# Better Programming

Chuncheng Zhang

March 26, 2021

**Abstract**

Every one has *computer*. It computes at a very high speed. Basically, you put your data in *objects*, and generate *functions* to do the computation. The *languages* are the communication between you and the computer. And the *algorithms* are the tools of how to do the computation precisely and quickly. A good *programming* is about all of the above, you have to manage them, like the arm controls the hand.

## Contents

## 1 Computer

The computer is PC or laptop in real-world. But in programming, the computer can be a very *abstraction* concept. Four components are necessary:

- Input
  Used for user input
- Output
  Used for output to screen or speaker
- Memory
  Where the variables stay during computing
- Disk
  Where the files stay forever

### 1.1 Input & Output

The input and output are the interfaces between the computer and users. Input means message from user to computer, and output means the reverse. They are applied in different syntax in different language. Take *Python* and *JavaScript* for examples.

Listing 1: InputOutput.py

```python
'''
File: InputOutput.py
Aim: Example of input and output in Python
'''

# Input
inp = input('Input:')

# Output
print(f'You just inputted: {inp}')
```

Listing 2: InputOutput.js

```javascript
/*
File: InputOutput.js
Aim:  Example of input and output in JavaScript
*/

// Input
const inp = prompt("Input:");

// Output
console.log("You just inputted:", inp);
```

### 1.2 Memory

When you practice programming, all the variables, functions and objects in your code is in the memory. In another word, the computation equals to the operation to the memory. More about memory can be found in the section of Objects. In current stage, all you need to know is everything you program is in the memory.

### 1.3 Disk

When computing is finished, users may have their stuff to be stored forever. The disk is where to put them.

It should be noticed that it can be very different between the things in memory and their storage in disk. For example, an article is structured as a characters array in the memory. However, it is stored as a highly compressed binary series in the disk. Although the difference between the two formats, they are the same article in fact.

At the viewpoint of the memory, there are fine programs to save the data to and read the data from the disk. In ideal condition, the two-way process is *transparent* to the user, which it frees the users to think about the conversion, thus the users can focus on the object in memory during computation.

## 2 Objects

The object is an overall calling to the things of interest. It can be a number, character, string, list or set. Moreover, it can even be a collection of them.

When you are thinking about a object, you are actually summarizing its features in the mind. But the computer works in the real world. That is a large separation.

> *In the abstraction level*, the object is the summary of features.
> *And in the concrete level*, the object is an instance of features.

The gap causes several problems.

### 2.1 Effect of precision

Basically, a number has two features, the value and the precision. In abstraction level, 100/3 is an existing number. However, in the real world, the computer can not represent it with infinite precision. As a result, every time you put your hand to it in the program, it shows as the given precision, and the output value can be different.

Listing 3: Precision.js

```
/*
File: Precision.js
Aim: Example of a number
*/

// The number of 100 / 3
const a = 100 / 3;

// The int precision
// 33
console.log(parseInt(a));

// The float precision
// 33.333333333333336
console.log(parseFloat(a));
```

Fortunately, the *float* precision is far beyond to meet the standard of daily usage. Evenly, in modern programming language, like JavaScript and Python, the precision can be intelligent assigned to the variables without causing problems in most cases, which is largely convenience to the users.

The effect of precision is only a small piece of the difference between the object in your mind and its instance in your computer. Keep it in your mind when practicing the programming, and it will save your from lots of unnecessary troubles.

## 2.2 Data Organization

Before you do anything with your computer, you need to understand the organization of data, since it may be too large to look at and place them one-by-one. Moreover, the data can be arbitrary on the using end. A good program is designed to deal with different values and sizes in the given data structure. As a result, the first thing of programming is in two folder:

- Decide how to organize the data;
- Estimate how long the operation will take at typical situation.

Does it look like two things? Yes, but actually No. I believe it is one thing with the name of *Data Structure*.

