Table of Contents

About

I feel that being a successful software developer requires a solid knowledge of fundamental Data Structures and Algorithms. However it is also important to be familiar with your tools of trade (your programming language / environment). This is the book that a CS student can use to refresh their knowledge of Data Structures and Algorithms or just review how to do them in TypeScript.

The code in this book should also provide idiomatic TypeScript code samples. Should should be able to see the advantages that having a Type system can bring to documenting your code.

> All the code + markdown files can be found on GitHub

Getting Started

Code

You need NodeJS. Go ahead and install the latest version you can find.

All code is writen in TypeScript *nightly*. After all, the compiler test suite is more reliable than our code bases

```
npm install typescript@next
```

Run the compiler

```
tsc -w
```

Run any file you want now:

```
node someCode.ts
```

Sections

I recommend going through the book page by page. Or jump straight to the section you want. If you feel you aren't getting it...go a step back.

Basics

Some background information and tips that I feel you should know to be comfortable with algorithms and data structures.

General Tips

Avoid off by one errors

Just solve with small numbers and the addition / subtraction gives you clue towards if you need to add / remove a `+1`.

Random Numbers

You can get random number in `O(1)` (for asymptomatic analysis). Operating systems generally have an entropy pool (using user input / cpu usage etc) that they provide to programs that need them (either natively or through the virtual machine of your language).

JavaScript `Math.random` provides you with a random floating point number between `range [0, 1)`. To get an integer just use the following function:

```
function getRandom(floor:number, ceiling:number) {
    return Math.floor(Math.random() * (ceiling - floor + 1)) + floor;
}
```

`+ floor` ensures its at least floor and then we just generate a random number between `[ 0 -- (ceiling - floor + 1) )`.

> Math.random is *pseudo-random* but good enough for most cases. For true randomness (entropy pool version) and security critical applications you need something like NodeJS `crypto`.

Powers of two

You need to know the powers of two like the back of your hand.

The most imporant one that people test on are `2^10`. Its generally safe to assume `1000` (instead of exact `1024`).

- You will be asked to use it for stuff like `log2 (n)`. A thousand is `10` a million is `20` a billion is `30`
- You will use it to get KB (`/1000`) MB (`/1000000`) etc.

> Note KB/MB/GB/TB are used for *bytes*. Its best to stick with this for napkin discussions.

Infinity

Probably not useful. But its fun so why not!

Hilberts Hotel

You have a hotel with infinite room and each room has a guest. A room can only have a single guest. A new guest shows up. Can you provide a room for this new guest?

Solution

Yes. Just take the person from the first room and move him to the next room. Move the person from the next room to the third room and so on. Now you have a spare room (first) for the new guest. In fact you can take infinite guests.

> Note: `1 + ∞ = ∞` , `∞ + ∞ = ∞` . There is lots of more weird stuff around ∞.

Data Structures

Algorithms that provide data storage and retrival with certain properties.

Stack

A stack is a last in first out (LIFO) data structure.

Implementation

Trivial, you just use JavaScript array `push` / `pop` . Or if you want to encapsulate it in a class:

```
class Stack<T> {
  _store: T[] = [];
  push(val: T) {
    this._store.push(val);
  }
  pop(): T | undefined {
    return this._store.pop();
  }
}
```

Queue

A queue is a first in first out (FIFO) data structure.

Implementation

Trivial, just use a JavaScript array `push` / `shift` methods. Or if you want to encapsulate it in a class:

```
class Queue<T> {
  _store: T[] = [];
  push(val: T) {
    this._store.push(val);
  }
  pop(): T | undefined {
    return this._store.shift();
  }
}
```

Shuffling

Shuffling is a procedure used to randomize an array.

> The key property is that each item should have an equal probability to end up in any index.

Solution

The recommended (simple) algorithm is the `Fisher–Yates shuffle`. Its time complexity is `O(n)`. It can even be done *inplace*.

You go from `0 - (n-1)` and at each index `i` pick a random number between `0` - `n-i` and move the element at the result into the `i + thisRandomNumber` th location in the array.

Maintains Key Property

It maintains that key property as:

```
Probablility any item makes it to the first position
= 1/n

Probablility any item makes it into the second position
=  didn't make it into the first * makes it into the second
=  (n-1)/(n) * 1/(n-1) = 1/n

... so on
```

Has complexity `O(n)`

Remember random number generation / assiging an item to an array is `O(1)`, so its just `n` iteration of `O(1)`

Code

```
function shuffleInPlace<T>(array: T[]): T[] {
  // if it's 1 or 0 items, just return
  if (array.length <= 1) return array;

  // For each index in array
  for (let i = 0; i < array.length; i++) {

    // choose a random not-yet-placed item to place there
    // must be an item AFTER the current item, because the stuff
    // before has all already been placed
    const randomChoiceIndex = getRandom(i, array.length - 1);

    // place our random choice in the spot by swapping
    [array[i], array[randomChoiceIndex]] = [array[randomChoiceIndex], array[i]];
  }

  return array;
}
```

Sorting in TypeScript

Given an array of `T` you are mostly best off using the built-in `sort` method (`O(n Log n)`). There are two warnings with this function:

- You generally want to always provide a `compare` function **even for numbers** otherwise the default compare function is lexicographical (i.e. an alphabetical sort after doing a `toString` on each number which doesn't work well).
- This function mutates the array in place.

```
let xs = [5,4,3,-22,1];
xs.sort((a,b)=>a-b); // Ascending sort
console.log(xs); // [-22,1,3,4,5]
```

Of course for descending you can use `b-a` (or use `.reverse`)

Insertion Sort

> Assuming western Left to Right reading card players of course

This is how humans naturally sort cards. Think about it, you go from left to right and step by step put each card in its *right place* (aka do an `insert`).

```
/**
 * Time complexity: O(N^2)
 */
function insertionSort<T>(
    array: T[],
    cmp: { (a: T, b: T): number } = (a: any, b: any) => a - b
): T[] {
    var current: T;
    var j: number;
    for (var i = 1; i < array.length; i += 1) {
        current = array[i];
        j = i - 1;
        while (j >= 0 && cmp(array[j], current) > 0) {
            array[j + 1] = array[j];
            j -= 1;
        }
        array[j + 1] = current;
    }
    return array;
}
```

Quiz

Some simple problems common to interviews with simple solutions.

Reverse a string

JavaScript `string` doesn't have a `reverse` method. But you can just use the `Array.prototype.reverse` method.

```
function reverse(str:string) {
  return str.split('').reverse().join('');
}
```

Palindrome

An Palindrome is a word, phrase, number, or other sequence of characters which reads the same backward or forward.

Problem 1

Tell if a word is a palindrome

Solution

You just need to check if the reverse of a string is the same as the string.

```
function isPalindrome(str:string) {
  return str.split('').reverse().join('') === str;
}
```

Since `reverse` is not a function on a JavaScript string you just leverage the one on `Array`

Fizzbuzz

Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.

Solution

You need to loop *right* `1-100` and do a check on both `three` and `five`

```
for (let i = 1; i <= 100; i++) {
  const t = i % 3 == 0, f = i % 5 == 0;
  console.log(
    t && f ? "FizzBuzz"
    : t ? "Fizz"
    : f ? "Buzz"
    : i
  );
}
```