●●●● Not Frequent 0/12

Introduction to Sets & Maps

Authors: Darren Yao, Benjamin Qi, Allen Li, Jesse Choe, Nathan Gong
Contributor: Elliott Harper

Maintaining collections of distinct elements/keys with sets and maps.

Language: Python Edit This Page

Prerequisites

- Bronze - Introduction to Sorting

TABLE OF CONTENTS

Introduction

RESOURCES

IUSACO	**4.4 - Sets & Maps**	module is based off this
CPH	**4.2, 4.3 - Sets, Maps**	covers similar material

A **set** is a collection of unique elements. Sets have three primary methods:

- one to add an element
- one to remove an element
- one to check whether an element is present

A **map** is a collection of entries, each consisting of a **key** and a **value**. In a map, all keys are required to be unique (i.e., they will form a set), but values can be repeated. Maps have three primary methods:

- one to add a specified key-value pairing

- one to remove a key-value pairing
- one to retrieve the value for a given key

C++ and Java both have two implementations of sets and maps; one uses **sorting** while the other uses **hashing.** Python's implementation of sets and maps uses hashing.

Sets

Distinct Numbers

CSES - Easy

Focus Problem – try your best to solve this problem before continuing! *View Internal Solution*

Sorted Sets

Sorted sets store elements in sorted order. All primary methods (adding, removing, and checking) run in $\mathcal{O}(\log N)$ worst-case time, where N is the number of elements in the set.

> **Warning!**
>
> Ignore this section, as Python doesn't implement sorted sets.

Hashsets

Hashsets store elements using hashing. Roughly, a hashset consists of some number of buckets B, and each element is mapped to a bucket via a hash function. If $B \approx N$ and the hash function independently maps each distinct element to a uniformly random bucket, then no bucket is expected to contain many elements, and all primary methods will all run in $\mathcal{O}(1)$ *expected* time.

> **Warning!**
>
> In the worst case, hashsets in Python may take proportional to N time per operation. This will be demonstrated later in the module.

Python's built-in `set` uses hashing to support $\mathcal{O}(1)$ insertion, deletion, and searches. Some operations on a Python `set` named `s` include:

- `s.add(x)` : adds element `x` to `s` if not already present
- `s.remove(x)` : removes an element `x` from `set` if present
- `x in s` : checks whether `s` contains the element `x`

Copy PYTHON

```python
s = set()
s.add(1)  # {1}
s.add(4)  # {1, 4}
s.add(2)  # {1, 4, 2}
s.add(1)  # {1, 4, 2}
# the add method did nothing because 1 was already in the set
print(1 in s)  # True
s.remove(1)  # {4, 2}
print(5 in s)  # False
s.remove(0)  # {4, 2}
# if the element to be removed does not exist, nothing happens
```

Solution - Distinct Numbers

This problem asks us to calculate the number of distinct values in a given list.

Method 1 - Sorted Set

Because sets only store one copy of each value, we can insert all the numbers into a set, and then print out the size of the set.

> Warning!
>
> Ignore this section, as Python doesn't implement sorted sets.

Method 2 - Hashset

Copy PYTHON

```
n = int(input())  # unused
nums = [int(x) for x in input().split()]
distinct_nums = set(nums)
print(len(distinct_nums))
```

Or we can do this more concisely by skipping the creation of the list, and use a set comprehension directly:

Copy PYTHON

```
n = int(input())  # unused
distinct_nums = {int(x) for x in input().split()}
print(len(distinct_nums))
```

However, it is possible to construct a test case that causes the above solution to run in $\Theta(N^2)$ time, so this solution does *not* receive full credit.

> Hack Case Generator

One way to get around this with high probability is to incorporate randomness; see **this comment** for more information.

Copy PYTHON

```
import random

RANDOM = random.randrange(2**62)


def Wrapper(x):
    return x ^ RANDOM


n = int(input())  # unused
distinct_nums = {Wrapper(int(x)) for x in input().split()}
print(len(distinct_nums))
```

Another (less efficient) way is to use strings instead of integers since the hash function for strings is randomized.

Copy PYTHON

```
n = int(input())  # unused
distinct_nums = set(input().split())
print(len(distinct_nums))
```

> Should I worry about anti-hash tests in USACO?
>
> No - historically, no USACO problem has included an anti-hash test. However, these sorts of tests often appear in Codeforces, especially in educational rounds, where open hacking is allowed.

Method 3 - Sorting

Check out the **solution** involving sorting.

Maps

Associative Array

YS - Easy

Focus Problem – try your best to solve this problem before continuing!

In sorted maps, the pairs are sorted in order of key. As with sorted sets, all primary methods run in $\mathcal{O}(\log N)$ worst-case time, where N is the number of pairs in the map.

In hashmaps, the pairs are hashed to buckets by the key, and as with hashsets, all primary methods run in $\mathcal{O}(1)$ expected time under some assumptions about the hash function.

