Hashing

Hashing is a technique that is frequently used in implementing efficient algorithms. In Python, the data structures `set` and `dict` (dictionary) are based on hashing.

In this chapter, we take a look at data structures based on hashing and their use in algorithm design. We will also cover some theory underlying the data structures.

Set

The Python data structure `set`, based on hashing, maintains a set of elements. The operations on the data structure include:

- the method `add` adds an element to the set
- the operator `in` finds if a given element is in the set
- the method `remove` removes an element from the set

The data structure is implemented so that all of the above operations take $O(1)$ time.

Example

The following code creates a set `numbers` and adds elements to the set:

```
numbers = set()

numbers.add(1)
numbers.add(2)
numbers.add(3)

print(numbers) # {1, 2, 3}
```

We can also create a set directly from a list:

```
numbers = set([1, 2, 3])

print(numbers) # {1, 2, 3}
```

The operator `in` tests if an element is in the set:

```
print(3 in numbers) # True
print(4 in numbers) # False
```

And we can remove an element from the set with the method `remove`:

```
print(numbers) # {1, 2, 3}
numbers.remove(2)
print(numbers) # {1, 3}
```

List vs. set

A list and a set are similar data structures in that both maintain a collection of elements and support additions and removals. However, there are significant differences in their efficiency and other properties.

Efficiency

Adding an element to a list is efficient, but finding an element and removing it can be slow.

With a set, adding elements, finding elements and removing elements are all efficient operations.

Operation	List	Set
Adding (`append` / `add`)	$O(1)$	$O(1)$
Finding (`in`)	$O(n)$	$O(1)$
Removing (`remove`)	$O(n)$	$O(1)$

Indexing

In a list, elements can be accessed using an index:

```
```

```
numbers = [1, 2, 3]
print(numbers[1]) # 2
```

A set does not support indexing:

```
numbers = set([1, 2, 3])
print(numbers[1]) # TypeError: 'set' object is not subscriptab
```

Repeated elements

In a list, an element can occur multiple times:

```
numbers = []

numbers.append(5)
numbers.append(5)
numbers.append(5)

print(numbers) # [5, 5, 5]
```

A set contains an element at most once. Adding an element that is already in the set has no effect:

```
numbers = set()

numbers.add(5)
numbers.add(5)
numbers.add(5)

print(numbers) # {5}
```

Example: How many numbers?

> **Task**

> You are given a list of numbers. How many distinct numbers does it contain?
>
> For example, when the list is $[3, 1, 2, 1, 5, 2, 2, 3]$, the desired answer is 4,

> because the distinct numbers are 1, 2, 3 and 5.

Slow solution (list)

We could solve the task using a list as follows:

```
def count_distinct(numbers):
    seen = []
    for x in numbers:
        if x not in seen:
            seen.append(x)
    return len(seen)
```

The algorithm goes through the numbers and adds a number to a list `seen` if it is not there already. At the end, the length of the list `seen` is the desired answer.

This algorithm is correct but not efficient, because every round of the loop calls the operator `in`, which can take $O(n)$ time. Thus the time complexity of the algorithm is $O(n^2)$. However, a simple improvement is to use a set instead of a list.

Efficient solution (set)

We can solve the task efficiently using a set as follows:

```
def count_distinct(numbers):
    seen = set()
    for x in numbers:
        if x not in seen:
            seen.add(x)
    return len(seen)
```

This function almost identical to the preceding one; the only differences are defining `seen` as a set instead of a list and using the method `add` instead of `append`. This change has a big effect on the efficiency of the algorithm. After the change, the operator `in` takes only $O(1)$ time and thus the time complexity of the algorithm is $O(n)$.

We can simplify the code further by using the fact that a set contains no duplicates. Thus we can remove the check if an element is already in the set:

```
def count_distinct(numbers):
    seen = set()
    for x in numbers:
        seen.add(x)
    return len(seen)
```

We can shorten the code further by creating the set directly from the list. Only one line is needed:

```
def count_distinct(numbers):
    return len(set(numbers))
```

Dictionary

The Python data structure `dict` or dictionary is based on hashing and stores key-value pairs. The idea is that we can use the key to retrieve the associated value.

A dictionary can be seen as a generalization of a list: In a list, keys are the indices $0 \ldots n$, while in a dictionary, keys can be arbitrary objects.

Adding, accessing and removing data using a key takes $O(1)$ time.

