 90
   B) 91
   C) 279
   D) 280
   E) 370

15. You are given a union-find data structure, which implements path compression. There are five elements in this container: A, B, C, D, and E. Item A has a representative of E, item B has a representative of A, item C has a representative of B, and item D has a representative of D. After calling `union_set()` on items C and D, you notice that the ultimate representative of item D was changed from D to E. Which of the following statement is **TRUE**?
    A) Before the call to `union_set()`, there were three disjoint sets in this union-find container
    B) Before the call to `union_set()`, items D and E were part of the same disjoint set
    C) Before the call to `union_set()`, items C and E were part of different disjoint sets
    D) Before the call to `union_set()`, the ultimate representative of item E was C
    E) Before the call to `union_set()`, the ultimate representative of item E was E

16. You are given an array of strings `equations` that represent relationships between variables, where `equations[i]` is a string of length 4 that takes on one of two different forms: `"x==y"` or `"x!=y"` (where `x` and `y` can be interchanged with any other character from a to z, which you may assume will always be lowercase). Implement a function that returns whether it is possible to satisfy all the given equations.

    For example, if you are given the equations `"a==b"`, `"b==c"` and `"a!=c"`, you would return `false`, since there is no way to satisfy all three of these equations. However, if you were just given the first two equations, you would return `true`.

```
bool are_equations_satisfiable(const std::vector<std::string>& equations);
```

17. You are given a list of n computers, labeled from 0 to $n-1$. You are also given a two-dimensional vector of integers `firewall`, where `firewall[i] = [x, y]` indicates that computers `x` and `y` cannot be connected at all, either directly or indirectly through other computers within the same network.

    Initially, none of the computers are connected to each other. You are given a list of connections that you are requested to make in the form of a vector `requests`, where `requests[i] = [u, v]` is a request to add a network connection between computers `u` and `v`. A connection request is successful if `u` and `v` can be connected without breaking any of the restrictions specified in the `firewall` vector. Each request is processed in the order specified, and upon a successful request, the two computers involved retain their direct connection for all future requests.

    Return a boolean array `result`, where `result[i]` is true if the $\texttt{i}^{\text{th}}$ request is successful, and false if it is not. You may assume that requests to connect two computers that are directly connected are trivially successful.

    For example, given $n = 3$ computers, the following firewall restrictions: `[[0, 1]]`, and the following requests: `[[1, 2], [0, 2]]`, you would return `[true, false]`. This is because the first request is successful, since connecting computers 1 and 2 does not break any firewall rules. However, the second request is not successful, since computer 2 is now connected to computer 1 via the first request, and connecting 2 with 0 would break the restriction that computers 0 and 1 cannot be connected in any fashion, both directly and indirectly.

```
std::vector<bool> connection_requests(int32_t n, const std::vector<std::vector<int32_t>>& firewall,
                                      const std::vector<std::vector<int32_t>>& requests);
```

Chapter 13 Exercise Solutions

1. **The correct answer is (A).** A student who took EECS 281 once and passed in the spring semester would be included in S, a student who took EECS 281 in both the fall and winter semesters would be included in $F \cap W$, and a student who took EECS 281 in both winter and spring would be included in S. Only a student who took EECS 281 once in the winter would be included in neither $F \cap W$ nor S.

2. **The correct answer is (C).** Since the vectors are sorted, you can find the set difference by simply performing a linear pass of the two vectors, using the sorted order to determine if an element exists in one vector but not the other. See the implementation of `set_difference()` in section 13.1.2 for an example of how this can be done.

3. **The correct answer is (D).** If you are not allowed to create any additional containers, the best time complexity you can do is $\Theta(n\log(n))$, since you would need to presort the vectors before you can solve the problem using a linear pass (similar to the solution for question 2). Note that the best solution if both vectors remain unsorted takes $\Theta(n^2)$ time, since for each value in one vector, you would need to perform a linear pass over the other vector to identify if it also exists there. This is inferior to the $\Theta(n\log(n))$ cost of sorting the vector beforehand.

4. **The correct answer is (A).** Only statement I is required for the STL's set algorithms, as they rely on the assumption that the provided values are in sorted order. Statement II is false since these methods take in input iterators, and statement III is false because the two provided iterator ranges can be of different sizes (what happens to any extra values in the larger iterator range depends on the method called).

5. **The correct answer is (E).** The union find data structure is most useful for working with disjoint sets. All of the options are examples where this may be applicable, since they all deal with identifying or merging elements into disjoint sets.

6. **The correct answer is (B).** If every element in the data structure is forced to keep track of its ultimate representative, a single union call may require $\Theta(n)$ elements to have their representatives updated. This is the "quick-find" implementation of union find, which allows `find_set()` to be done in constant time at the expense of a more expensive `union_set()` call.

7. **The correct answer is (B).** An element has a representative equal to itself if it is the ultimate representative of its disjoint set. Since every element has its representative equal to itself, each element must be in its own disjoint set — because of this, a call to `union_set()` would merge two sets of size one, and thus will only be able to update one representative.

8. **The correct answer is (C).** The worst-case performance of `find_set()` depends on length of the representative chain from an element to its ultimate representative. Normally, this would be Θ(n) since you could have a representative chain in the form of a stick (where A is an ultimate representative, B's representative is A, C's representative is B, etc.). However, if you enforce that the smaller set is always updated to be part of the larger set after a union, then the maximum length of a representative chain drops down from Θ(n) to Θ(log(n)), which also reduces the worst-case time complexity of a single `find_set()` call.

