Practice Exam – Model Solution

The Art of Computer Programming

Pierre Senellart

January 21, 2026

This document provides a model solution for the practice exam on *union–find* (disjoint-set) data structures.

A. Preliminaries

We successively build a partition of $\{0, 1, 2, 3\}$.

1. After inserting the singletons, we have

   $$\{\{0\}, \{1\}, \{2\}, \{3\}\}.$$

   Merging the equivalence classes of 0 and 2 yields

   $$\{\{0, 2\}, \{1\}, \{3\}\}.$$

   Merging the equivalence classes of 0 and 3 then yields

   $$\boxed{\{\{0, 2, 3\}, \{1\}\}.}$$

2. At the end of the sequence, representatives are displayed for the elements $0, 1, 2, 3$. Element 1 must be its own representative. The elements $0, 2, 3$ must all return the same representative, which may be any of 0, 2, or 3.

   Hence the possible outputs (in order $0, 1, 2, 3$) are

   $$\boxed{(r, 1, r, r) \text{ with } r \in \{0, 2, 3\}.}$$

3. The partitions of $\{0, 1, 2\}$ are:

   $$\{\{0\}, \{1\}, \{2\}\},\ \{\{0, 1\}, \{2\}\},\ \{\{0, 2\}, \{1\}\},\ \{\{1, 2\}, \{0\}\},\ \{\{0, 1, 2\}\}.$$

   There are therefore $\boxed{5}$ partitions (the number of partitions of a set of n elements is called the *Bell number* B_n, which grows as $1, 1, 2, 5, 15, 52, 203, 877, 4140 \ldots$).

4. Consider the set $\{1, 2, \ldots, n\}$. For each subset $S \subseteq \{2, \ldots, n\}$, define the partition

   $$\big\{\{1\} \cup S,\ \{2, \ldots, n\} \setminus S\big\}.$$

   Distinct subsets S yield distinct partitions. Since there are 2^{n-1} such subsets, the number of partitions is at least 2^{n-1}, hence in $\Omega(2^n)$.

B. Dynamic array

5. We store, for each element p, a representative of its equivalence class in an array T.

```python
class PartitionArray:
    def __init__(self):
        self._class = []

    def insert_singleton(self):
        n = len(self._class)
        self._class.append(n)

    def find(self, p):
        return self._class[p]

    def merge(self, p, q):
        if p == q:
            return
        for i in range(len(self._class)):
            if self._class[i] == q:
                self._class[i] = p
```

6. The asymptotic complexities are:
   - insert_singleton: $\Theta(1)$ (amortized, dynamic array append);
   - find: $\Theta(1)$ (array access);
   - merge: $\Theta(n)$, since the entire array may need to be scanned and updated.

C. Linked lists

7. Each equivalence class is represented by an object containing a `std::list<int>` of its elements. For each element, we store a pointer to the object representing its equivalence class.

   The `merge` operation concatenates the lists of two classes and updates the pointers of the moved elements. The destructor deallocates all dynamically allocated class objects.

```cpp
#include <vector>
#include <list>
#include <unordered_set>

class PartitionList {
    struct Cls {
        std::list<int> elems;
    };

    std::vector<Cls*> cls_of;          // element -> class
    std::unordered_set<Cls*> all_cls;  // to delete in destructor

public:
    PartitionList() {}

    ~PartitionList() {
```

```
        for (Cls* c : all_cls) delete c;
    }

    void insert_singleton() {
        int n = (int)cls_of.size();
        Cls* c = new Cls();
        c->elems.push_back(n);
        cls_of.push_back(c);
        all_cls.insert(c);
    }

    int find(int p) const {
        return cls_of[p]->elems.front();  // representative
    }

    void merge(int p, int q) {
        Cls* A = cls_of[p];
        Cls* B = cls_of[q];
        if (A == B) return;

        // append B into A and redirect pointers
        for (int x : B->elems) {
            A->elems.push_back(x);
            cls_of[x] = A;
        }

        all_cls.erase(B);
        delete B;
    }
};
```

8. The complexities are:
   - insert_singleton: $\Theta(1)$, amortized;
   - find: $\Theta(1)$;
   - merge: $\Theta(m)$, where m is the size of the class being merged (worst case $\Theta(n)$).

9. If we always merge the smaller class into the larger one, each time an element is moved, it goes into a class of at least double the previous size. So each element can be moved at most $O(\log n)$ times. Total work across any sequence of merges is $O(n \log n)$, hence amortized per merge is $O(logn)$. Hence the amortized complexity of `merge` becomes $\boxed{\Theta(\log n)}$.

D. Disjoint-set forest

10. One possible parent array for the partition $\{\{0, 2, 3\}, \{1\}\}$ is:

$$P[0] = 0, \quad P[1] = 1, \quad P[2] = 0, \quad P[3] = 0.$$

11. With this representation:
    - insert_singleton runs in $\Theta(1)$ amortized time;

- merge runs in $\Theta(1)$ time, as it only updates a constant number of values.

12. The find operation follows parent up to the root of the tree. Its complexity is therefore $\Theta(h)$, where h is the maximum height of a tree in the forest.

13. If merges are performed without any constraint, one can build a chain of n elements by repeatedly making the root of a tree a child of a new root: Start with singleton 0. Merge(0, 1): attach 0 under 1, resulting in height 1. Merge(root, 2): attach previous root under 2, resulting in height 2. Continue.

    The maximum height can thus be $n - 1 = \Theta(n)$.

14. Using union by rank, we always attach the root of smaller rank under the root of larger rank, and increase the rank only when merging two trees of equal rank. A tree of rank r then contains at least 2^r elements, so the rank (and thus the height) is at most $O(\log n)$.

15. We have the following complexities:

Data structure	insert_singleton	find	merge
Dynamic array	$\Theta(1)$ amortized	$\Theta(1)$	$\Theta(n)$
Linked lists (naive merge)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Linked lists (by size)	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$ amortized
Disjoint-set forest (no compression)	$\Theta(1)$ amortized	$O(\log n)$	$\Theta(1)$

    With this implementation, the disjoint-set forest is better for insertions and merge, but worse for find. Linked lists (by size) are better than dynamic arrays.

16. With union by rank and path compression, a Python implementation is:

```python
class UnionFind:
    def __init__(self):
        self.parent = []
        self.rank = []

    def insert_singleton(self):
        n = len(self.parent)
        self.parent.append(n)
        self.rank.append(0)

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def merge(self, p, q):
        if p == q:
            return
        if self.rank[p] < self.rank[q]:
            self.parent[p] = q
        elif self.rank[p] > self.rank[q]:
            self.parent[q] = p
        else:
```

```
            self.parent[q] = p
            self.rank[p] += 1
```

17. To compute the connected components of a graph (V, E):
    - create a disjoint-set forest with one element per vertex;
    - for each edge $\{u, v\} \in E$, apply $\mathsf{merge}(u, v)$;
    - vertices with the same representative belong to the same connected component.

    The total complexity is

    $$\boxed{O\big((|V| + |E|)\,\alpha(|V|)\big),}$$

    where α is the inverse Ackermann function.