Example

The following code creates a dictionary `weights` where the keys are strings and the values are numbers.

```
weights = {}

weights["apina"] = 100
weights["banaani"] = 1
weights["cembalo"] = 500
```

The same dictionary can also be created as follows:

```
weights = {"apina": 100, "banaani": 1, "cembalo": 500}
```

The values in a dictionary can be used in the same way as the elements of a list:

```
print(weights["apina"]) # 100
weights["apina"] = 150
print(weights["apina"]) # 150
```

The operator `in` checks if a given key is in the dictionary:

```
print("apina" in weights) # True
print("ananas" in weights) # False
```

The command `del` removes a key and the associated value from a dictionary:

```
print(weights) # {'apina': 100, 'banaani': 1, 'cembalo': 500}
del weights["banaani"]
print(weights) # {'apina': 100, 'cembalo': 500}
```

Using a dictionary

We will next take a look at three common ways to use a dictionary in algorithm design.

Has an element occured

A dictionary can be used similarly to a set to keep track of elements that have been seen:

```
seen = {}
for x in items:
    seen[x] = True
```

This code has approximately the same functionality as the following code:

```
seen = set()
for x in items:
    seen.add(x)
```

Indeed, a set can be seen as a special case of a dictionary, where each key is associated with the value `True` (or any fixed value).

Counting occurrences

A common use of dictionaries is counting element occurrences:

```
count = {}
for x in items:
    if x not in count:
        count[x] = 0
    count[x] += 1
```

This code counts the number of occurrences of each element using the dictionary `count`. If the element is not yet in the dictionary, the code adds the element as a key with the initial count of zero as the associated value. Then the count is incremented by one for every occurrence of the element.

Position of occurrence

In some algorithms, it is useful to keep track of where each element has occurred.

```
pos = {}
for i, x in enumerate(items):
    pos[x] = i
```

Here the dictionary `pos` stores the index of the most recent occurrence of each element. Using the function `enumerate`, the code iterates through the list `items` so that in each round `i` is the index of an element and `x` is the element itself.

Example: Mode

> **Task**

> You are given a list of numbers, and your task is to compute the *mode*, which is the most frequent number on the list. If the mode is not unique, you can choose any of the possible choices for the most frequent number.
>
> For example, when the list is $[1, 2, 3, 2, 2, 3, 2, 2]$, the desired answer is 2.

We can solve the task efficiently by using a dictionary to count the number of occurrences:

```
def find_mode(numbers):
    count = {}
    mode = numbers[0]

    for x in numbers:
        if x not in count:
            count[x] = 0
        count[x] += 1

        if count[x] > count[mode]:
            mode = x

    return mode
```

Here `count` is a dictionary that stores the occurrence count for each element, and the variable `mode` stores the mode among the elements seen so far. Initially, `mode` is the first number on the list, and it is updated whenever the just updated count of an element exceeds the count of the current mode. Since the dictionary operations take $O(1)$ time, the time complexity of the algorithm is $O(n)$.

Here is another way to implement the algorithm:

```
def find_mode(numbers):
    count = {}
    mode = (0, 0)

    for x in numbers:
        if x not in count:
            count[x] = 0
        count[x] += 1

        mode = max(mode, (count[x], x))

    return mode[1]
```

Now the variable `mode` is a pair, where the first element is the occurrence count of the mode and the second element is the mode itself. For example, the value `(5, 2)` means that the number `2` has occurred `5` times.

The advantage of this implementation is that we can use the function `max` to update the mode. Here `max` uses the first element of the pair as the primary

comparison key and the second element as a secondary comparison key. Since the first element is the occurrence count, the pair with the larger count gets chosen by `max`.

Notice that the two functions may operate differently when there are multiple choices for the mode. The first function chooses the mode that reaches the final count first. The second function chooses the mode with the largest value, since the value of the mode is used as a secondary comparison key.

Example: Rounds

> **Task**

> You are given a list that contains the numbers $1, 2, \ldots, n$ in some order. Your task is to collect all the numbers in order from smallest to largest so that in each round you go through the list from left to right. How many rounds do you need?
>
> For example, the list $[3, 6, 1, 7, 5, 2, 4, 8]$ requires 4 rounds. The first round collects the numbers 1 and 2, the second round the numbers 3 and 4, the third round the number 5, and the fourth round the numbers 6, 7 and 8.

A useful observation is that a new round starts whenever the number to be collected next is to the left of the most recently collected number. In the example list above, the number 3 starts a new round because it is to the left of the number 2.