### 2.2.1 Example of Scoring

Think of a simple program to score the kids with their raw scores.

Listing 4: Score.js

```
/*
File: Score.js
Aim: Example of the scoring object
*/

// We think the obj is the collection of scores.
// It records the raw scores and knows how to convert them
    into discrete scores.

// The raw scores
let obj = [79, 54, 80, 90];

// The scoring thresholds
obj.thresholds = {
    A: 90,
    B: 80,
    C: 70,
    D: 60,
    E: 0,
};

// Scoring the raw scores
obj.score = (e, i, t) => {
    for (let s in t.thresholds) {
        if (e >= t.thresholds[s]) {
            return [i, s];
        }
    }
};

// See what we got
// [ [ 0, 'C' ], [ 1, 'E' ], [ 2, 'B' ], [ 3, 'A' ] ]
console.log(obj.map(obj.score));
```

The script can be used to generate discrete scores based on the raw scores. As it writes, one can tell the discrete score based on the raw score value using the threshold table. It is quite short and simple. But it contains two classic data structures, the coder can solve more than 80% computation problems with them. They are *chain list* and *hash table*.

### 2.2.2 Chain List

A chain list contains countless nodes, the length can be very large. They are linked one after another like the nodes in a chain. It is almost the simplest data structure in the computer. The advantage is memory saving and easy to access to certain position, and the pitfall is it can be slow for query the certain value.

Listing 5: Array.js

```
/*
File: Array.js
Aim: Example of array.
     Explain why it is good or bad.
*/

// Generate array
// Assume it is very long, like 1,000,000 elements.
let arr = [4, 5, 70, 29, ...., ];

// Example of Good
// Access to the 7,364th element,
// it only requires ONE operation.
console.log("The 7,364th element is", arr[7363]);

// Example of Bad
// Tell if 365 is in the array, and where it first appears,
// it may require 1,000,000 operations to find an answer.
let found = false;
for (let i = 0; i <arr.length; i++) {
    if (arr[i] === 365) {
        console.log("Found 365 in the array at", i)
        found = true;
        return
    }
}
if (!found) {
    console.log("The 365 is not in the array")
}
```

The code describes the thing. It can be time consuming to find out if certain value exists. Commonly, it will find it after too many operations or report it can not be found in the list. At the worst condition, the computer has to check every node in the chain list to find the given value. As a result, the expectation of the time consuming is

$$\mathcal{O}(n) \tag{1}$$

where $n$ refers the length of the chain list. It is a common notion of time consuming estimation in the field of *computational complexity*.

> Computational complexity! As we know, Computation is the process of calculating something by mathematical or logical methods. Execution of an algorithm is a well-known example of computation. In order to perform any sort of computation, we will need energy. This energy can be directly related to the resources that we spend to perform computation.
>
> *Data from: Wikipedia*

It refers the time consuming can be estimated by $n$

$$\mathcal{E}(t) \propto f(n) + c$$

where $f(\cdot)$ refers a linear function, and $c$ is constant independent with $n$. As a result, when the $n$ is too large, the program may use up countless computation times.

No one wants to wait almost forever to just find out some value. The question is how to speed up the process? The hash table is one solution.

### 2.2.3 Hash Table

The thresholds in Listing 4 is a hash table. As it writes, the elements in the hash table has two factors:

- Key: The unique key to access the record;
- Value: The value belonging to the key.

The hash table is used for quickly access to the value by certain key.

The idea of hash table is to use its hash value to identify the key and thus find its value. Since the key's location is computed rather than searched by exhaustively search [1]. The process can be expressed as

$$h = \mathbb{H}(key)$$
$$i = \mathbb{L}_n(h)$$

where $\mathbb{H}$ is hash function to compute the hash value [2], and $\mathbb{L}_n$ is projection function to compute its location $i$ in the $n$ sized hash table. A qualified hash system of $(\mathbb{H}, \mathbb{L}_n$ can map the keys into the hash table evenly. That says the keys will be located to $n$ locations with equal probability.