Colloquially, hashmaps are referred to as **dicts** in python.

Copy PYTHON

```
d = {}
d[1] = 5  # {1: 5}
d[3] = 14  # {1: 5, 3: 14}
d[2] = 7  # {1: 5, 2: 7, 3: 14}
del d[2]  # {1: 5, 3: 14}
print(d[1])  # 5
print(7 in d)  # False
print(1 in d)  # True
```

Iterating Over Maps

To iterate over `dict`s, there are three options, all of which involve for loops. Dicts will be returned in the same order of insertion in **Python 3.6+**. You can iterate over the keys:

Copy PYTHON

```
for key in d:
    print(key)
```

Over the values:

Copy PYTHON

```
for value in d.values():
    print(value)
```

And even over key-value pairs:

Copy PYTHON

```
for key, value in d.items():
    print(key, value)
```

It's also possible to change the values while iterating over the keys (or over the values themselves, if they're mutable):

Copy PYTHON

```
for key in d:
    d[key] = 1234  # Change all values to 1234
```

While you are free to change the *values* in a map when iterating over it (as demonstrated above), it is generally a bad idea to insert or remove elements of a map while iterating over it.

For example, the following code attempts to remove every entry from a map, but results in a runtime error.

Copy PYTHON

```
d = {i: i for i in range(10)}

for i in d:
    del d[i]
```

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
```

```
    for i in d:
RuntimeError: dictionary changed size during iteration
```

One way is to get around this is to create a new map.

Copy PYTHON

```python
d = {i: i for i in range(10)}

# only includes every third element
d_new = dict(item for i, item in enumerate(d.items()) if i % 3 == 0)

print("new dict:", d_new)  # new dict: {0: 0, 3: 3, 6: 6, 9: 9}
```

Another is to maintain a list of all the keys you want to remove and remove them after the iteration finishes:

Copy PYTHON

```python
d = {i: i for i in range(10)}

# removes every third element
to_remove = {key for i, key in enumerate(d) if i % 3 == 0}

for key in to_remove:
    del d[key]

print("new dict:", d)  # new dict: {1: 1, 2: 2, 4: 4, 5: 5, 7: 7, 8: 8}
```

Solution - Associative Array

To solve this problem efficiently, we need a data structure that can:

- Assign a value to any index `k` (where `k` can be as large as 10^{18}).
- Retrieve the value at any index `k` quickly.

A regular array won't work because the indices can be extremely large, making it impossible to allocate enough memory. However, since all values are initially `0` and only a small subset of indices will ever be set or queried, we can use a **map** (also called an associative array or dictionary) to store only the indices that have been assigned a value.

- When we receive a `0 k v` query, we set `a[k] = v` in the map.
- When we receive a `1 k` query, we print `a[k]` if it exists in the map, or `0` otherwise.

This approach is efficient because both operations are fast in maps, and we only store the keys that are actually used.

Note that we use 64-bit integers since k and v may be large.

Unfortunately, the straightforward solution fails a few tests specifically designed to make Python `dict`s run slowly:

Copy PYTHON

```python
a = dict()  # Dictionary to store assigned indices
for _ in range(int(input())):
    nums = list(map(int, input().split()))
    if nums[0] == 0:
        a[nums[1]] = nums[2]
    elif nums[0] == 1:
        # Print a[k] if present, else 0
        print(a.get(nums[1], 0))
```

To pass all the tests, we can use one of the workarounds mentioned for `set`.

Copy PYTHON

```
5
6  def wrap(x):
7      return x ^ RANDOM
8
9
10 a = dict()  # Dictionary to store assigned indices
```

Problems

Some of these problems can be solved by sorting alone, though sets or maps could make their implementation easier.

STATUS	SOURCE	PROBLEM NAME	DIFFICULTY	TAGS
	CSES	Sum of Two Values	Easy	▸ Show Tags Map
	Bronze	★ Where Am I?	Easy	▸ Show Tags Set
	Bronze	Team Tic Tac Toe	Normal	▸ Show Tags Set, Simulation
	Bronze	Year of the Cow	Normal	▸ Show Tags Map
	Bronze	★ Don't Be Last!	Normal	▸ Show Tags Map, Sorting
	Silver	Cities & States	Normal	▸ Show Tags Map
	CF	Jury Marks	Normal	▸ Show Tags Prefix Sums, Set
	Bronze	It's Mooin' Time	Hard	▸ Show Tags Maps, Set
	AC	Made Up	Hard	▸ Show Tags Map
	CF	Into Blocks	Hard	▸ Show Tags Map, Set

Quiz

What is the worst-case time complexity of insertions, deletions, and searches in a `set` of size N?

1. $\mathcal{O}(1)$
2. $\mathcal{O}(N \log N)$
3. $\mathcal{O}(N)$
4. $\mathcal{O}(\log N)$