Slow solution (list)

The following algorithm solves the task using just the input list:

```
def count_rounds(numbers):
    n = len(numbers)

    rounds = 1
    for i in range(1, n):
        if numbers.index(i + 1) < numbers.index(i):
            rounds += 1

    return rounds
```

Here the number of rounds is computed into the variable `rounds`, which is initially set to 1. Then the loop goes through the numbers $1 \ldots n-1$ and increments `rounds` whenever the number $i+1$ occurs to the left of the number i.

The implementation uses the method `index` to locate the occurrence of a number. This makes the algorithm slow, because the method `index` needs $O(n)$ time and thus the total time complexity of the algorithm is $O(n^2)$.

Efficient solution (dictionary)

We can implement the same idea efficiently by using a dictionary to locate each number:

```
def count_rounds(numbers):
    n = len(numbers)

    pos = {}
    for i, x in enumerate(numbers):
        pos[x] = i

    rounds = 1
    for i in range(1, n):
        if pos[i + 1] < pos[i]:
            rounds += 1

    return rounds
```

Now locating a number needs only $O(1)$ time. Since the algorithm has two separate loops, each of which needs $O(n)$ time, the total time complexity of the algorithm is $O(n)$.

Example: Play list

> **Task**

> You are given a play list, where each song is represented by an integer. Your task to find out how long is the longest part of the play list that contains no song twice.

> For example, when the play list is $[1, 2, 1, 3, 5, 4, 3, 1]$, the desired answer is 5, which is the length of the play list part $[2, 1, 3, 5, 4]$.

A good approach to this task is to compute, for each position in the play list, how long is the longest non-repeating part ending at that position. The maximum of those lengths is the final answer. With the above example play list, these lengths are:

Song	1	2	1	3	5	4	3	1
Length	1	2	2	3	4	5	3	4

When we are at a given position in the play list and encounter a song that has occurred earlier, this may reduce the length of the non-repeating part, since any earlier occurrence of the song can not appear on the part. Thus the non-repeating part can start only after the nearest earlier occurrence. Using this principle, we can figure out the earliest possible start for the non-repeating part.

The following efficient algorithm implements these ideas:

```
def max_length(songs):
    n = len(songs)

    pos = {}
    start = 0
    length = 0

    for i, song in enumerate(songs):
        if song in pos:
            start = max(start, pos[song] + 1)
        length = max(length, i - start + 1)
        pos[song] = i

    return length
```

The dictionary `pos` stores the position of the last occurrence of each song. The variable `start` keeps track of the earliest possible starting position of a non-repeating part ending at the current position, and the variable `length` is the length of the longest non-repeating play list part we have found so far.

The algorithm goes through the play list and updates `start` whenever it encounters a song that it has seen before. In such a case, the value of `start` can increase to avoid a repeat of the song.

The time complexity of the algorithm is $O(n)$ thanks to the efficient dictionary operations based on hashing.

Example: List sums

> **Task**

> You are given a list containing n integers. Your task is to count, how many sublists of the list have x as the sum of its elements.
>
> For example, when the list is $[2, 3, 5, -3, 4, 4, 6, 2]$ and $x = 5$, the desired answer is 4. The sublists with sum x are $[2, 3]$, $[5]$, $[3, 5, -3]$ and $[-3, 4, 4]$.

A useful technique in this kind of a task is to consider the prefix sums of the list, i.e., to compute for each position the sum of the numbers from the beginning of the list to that position. In our example list, the prefix sums are as follows:

Index	0	1	2	3	4	5	6	7
Number	2	3	5	–3	4	4	6	2
Prefix sum	2	5	10	7	11	15	21	23

For example, the prefix sum at the position 4 is 11, because the numbers from the position 0 to the position 4 sum up to $2 + 3 + 5 - 3 + 4 = 11$.

Given the prefix sums, any sublist sum can be computed efficiently as a difference of two prefix sums. If a sublist starts at the position a and ends at the position b, the sublist sum is obtained by subtracting the prefix sum at the position $a - 1$ from the prefix sum at the position b.

In our example, the sublist that starts at the position 2 and ends at the position 4 has the sublist sum $5 - 3 + 2 = 6$. With the prefix sums, we can equivalently compute the sublist sum as the difference $11 - 5 = 6$.