Using the hash method, the key-value pairs are separated evenly into the $n$ length hash table. When query a key, the program only has to compute its hash value and find it from the location. Assume there $m$ pairs in the hash table, the computation complexity is

$$\mathcal{O}(\frac{m}{n}) \tag{2}$$

[1]Exhaustively search: Including or considering all elements or aspects; fully comprehensive. Like search one-by-one in a chain list.

[2]Hash value is usually a string.

The computation is because the exhaustively search process only has to deal with the list with length of $\frac{m}{n}$ in ideal condition [3]. As a result, the hash table is commonly used for storage large data for arbitrary query.

Below is an example for explanation.

Listing 6: HashTable.js

```
/*
File: HashTable.js
Aim: Example of hash table.
*/

// The size of the table
const size = 7;

// The hash seed parameter
const I64BIT_TABLE = "0123456789abcdef".split("");

// Compute hash value
function hash(input) {
    var hash = 5381;
    var i = input.length - 1;

    if (typeof input == "string") {
        for (; i > -1; i--) hash += (hash << 5) + input.
            charCodeAt(i);
    } else {
        for (; i > -1; i--) hash += (hash << 5) + input[i];
    }
    var value = hash & 0x7fffffff;

    var retValue = "";
    do {
        retValue += I64BIT_TABLE[value & 0xf];
    } while ((value >>= 6));

    return retValue;
}

// Convert hash value to idx of hash table
function hash2idx(h) {
    const hex = "0x" + h;
    return parseInt(hex) % size;
}

// Init empty hash table
let table = [];
for (let i = 0; i < size; i++) {
    table[i] = undefined;
}

// The pairs to be restored in the hash table
const pairs = {
```

[3]Ideal refers the hash function is qualified.

```
    AliceBlue: "#F0F8FF",
    AntiqueWhite: "#FAEBD7",
    Aqua: "#00FFFF",
    Aquamarine: "#7FFFD4",
    Azure: "#F0FFFF",
    Beige: "#F5F5DC",
    Bisque: "#FFE4C4",
    Black: "#000000",
    BlanchedAlmond: "#FFEBCD",
};

// Insert the pairs in to the hash table
console.log("Inserting hash table");
for (const name in pairs) {
    const h = hash(name);
    const i = hash2idx(h);
    console.log("\t", name, "->", h, "->", i);

    if (table[i] === undefined) {
        table[i] = [[name, pairs[name]]];
    } else {
        table[i].push([name, pairs[name]]);
    }
}

// The generated hash table
console.log("Hash table is");
console.log(table);

/*
The output reads as following:
Inserting hash table
         AliceBlue -> bc4bc1 -> 1
         AntiqueWhite -> db6231 -> 4
         Aqua -> d345c1 -> 2
         Aquamarine -> 9c1c31 -> 4
         Azure -> cf28f -> 1
         Beige -> 18f7f -> 1
         Bisque -> e42fb1 -> 2
         Black -> 2931f -> 0
         BlanchedAlmond -> 1b398 -> 2
Hash table is
[ [ [ 'Black', '#000000' ] ],
  [ [ 'AliceBlue', '#F0F8FF' ],
    [ 'Azure', '#F0FFFF' ],
    [ 'Beige', '#F5F5DC' ] ],
  [ [ 'Aqua', '#00FFFF' ],
    [ 'Bisque', '#FFE4C4' ],
    [ 'BlanchedAlmond', '#FFEBCD' ] ],
  undefined,
  [ [ 'AntiqueWhite', '#FAEBD7' ], [ 'Aquamarine', '#7FFFD4' ]
      ],
  undefined,
  undefined ]
```

```
*/
```

### 2.3 Stack & Heap

# 3 Functions

# 4 Languages

# 5 Algorithms

# A Computation Complexity