Arrays (and Hashmaps)

Brandon W

Contents

1 Overview **2**
1.1 Arrays 2
1.1.1 Big-O Notation 2
1.1.2 Creation 2
1.2 Hashmaps 4
1.2.1 Big-O Notation 4
1.2.2 Creation 4

2 Algorithms **6**
2.1 Binary Search 6
2.2 Sliding Window 8
2.3 Two Sum 11
2.4 Top K Frequent Elements 12

Chapter 1: Overview

1.1 Arrays

1.1.1 Big-O Notation

- **Space**: $O(n)$
- **Search**: $O(n)$
- **Access**: $O(1)$
- **Insertion**: $O(n)$
- **Deletion**: $O(n)$

1.1.2 Creation

Listing 1.1: Arrays in Python

```
# Create empty list
my_nums = []

# Create list with values
my_nums = [5,6,7]

# Create list from string
chars = list('Hello')
# chars => ['h', 'e', 'l', 'l', 'o']

# List from set
unique_set = set(1,2,3)
unique_list = list(unique_set)
# unique_list => [1, 2, 3]

// Size of unique_list
arrLen = len(unique_list)
```

Listing 1.2: Arrays in C

```c
int main() {
    // Create array with fixed size
    int array[5];
    array = {0, 1, 2, 3, 4};

    // Or in one fell swoop
    int array2[5] = {0, 1, 2, 3, 4};

    // With chars
    char myString = "Hello World";

    // Array with 5 items with malloc()
    int *myInts = (int) malloc(sizeof(int) * 5);

    // Size of myInts
    int arrLen = sizeof(myInts) / sizeof(myInts[0]);
}
```

Listing 1.3: Arrays in C++

```cpp
#include <vector>
#include <string>
using namespace std;

int main() {
    // Create a vector (a list) of strings
    vector<string> myStrings{ "hello", "world" };

    // Create a vector and add items after
    vector<int> myInts;
    myInts.push_back(7);
    myInts.push_back(2);

    // Size of myInts
    int arrLen = myInts.size();

    return 0;
}
```

1.2 Hashmaps

1.2.1 Big-O Notation

- **Space**: $O(n)$
- **Get (Access)**: $O(1)$
- **Contains Key (Access)**: $O(1)$
- **Insertion**: $O(1)$
- **Delete Key**: $O(1)$

1.2.2 Creation

Listing 1.4: Dicts (maps) in Python

```python
# Create empty dict
fruits = {}

# Add item to dict
fruits['apple'] = 7

# Create dict with items in it
fruits = {
    'apple': 10,
    'banana': 2
}

# Create dict with 'dict' builtin func
cat_names = dict(one='Ham', two='Beanbag')
print(cat_names['two']) # => Beanbag

# You can set a default value for keys
# using a default dict so you don't
# get an error when trying to access
# an undefined key.
#
# This is very helpful for
# frequency lists.
from collections import defaultdict

values = defaultdict(int)
values['a'] = 10

print(values['a']) # => 10
print(values['b']) # => 0
```

Listing 1.5: Maps in C++

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {

    map<string, int> fruits;
    fruits.insert(pair<string,int>("Apples", 10));
    fruits.insert(pair<string,int>("Bananas", 2));

    // Fetch specific value
    map<string,int>::iterator it;
    it = fruits.find("Apples");

    cout << it->first << ": " << it->second;
    // => Apples: 10

    return 0;
}
```

Chapter 2: Algorithms

2.1 Binary Search

Binary Search is a popular algorithm often used to find a target in an already-sorted list. The methodology goes as such:

- Create two variables that will act as a **left pointer** and a **right pointer**, with left initialized to 0 to represent the beginning of the array and right initialized to the last index of the array (size - 1)
- Start a while-loop for condition left <= right
- Calculate the middle index between left and right:
  middle = left + (right − left) / 2
- Check if we found target, if so return
- Check if current middle item is smaller than target. If so, discard the smaller (left-hand) half, and set our left pointer to 1 after the middle:
  left = middle + 1
- Otherwise, our current item is larger than the target. So we discard the larger half, and set our right pointer to 1 less than the middle:
  right = middle − 1

The time complexity is $O(log(n))$. This is because at every iteration of the loop, we are halving what part of the array we're looking at.

Listing 2.1: Binary Search In Python