9. **The correct answer is (C).** There are two disjoint sets: `{0, 2, 3, 5, 7, 8, 9}` and `{1, 4, 6}`. One strategy is to count the number of elements with a representative equal to itself, since this is the number of ultimate representatives that exist in the container.

10. **The correct answer is (B).** Since there are two disjoint sets, only one union operation is needed to bring the total number of disjoint sets down to one (by calling union on two elements in different sets). In general, if there are n disjoint sets, at least $n-1$ calls to union are needed to merge everything into a single set.

11. **The correct answer is (D).** Using the path-compression approach, if we were to call `find_set()` on an element j, we would traverse all elements on the way to its ultimate representative k and change all these elements to have a representative of k. Since we called `find_set()` on element 0, we follow the chain all the way to 0's ultimate representative of 8:

$$0 \to 5 \to 3 \to 2 \to 7 \to 9 \to 8.$$

Here, all of these elements would have their representatives changed to 8.

12. **The correct answer is (D).** Calling `find_set()` on an element would not change its representative if there are no intermediaries between such an element and its ultimate representative. This is only true for element 1, as 1's representative is 6, which is its own representative. Calling find on element 7 would change its representative fo 8, calling find on element 4 would change its representative to 6, and calling find on element 3 would change its representative to 8 (as well as others down the chain to 8).

13. **The correct answer is (C).** If there are 281 elements in the container, and 203 have themselves as their ultimate representative, then there must be 203 disjoint sets. If 202 of these sets have only one element, then there are 281 - 202 = 79 elements remaining to form the last disjoint set. The size cannot be larger than 79, or it would be impossible for there to be 203 disjoint sets in total.

14. **The correct answer is (A).** Since path compression is supported, the largest possible number of elements that could have their representatives updated is equal to the size of the largest disjoint set, minus 1 (since the ultimate representative of the set does not get updated), which could happen if you have a representative chain in the form of a stick, and you call `find_set()` on the ultimate representative and farthest element along the stick. The largest disjoint set can have a total of 370 - 279 = 91 elements, so the most number of representatives that could potentially be updated is 90.

15. **The correct answer is (E).** Option (A) is false because, from the existing representatives, there were only two disjoint sets before the union: {A, B, C, E} and {D}. This also means (B) and (C) are false as well. Of the remaining options, (E) is correct because the disjoint set containing {A, B, C, E} would not have an ultimate representative otherwise, as none of A, B, or C have its representative equal to itself.

16. This problem can be solved using a union-find container. To do so, we would first iterate over all the equality (==) equations and merge them into the same disjoint set. Then, we iterate over all the non-equality (!=) equations and call find to determine if they belong to different disjoint set. If not, then we have a contradiction, so we return false. Otherwise, if all the != equations involve variables that belong in different disjoint sets, we return true. One implementation of this solution is shown below:

```cpp
int32_t find(std::vector<int32_t>& reps, int32_t id) {
  if (id == reps[id]) {
    return id;
  } // if
  reps[id] = find(reps, reps[id]);
  return reps[id];
} // find()

bool are_equations_satisfiable(const std::vector<std::string>& equations) {
  std::vector<int32_t> reps(26);  // 'a' to 'z'
  for (int32_t i = 0; i < 26; ++i) {
    reps[i] = i;
  } // for i

  for (const std::string& equation : equations) {
    char var1 = equation[0] - 'a';
    char var2 = equation[3] - 'a';
    if (equation[1] == '=') {
      reps[find(reps, var1)] = find(reps, var2);
    } // if
  } // for equation

  for (const std::string& equation : equations) {
    char var1 = equation[0] - 'a';
    char var2 = equation[3] - 'a';
    if (equation[1] == '!' && find(reps, var1) == find(reps, var2)) {
      return false;
    } // if
  } // for equation

  return true;
} // are_equations_satisfiable()
```

17. This problem can also be solved using a union-find container, since we need to be able to quickly identify if two computers are on the same network (i.e., disjoint set). The solution is pretty similar to the previous problem: we iterate over the requests and check if we can make a connection without breaking any restrictions (using a call to find). If we can, then a connection is made (using a call to union). One implementation of this solution is shown below:

```cpp
int32_t find_set(std::vector<int32_t>& reps, int32_t id) {
  if (id == reps[id]) {
    return id;
  } // if
  reps[id] = find_set(reps, reps[id]);
  return reps[id];
} // find_set()

void union_set(std::vector<int32_t>& reps, int32_t x, int32_t y) {
  int32_t x_rep = find_set(reps, x);
  int32_t y_rep = find_set(reps, y);
  reps[y_rep] = x_rep;
} // union_set()

std::vector<bool> connection_requests(int32_t n, const std::vector<std::vector<int32_t>>& firewall,
                                      const std::vector<std::vector<int32_t>>& requests) {
  std::vector<int32_t> reps(n);
  for (int32_t i = 0; i < n; ++i) {
    reps[i] = i;
  } // for i

  std::vector<bool> result;
  for (const std::vector<int32_t>& request : requests) {
    bool can_connect = true;
    int32_t comp1 = find_set(reps, request[0]);
    int32_t comp2 = find_set(reps, request[1]);
    for (const std::vector<int32_t>& f : firewall) {
      int32_t firewall_rep1 = find_set(reps, f[0]);
      int32_t firewall_rep2 = find_set(reps, f[1]);
      if ((comp1 == firewall_rep1 && comp2 == firewall_rep2) ||
          (comp2 == firewall_rep1 && comp1 == firewall_rep2)) {
        can_connect = false;
        break;
      } // if
    } // for f

    result.push_back(can_connect);
    if (can_connect) {
      union_set(reps, comp1, comp2);
    } // if
  } // for request

  return result;
} // connection_requests()
```