Practice Exam

The Art of Computer Programming

Pierre Senellart

January 21, 2026

This is a practice exam, consisting of a single problem.

We consider data structures for representing a *partition* of a finite set of integers $\{0, \ldots, n-1\}$. Recall that a partition of a set X is a set $\{X_0, \ldots, X_{k-1}\}$ of subsets of X such that:

- $\bigcup_{i=0}^{k-1} X_i = X$;
- $X_i \neq \varnothing$ for all $0 \leqslant i < k$;
- $X_i \cap X_j = \varnothing$ for all $0 \leqslant i, j < k$ with $i \neq j$.

Each X_i is called an *equivalence class*.

The data structures we will handle must support the following operations to manage a partition $\mathcal{X}$ of a set $\{0, \ldots, n-1\}$:

constructor() Build an empty partition $\mathcal{X} = \varnothing$ of the empty set.

insert_singleton() Add a new integer n to the set $\{0, \ldots, n-1\}$ and add the singleton $\{n\}$ to the partition $\mathcal{X}$: we thus obtain a partition $\mathcal{X} \cup \{\{n\}\}$ of the set $\{0, \ldots, n\}$.

find(p) Given an integer p with $p \in \{0, \ldots, n-1\}$, return an integer p' that is a *representative* of the equivalence class of p in $\mathcal{X}$. In the case of repeated calls to this function, p' must be the same for all integers belonging to the same equivalence class.

merge(p,q) Given two integers p and q that are the representatives of the equivalence classes X_i and X_j, merge the equivalence classes X_i and X_j. Thus, $\mathcal{X}$ becomes $(\mathcal{X} \setminus \{X_i, X_j\}) \cup \{X_i \cup X_j\}$.

This is traditionally known as a *union–find* data structure. The goal of this problem is to study several ways of implementing such a data structure, and their complexity. Throughout the exam, we denote by n the number of elements in the set whose partition we are building, and we will therefore speak, for example, of an operation having complexity $\Theta(n)$ when this complexity is linear in the number of elements of the set.

A. Preliminaries

We wish to perform the following operations on a partition:

- We build an empty partition.
- We insert the singleton $\{0\}$.
- We insert the singleton $\{1\}$.
- We insert the singleton $\{2\}$.
- We insert the singleton $\{3\}$.
- We merge the equivalence classes of elements 0 and 2.
- We merge the equivalence classes of elements 0 and 3.
- We display a representative element for each of the elements 0, 1, 2, 3.

1. What is the partition represented by this data structure after all these operations?
2. What are all the possible outputs of this sequence of operations?
3. How many different partitions are there on the set $\{0, 1, 2\}$?
4. Show that the number of partitions of a set with n elements is in $\Omega(2^n)$.

B. Dynamic array

We first consider an implementation in which a partition is implemented by a *dynamic array* T of n cells, where the cell $T[p]$ is a representative of the equivalence class of p (chosen arbitrarily, but unique for all elements of that equivalence class).

5. Write such an implementation of a partition in the form of a Python class (with its constructor and its methods `insert_singleton`, `find`, `merge`). You may use a Python list as an implementation of a dynamic array.
6. What is the amortized asymptotic complexity of the operations insert_singleton, find, and merge with this implementation? Briefly justify.

C. Linked lists

We now consider an implementation in which each equivalence class is stored as a doubly linked list of the integers in the equivalence class. We also assume that we have a dynamic array that associates to each integer $0 \leqslant p < n$ the list node corresponding to that integer.

7. Write such an implementation of a partition in the form of a C++ class (with a constructor, destructor, and methods `insert_singleton`, `find`, `merge`). Each equivalence class may be represented by a `std::list` container stored in an object allocated on the heap. You may assume that the data structure maintains, for each element, a pointer to the object representing its equivalence class. The destructor should correctly deallocate all dynamically allocated equivalence-class objects.
8. What is the amortized asymptotic complexity of the operations insert_singleton, find, and merge with this implementation? Briefly justify.
9. By possibly modifying the operation `merge`, show that you can obtain a $\Theta(\log n)$ amortized complexity for that operation.

D. Disjoint-set forest

We now turn to a new implementation in which the partition is represented by a set of trees (this is called a *forest*). Each tree represents an equivalence class, and the nodes of the tree are the elements of that equivalence class. The representative of each equivalence class is the integer at its root.

Note that we can represent this forest as a single dynamic array that indicates the integer stored in the parent of every node; by convention, in that dynamic array, we can set the value of tree roots to themselves.

When two equivalence classes must be merged, one of the trees is made a child of the root of the other tree.

In addition, we associate with each tree root a *rank*: this rank is initially 0 and, when two trees are merged such that the height of the resulting tree is greater than that of both trees, the rank is increased by 1.

10. Represent the dynamic array representing one of the possible outputs of the union–find structure from Part A.

11. What is the complexity of the operations `insert_singleton` and `merge`? Justify.

12. What is the complexity of the `find` operation in terms of the maximum height h of a tree in the forest?

13. Show that if no additional constraint is imposed on how merging is performed, the maximum height of a tree in the forest can be $\Theta(n)$.

14. Exploiting the rank, propose an approach for merging (without changing the asymptotic complexity of the `merge` operation) that guarantees that the maximum height of a tree in the forest is in $O(\log n)$. Prove that this is indeed the case.

15. Compare the complexities obtained with this approach to those of Parts B and C.

16. This structure can in fact be further optimized as follows: each time the `find` operation is applied to a node u, the node u and each of its ancestors (excluding the root) are made direct children of the root.

    Implement this data structure in the form of a Python class, with this refinement applied to the `find` method and the refinement from Question 14 applied to the `merge` method.

17. We admit that the amortized asymptotic complexity of a sequence of operations with these optimizations is $O(\alpha(n))$, where α is the *inverse Ackermann function*, a function that grows extremely slowly (for all values of n one can imagine using in practice, $\alpha(n) \leqslant 4$).

    We now study the following problem: we are given a *graph*, that is, a finite set V of nodes and a finite set E of edges, each of which is a pair of nodes of V indicating a connection between them. We say that there exists a path from a node u to a node v if there exists a finite sequence of edges $\{u_0, u_1\}, \{u_1, u_2\}, \ldots, \{u_{\ell-1}, u_\ell\}$ in E such that $u_0 = u$ and $u_\ell = v$. A *connected component* is a maximal set of nodes C such that, for any $u, v \in C$, there exists a path from u to v.

    Show that one can use a disjoint-set forest to compute the set of connected components of a graph by proposing an algorithm. What is the complexity of this computation in terms of $|V|$ and $|E|$?

For more information about union–find, see Chapter 21 of the *Introduction to Algorithms* textbook.