```python
def binary_search(items, target):
    left = 0
    right = len(items) - 1

    while left <= right:
        # // in Python rounds to the nearest integer
        mid = (left + right) // 2

        if items[mid] == target:
            return mid
        elsif items[mid] < target:
            left = mid + 1
        else items[mid] > target:
            right = mid - 1

    return -1

binary_search([2,3,4,5,6,7,8,9], 3)
```

Listing 2.2: Binary Search In C++

```
int binary_search(vector<int>& items, int target) {
    int left = 0;
    int right = items.size() - 1;

    while (left <= right):
        int mid = left + (right - left) / 2;

        if(items[mid] == target) {
            return mid;
        } else if (items[mid] < target) {
            left = mid + 1;
        } else (items[mid] > target) {
            right = mid - 1;
        }

    return -1;
}
```

2.2 Sliding Window

Sliding Window is a technique used to search or traverse an array while looking at a subsection of the array (a "window") at a time. The goal of sliding window algorithms is to minimize nested loops and reduce their time complexities. These algorithms are used often for the following types of problems:

- Minimum / Maximum Sum Array
- Longest Sequence / Substring
- K Closest Elements

Assume we have a problem where we want to find the largest subarray in a given array:

Listing 2.3: Sliding Window In Python

```python
# k is the target size of the subarray
def maximum_subarray(nums, k):
    # Set the current maximum to lowest number
    max_sum = float('-inf')

    # Keep track of local sums
    current_sum = 0

    # Calculate the initial max sum
    # in the first window (first k items)
    for i in range(k):
        current_sum += nums[i]

    # Loop through the rest of array
    for i in range(k, len(nums)):
        # Update the largest subarray that we've seen
        max_sum = max(max_sum, current_sum)

        # Add the current num to the current_sum
        # Subtracting the first item from the last window
        current_sum += nums[i] - nums[i - k]

    return max_sum


maximum_subarray([5,8,3,6,9,1,0,9], 3)


"""
So for the above example, we effectively end up seeing:

Step 1:
Our window is [5,8,3], the first k elements.
The pointer is at index 3 (value of k), but we only look
at the values before that (exclusive).
```

```
    [5, 8, 3, 6, 9, 1, 0, 9]
     ^         ^
     |         |
   left     right

    Subarray (window) sum: 5 + 8 + 3 = 16

    Step 2:
    [5, 8, 3, 6, 9, 1, 0, 9]
        ^         ^
        |         |
      left     right
  """
```

Listing 2.4: Sliding Window In C

```c
#include <stdio.h>

int maximumSubarray(int* nums, int numsSize, int target);
int max(int, int);

int main() {
  int nums[8] = {5, 8, 3, 6, 9, 1, 0, 9};
  int numsSize = sizeof(nums) / sizeof(int);
  int maxSum = maximumSubarray(nums, numsSize, 3);
  printf("Maximum sub in subarray: %d", maxSum);
}

int maximumSubarray(int* nums, int numsSize, int k) {
  int maxSum = -__INT_MAX__;
  int currentSum = 0;

  for (int i = 0; i < k; i++) {
    currentSum += nums[i];
  }

  for (int i = k; i < numsSize; i++) {
    maxSum = max(maxSum, currentSum);

    currentSum += nums[i] - nums[i - k];
  }

  return maxSum;
}

int max(int a, int b) {
  if (a > b)
    return a;
  return b;
}
```

2.3 Two Sum

The two sum algorithm is a simple one used when needing to find if two numbers in a list can be added to equate to a given target.

For the simplest form of two sum (exemplified below), the algorithm goes as such:

- Loop through input
- Calculate the difference between the current item and the target
- If the complement is in the hashmap, return the current index and the index of the complement
- Add the current number to a hashmap as a key, who's value is the index

This can be done in $O(n)$ time. The reason this works is because at every iteration of our loop, we're seeing if we already came across a number (an appropriate complement of the current number) that will make the current number equal our target when added together.

Listing 2.5: Two Sum in Python

```python
def two_sum(nums, target):
    complements = {}
    for i in range(len(arr)):
        diff = target - nums[i]

        if diff in complements:
            return [i, complements[diff]]
        complements[nums[i]] = i

two_sum([9,6,11,15], 17)
```

Listing 2.6: Two Sum in C++

```cpp
vector<int> twoSum(vector<int>& nums, int target) {
    map<int, int> complements;
    vector<int> result;

    for(int i = 0; i < nums.size(); i++) {
      if (complements.count(target - nums[i]) > 0) {
        return {i, complements[target - nums[i]]};
      }
      complements.insert(pair<int,int>(nums[i], i));
    }

    return {-1, -1};
}
```

2.4 Top K Frequent Elements