The following algorithm is based on this technique:

```
def count_sublists(numbers, x):
    count = {0: 1}
    prefix_sum = 0
    result = 0
```

```
    for i in range(len(numbers)):
        prefix_sum += numbers[i]
        if prefix_sum - x in count:
            result += count[prefix_sum - x]

        if prefix_sum not in count:
            count[prefix_sum] = 0
        count[prefix_sum] += 1

    return result
```

Here the dictionary `count` is used for storing how many times each prefix sum has occured. With the dictionary, we can efficiently find out how many earlier prefix sums match the current prefix sum so that the difference is exactly x. The dictionary is initialized with the prefix sum 0 corresponding to the empty prefix list to account for the sublists starting at the beginning of the list.

The time complexity of the resulting algorithm is $O(n)$.

How does hashing work?

The Python data structures of this chapter, `set` and `dict`, are based on hashing and a data structure called the hash table. In python, a hash table is implemented using open hashing.

Python has a built-in function `hash` that is used for computing a hash value for an object. Python calls this function to determine the location of the object in a hash table. The function can be tested as follows:

```
> hash(42)
42
> hash(10**100)
9106852137541678 45
> hash("apina")
4992529190565255982
```

As the above shows, in Python, the hash value of a small integer is the integer itself. Otherwise, the hash values are random looking numbers.

The Python data structures based on hashing are usually efficient, and you can

assume that an addition, access or removal takes $O(1)$ time. However, there is a possiblity that hashing is slow if the input chosen in a specific way.

Which objects can be hashed?

The following code does not work in Python:

```
lists = set()
lists.add([1, 2, 3]) # TypeError: unhashable type: 'list'
```

The problem is that it is not possible to compute a has value for a list:

```
print(hash([1, 2, 3])) # TypeError: unhashable type: 'list'
```

A basic principle in Python is that a hash value can be computed only for an *immutable* object. A list is not immutable, because we can change the list with operations like `append`, and thus hashing a list is not possible.

Immutable objects in Python include numbers, strings and tuples consisting of immutable objects. For example, the following code works, because a tuple of numbers is immutable:

```
lists = set()
lists.add((1, 2, 3))
```

Notice that in a dictionary the hash value is computed only for the key and the associated value does not need to be hashable. An example of this is the following code with a string as a key and a list as a value:

```
lists = {}
lists["apina"] = [1, 2, 3]
```

Hashing for your own class

If you define your own class, you can apply hashing to it by defining the following methods:

- `__hash__`: returns the hash value of the object (the function `hash` calls this

method)
- `__eq__`: compares if two objects have identical content (the operator `==` calls this method)

The following shows an example of defining these methods. Here the method `__hash__` returns the hash value of a tuple representing the contents of the object.

```
class Location:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __hash__(self):
        return hash((self.x, self.y))

    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)
```

With these definitions, the following code works as expected:

```
locations = set()
locations.add(Location(1, 2))
locations.add(Location(3, -5))
locations.add(Location(1, 4))
```

Hashing in other programming languages

Data structures based on hashing are available in many programming languages. Often the data structure corresponding to a Python dictionary is called a *map*.

In C++, the data structures `std::unordered_set` and `std::unordered_map` implement a set and a map using hashing.

```
std::unordered_set<int> numbers;

numbers.add(1);
numbers.add(2);
numbers.add(3);
```

```
std::unordered_map<std::string, int> weights;

weights["apina"] = 100;
weights["banaani"] = 1;
weights["cembalo"] = 500;
```

In Java, the corresponding data structures are `HashSet` and `HashMap`:

```
HashSet<Integer> numbers = new HashSet<Integer>();

numbers.add(1);
numbers.add(2);
numbers.add(3);
```

```
HashMap<String, Integer> weights = new HashMap<String, Integer>

weights.put("apina", 100);
weights.put("banaani", 1);
weights.put("cembalo", 500);
```

And in JavaScript, the data structure `Set` implements a set:

```
let numbers = new Set();

numbers.add(1);
numbers.add(2);
numbers.add(3);
```

The traditional way to create a map in JavaScript is to define an object:

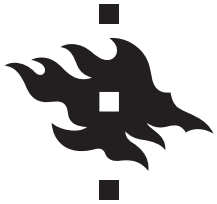
```
let weights = {};

weights["apina"] = 100;
weights["banaani"] = 1;
weights["cembalo"] = 500;
```

A newer way is to use a separate data structure `Map`:

```
let weights = new Map();

weights.set("apina", 100);
weights.set("banaani", 1);
weights.set("cembalo", 500);
```



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI