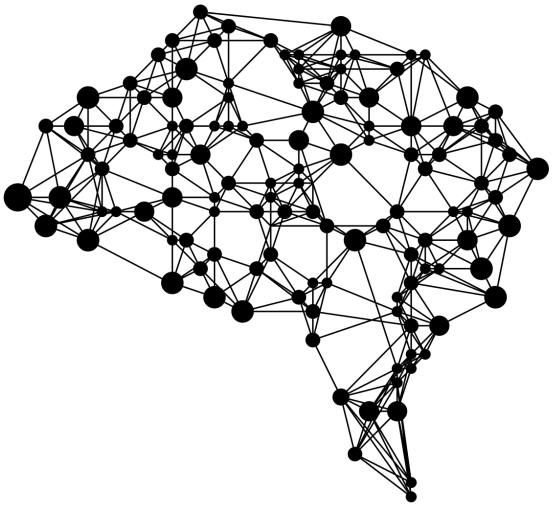


Algorithms

Mingming Li

First Created: September 29, 2020
Last Modified: December 15, 2022

Contents

List of Figures

List of Tables

Part I

Theory

Chapter 1

Algorithm

1.1 What is Algorithm?

The following relation show what is an algorithm.

```
input -> algorithm -> output
```

An algorithm is a sequence of computational steps that transform the input into the output. An algorithm describes a specific computational procedure for achieving the input/output relationship.

1.2 Instance of a Problem

An instance of a problem is the input needed to compute a solution to the problem.

1.3 Correct Algorithms

For every input instance, the algorithm halts out the correct output. We say the algorithm is correct.

1.4 Two Characteristics of Many Algorithms

- ♠ There are many candidate solutions, but finding the one that solve or the one is best is challenge.
- ♠ They have practical applications.

1.5 Data Structure

Data structure is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and it is important to know the strengths and limitations of several of them.

1.6 The Core Technique

learn the technique of algorithm design and analysis.

1.7 Hard Problems

Like the NP-complete problem, there are problem that has no efficient solutions. Before you delve into the real problem, take a overview of it.

1.8 Algorithm Efficiency

Computers are not infinitely fast and memory is not free, thus the efficiency of a algorithm matters.

1.9 Algorithms as a Technology

Algorithms are at the core of most technologies.

1.10 Loop Invariant

Loop invariant is used to help us to understand why an algorithm is correct. The three elements of loop invariant:

Initialization It is true prior to the first iteration of the loop.
Maintenance If it is true before an iteration of the loop, it remains true before the next iteration.
Termination When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

1.11 Analyzing Algorithms

Analyzing an algorithm is to predict the resources usage. Resources include time and space (memory, communication bandwidth, computer hardware, computational time...).

1.12 Resource Model

Before analyzing an algorithm, there must be a model to measure the resource cost.

1.13 The Information You Used in You Algorithm

The are much information in you problem. The general is that the more information you use, the more efficient your algorithm is. Data structure is on way of using the information in you problem.

1.14 Core Idea in Modeling

When you do algorithm modeling, remember to show the important characteristics of algorithms and suppress the tedious details.

1.15 Analysis of a Algorithm

In general, the time grows with the size of the input, so it is traditional to describe the running time as the function of the size of its input.

1.16 Worst-Case Analysis

Because the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

The reason to analyze worst-case running time is:

- ♠ It gives an upper bound on the running time.
- ♠ Worst case ocurrs fairly often.
- ♠ The "average case" is often roughly as bad as the worst case.

1.17 Abstraction

We often use some simplifying abstractions to ease the algorithm analysis. These abstractions are:

- ♠ Ignore the actual cost of each statement, using the constants c_i to represent these costs.
- ♠ Ignore the abstract costs c_i ($an^2 + bn + c$).
- ♠ We only use rate of growth or order of growth of the running time ($\Theta(n^2)$) (pronounced "theta of n-squared") instead of exact running time function.

1.18 Growth of Functions

Although we can sometimes determine the exact running time of an algorithm, the extra procision is not usually worth the effort of computing it. When we look at input sizes large enought to make only the order of growth of the running time relevant, we are studying the "asymptotic efficiency of algorithms".

Part II

Data Structure

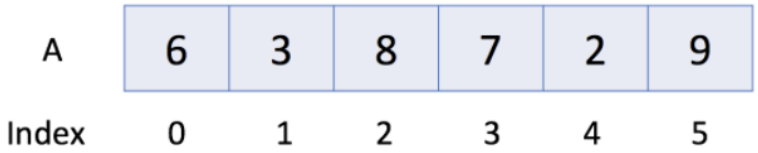
Chapter 2

Array

2.1 What is an array?

Arrays are collections of data that are stored in adjacent memory locations, so data elements are all available at random as each element can be identified with an array index.

For example in Figure 2.1:



Figures 2.1: Array index

In the example above, there are 6 elements in the array A, i.e. the length of the array is 6. We can use $A[0]$ to indicate the first element in the array A, so$A[0] = 6$. Similarly, $A[1] = 3$.

2.2 Capacity and length

The capacity is the maximum amount of data the array can hold, specified when you create the array. The length is the amount of data that the current array holds.

2.3 Operations

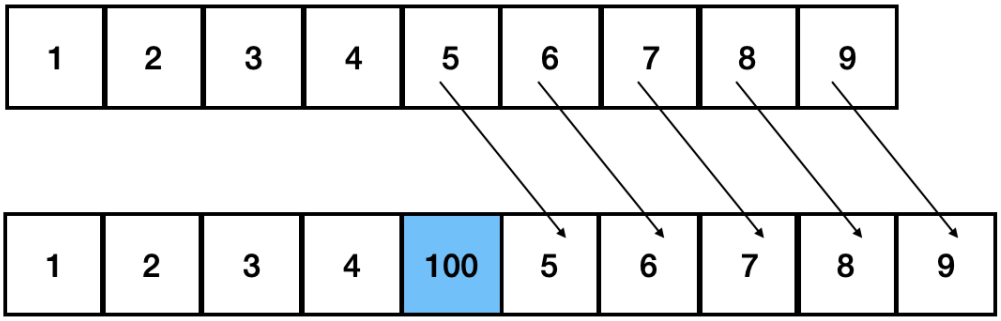
An array is a data structure, which means it stores data in a specific format and supports specific operations on the data.

There are 3 operations on arrays:

- ♠ insert
- ♠ delete
- ♠ search

2.4 Insert

The insert into an array is like the Figure 2.2



Figures 2.2: Insert into an array

Suppose the length of the array A is n and the start index is 0. You can insert an element x at index $0, 1, \ldots, n$. x will be inserted into array A randomly so the probability to insert at $i (0 \leq i \leq n)$ is the same. Suppose the insert operation and movement operation take constant time 1. The time complexity T will be:

$$\begin{aligned} T &= \frac{1}{n+1}(n+1+n+\ldots+1) \\ &= \frac{1}{n+1} \cdot \frac{(n+1)(n+2)}{2} \\ &= \frac{n+2}{2} \\ &= \frac{1}{2}n+1 \end{aligned}$$

The average time complexity of array insertion is $O(n)$, and the space complexity is $O(1)$.

2.5 Delete

The average time complexity of array deletion is $O(n)$, and the space complexity is $O(1)$.

2.6 Search

For an unordered array, the time complexity of searching for an element in it is $O(n)$.

2.7 Strings

A String is an array whose elements are characters.

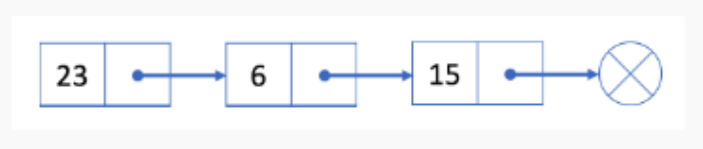
2.8 Examples

Here are some codes about array on Github

Chapter 3

Linked list

Similar to the array, the linked list is also a linear data structure. Here is an example in Figure 3.1 and 3.2:



Figures 3.1: Singly linked list

Figures 3.2: Doubly linked list

As you can see, each element in the linked list is actually a separate object while all the objects are linked together by the reference field in each element.

3.1 Singly linked list

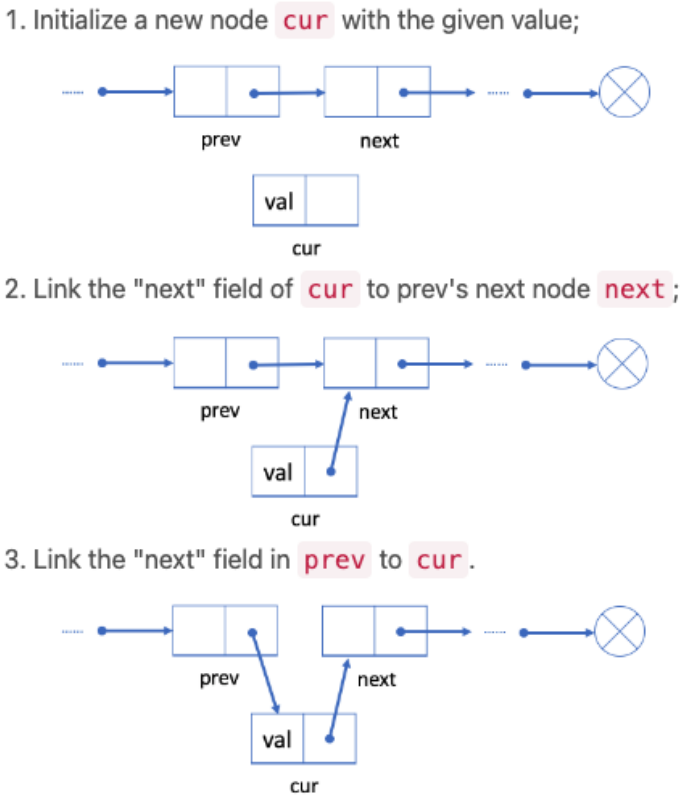
In a singly linked list, each node in a singly-linked list contains not only the value but also a reference field to link to the next node.

3.1.1 Insert

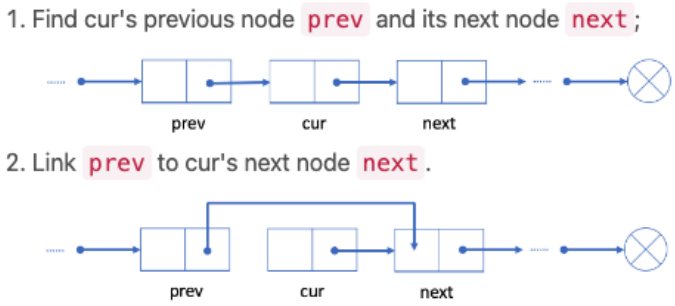
Figure 3.3 shows the operations to add a new value after a given node **prev**.

3.1.2 Delete

Figure 3.4 shows the operations to delete an existing node **cur** from the singly linked list.



Figures 3.3: Insert into singled linked list



Figures 3.4: Delete from singled linked list

3.2 Doubly Linked List

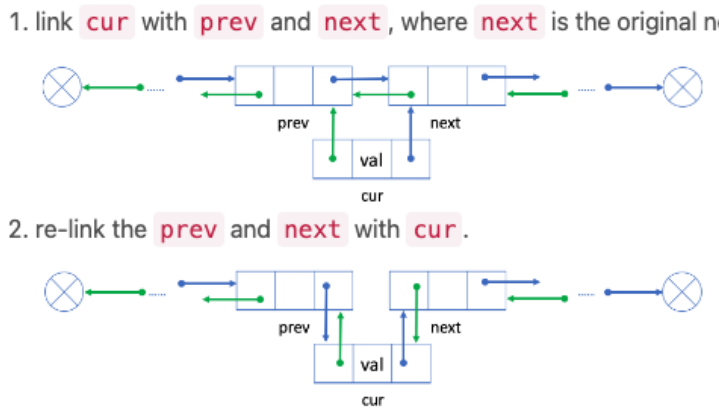
The doubly linked list works in a similar way but has one more reference field which is known as the "prev" field. With this extra field, you are able to know the previous node of the current node.

3.2.1 Add Operation

If we want to insert a new node **cur** after an existing node **prev**, we can divide this process into two steps as shown in Figure 3.5.

3.2.2 Delete Operation

If we want to delete an existing node **cur** from the doubly linked list, we can simply link its previous node **prev** with its next node **next**.



Figures 3.5: Add to linked list

3.3 Comparison

Here we provide a comparison of time complexity between the linked list and the array shown in Figure 3.6.

		Array	Singly-Linked List	Doubly-Linked List
Access	by index	O(1)	O(N)	O(N)
Add	before first node	O(N)	O(1)	O(1)
	after given node	O(N)	O(1)	O(1)
	after last node	O(1)	O(N)	O(1)
Delete	the first node	O(N)	O(1)	O(1)
	a given node	O(N)	O(N)	O(1)
	the last node	O(1)	O(N)	O(1)
Search	a given node	O(N)	O(N)	O(N)

Figures 3.6: Array and Linked List

3.4 Examples

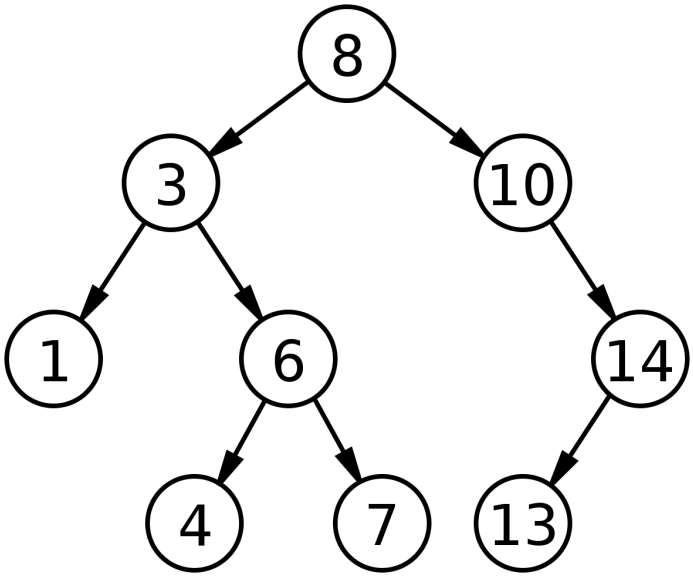
Here are some codes about linked list on Github.

Chapter 4

Binary Tree

A **tree** is a frequently-used data structure to simulate a hierarchical tree structure. Each node of the tree will have a **root** value and a list of references to other nodes which are called **child** nodes.

A **Binary Tree** is one of the most typical tree structure. As the name suggests, a binary tree is a tree data structure in which each node has at most two children, which are referred to as the **left** child and the **right** child, as shown in Figure 4.1.



Figures 4.1: Binary tree

4.1 Traverse a Tree

There are 4 common used traversal:

- ♠ Pre-order Traversal
- ♠ In-order Traversal
- ♠ Post-order Traversal

♠ Level-order Traversal

The pre-in-post order references the **root**.

Pre-order traversal is to visit the **root** first. Then traverse the left subtree. Finally, traverse the right subtree.

In-order traversal is to traverse the left subtree first. Then visit the **root**. Finally, traverse the right subtree.

Post-order traversal is to traverse the left subtree first. Then traverse the right subtree. Finally, visit the **root**.

Level-order traversal is to traverse the nodes by level from root to leaves. In the same level, traverse from left to right.

Take the Figure 4.1 for example. Pre-order traversal generate the sequence: [8 3 1 6 4 7 10 14 13]. In-order traversal generate the sequence: [1 3 4 6 7 8 10 13 14]. Post-order traversal generate the sequence: [1 4 7 6 3 13 14 10 8]. Level-order traversal generate the sequence: [8 3 10 1 6 14 4 7 13].

4.2 Examples

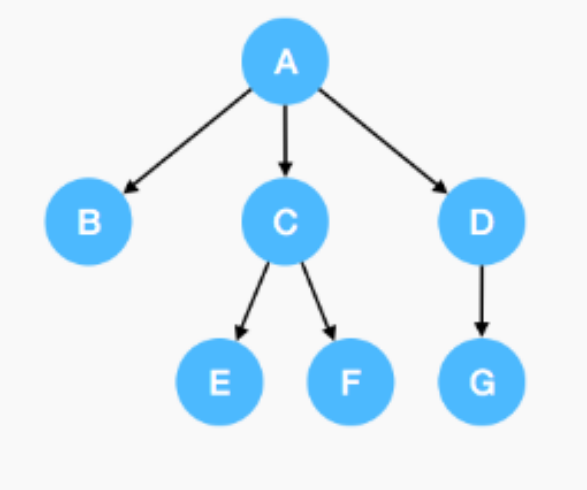
Here are some codes about binary tree on Github.

Chapter 5

N-ary Tree

If a tree is a rooted tree in which each node has no more than `N` children, it is called **N-ary tree**.

Here is an example of 3-ary tree:



Figures 5.1: N-ary tree

Trie (Chapter 6) is one of the most frequently used N-ary trees.

5.1 Travesal

A binary tree can be traversed in preorder, inorder, postorder or level-order. Among these traversal methods, preorder, postorder and level-order traversal are suitable to be extended to an **N-ary tree**.

5.2 Examples

Here are some examples on Github.

Chapter 6

Trie

A Trie is a special form of a N-ary tree. Typically, a trie is used to store strings. Each Trie node represents a string (a prefix). Each node might have several children nodes while the paths to different children nodes represent different characters. And the strings the child nodes represent will be the origin string represented by the node itself plus the character on the path.

One important property of Trie is that all the descendants of a node have a common prefix of the string associated with that node. That's why Trie is also called prefix tree.

6.1 How to represent a Trie?

There are a lot of different representations of a trie node. Here we provide two of them.

6.1.1 Array

The first solution is to use an array to store children nodes.

For instance, if we store strings which only contains letter a to z, we can declare an array whose size is 26 in each node to store its children nodes. And for a specific character c, we can use c - 'a' as the index to find the corresponding child node in the array. Here's the Python code.

```
class TrieNode:
    def __init__(self, N=26):
        # you might need some extra values according to different cases
        self.N = N
        self.children = [''] * self.N

# Usesage.
# Initialization
root = TrieNode()
# Return a specific child node with char c
root[ord[c] - ord['a']]
```

It is really fast to visit a child node. It is comparatively easy to visit a specific child since we can easily transfer a character to an index in most cases. But not all children nodes are needed. So there

might be some waste of space.

6.1.2 Hashmap

The second solution is to use a hashmap to store children nodes. Here's the Python code.

```
class TrieNode:
    def __init__(self):
        self.children = {}

# Usesage.
# Initialization
root = TrieNode()
# Return a specific child node with char c
root[c]
```

It is even easier to visit a specific child directly by the corresponding character. But it might be a little slower than using an array. However, it saves some space since we only store the children nodes we need. It is also more flexible because we are not limited by a fixed length and fixed range.

6.2 Insertion in Trie

Here is the pseudo-code:

```
1. Initialize: cur = root
2. for each char c in target string S:
3.   if cur does not have a child c:
4.     cur.children[c] = new Trie node
5.     cur = cur.children[c]
6. cur is the node which represents the string S
```

6.3 Search in Trie

6.3.1 Search prefix

Here is the pseudo-code:

```
1. Initialize: cur = root
2. for each char c in target string S:
3.   if cur does not have a child c:
4.     search fails
5.   cur = cur.children[c]
6. search successes
```

6.3.2 Search word

You might also want to know how to search for a specific word rather than a prefix. We can treat this word as a prefix and search in the same way we mentioned above.

1 If search fails which means that no words start with the target word, the target word is definitely not in the Trie.

2 If search succeeds, we need to check if the target word is only a prefix of words in Trie or it is exactly a word.

6.4 Examples

Here are some codes about hash table on Github

Chapter 7

Queue and Stack

7.1 Queue

Queue is first in first out data structure. In a FIFO data structure, the first element added to the queue will be processed first. It is like an array, but the elements can be put into the queue at the end ant put out of the queue at the front. As show in Figure 7.1.

Figures 7.1: Queue

There are two operations:

- ♠ enqueue: insert a new element at the end of the queue
- ♠ dequeue: remove the element at the front(head) of the queue

7.1.1 Queue and BFS

Queue is usually used in BFS (Breadth-first Search).

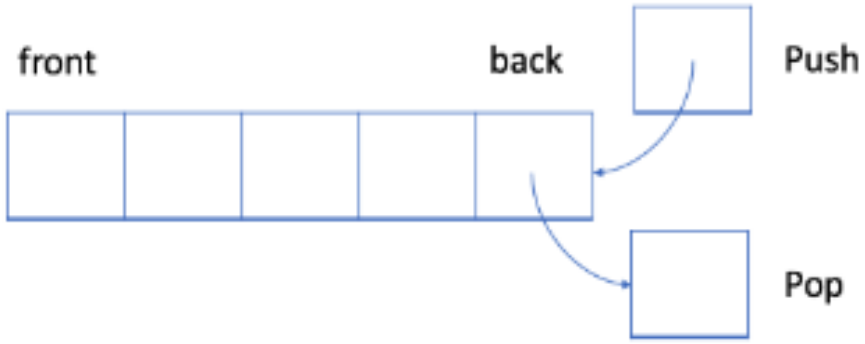
Here is a simple template (Python):

```
1  def BFS(root, target) -> int:
2    queue = []
3    step = 0
4    queue.append(root)
```

```
    while queue:
        for node in queue:
            if node == target:
                return step
            # Add all node neighbors into the queue
            for neighbor in node.neighbors:
                queue.append(neighbor)
            queue.pop(0)
        step += 1
    # There is no path from root to target.
    return -1
```

7.2 Stack

Stack is last in first out data structure. In a LIFO structure, the newest element added to the queue will be processed first. It is like an array, but the elements can be put into the stack and put out of the queue at the end. As show in Figure 7.2



Figures 7.2: Stack

There are two operations:

- ♠ push: add a new element at the end(top) of the stack
- ♠ pop: remove the element at the end(top) of the stack

7.2.1 Stack and DFS

In most cases, we can also use DFS when using BFS. But there is an important difference: the traversal order. Different from BFS, the nodes you visit earlier might not be the nodes which are closer to the root node. As a result, the first path you found in DFS might not be the shortest path.

Here is a simple template (Python):

```
def DFS(cur, target, visited: set) -> bool:
    if cur == target:
        return True
    for nei in cur.neighbors:
        if nei not in visited:
```

```
            visited.add(nei)
        if DFS(nei, target, visited):
            return True
    return False
```

The advantage of the recursion solution is that it is easier to implement. However, there is a huge disadvantage: if the depth of recursion is too high, you will suffer from stack overflow. In that case, you might want to use BFS instead or implement DFS using an explicit stack.

7.3 Examples

Here are some example on Github.

Chapter 8

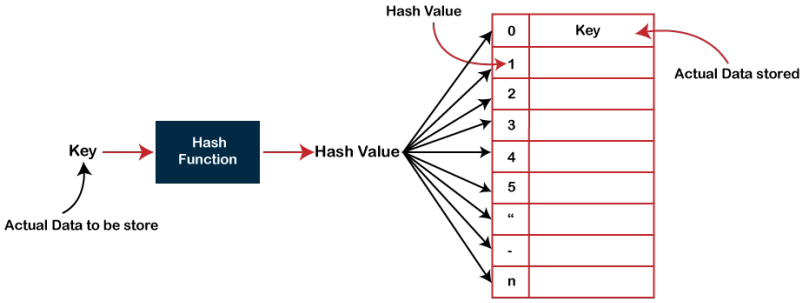
Hash Table

Hash Table is a data structure which organizes data using hash functions in order to support quick insertion and search.

The key idea of Hash Table is to use a hash function to map keys to buckets. To be more specific,

- When we insert a new key, the hash function will decide which bucket the key should be assigned and the key will be stored in the corresponding bucket;
- When we want to search for a key, the hash table will use the **same** hash function to find the corresponding bucket and search only in the specific bucket.

For example in Figure:



Figures 8.1: Hash table

8.1 Designing a Hash Table

There are two essential factors that you should pay attention to when you are going to design a hash table: hash function and collision resolution.

8.1.1 Hash Function

The hash function is the most important component of a hash table which is used to map the key to a specific bucket. The hash function will depend on **the range of key values** and **the number of buckets**. It is an open problem to design a hash function. The idea is to try to assign the key to the

bucket as **uniformly** as you can. Ideally, a perfect hash function will be a one-one mapping between the key and the bucket. However, in most cases, a hash function is not perfect and it is a tradeoff between the amount of buckets and the capacity of a bucket.

8.1.2 Collision Resolution

Ideally, if our hash function is a perfect one-one mapping, we will not need to handle collisions. Unfortunately, in most cases, collisions are almost inevitable.

A collision resolution algorithm should solve the following questions:

1. How to organize the values in the same bucket?
2. What if too many values are assigned to the same bucket?
3. How to search for a target value in a specific bucket?

These questions are related to the capacity of the bucket and the number of keys which might be mapped into the same bucket according to our hash function. Let's assume that the bucket, which holds the maximum number of keys, has N keys. Typically, if N is constant and small, we can simply use an array to store keys in the same bucket. If N is variable or large, we might need to use height-balanced binary search tree instead.

8.2 Built-in Hash Table

The typical design of built-in hash table is:

1. The key can be any **hashable** type. And a key with belongs to a hashable type will have a **hashcode**. This code will be used in the mapping function to get the bucket index.
2. Each bucket contains **an array** to store all the values in the same bucket initially.
3. If there are two many values the same bucket, these values will be maintained in a **height-balanced binary search tree** instead.

The average time complexity of both insertion and search is still **O(1)**. And the time complexity in the worst case is **O(logN)** for both insertion and search by using height-balanced BST.

8.3 Designing a Key Example

Sometimes you have to think it over to design a suitable key when using a hash table.

For example:

Given an array of strings, group anagrams[1] together.

As we know, a hash map can perform really well in grouping information by key. But we cannot use the original string as key directly. We have to design a proper key to present the type of anagrams. For instance, "eat" and "ate" should be in the same group. While "eat" and "act" should not be grouped together.

When you design a key, you need to guarantee that:

[1] An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

1. All values belong to the same group will be mapped in the same group.
2. Values which needed to be separated into different groups will not be mapped into the same group.

```python
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        d = collections.defaultdict(list)
        for s in strs:
            hashkey = self.hash(s)
            d[hashkey].append(s)
        return list(d.values())

    def hash(self, s: str) -> str:
        return ''.join(sorted(s))
```

This process is similar to design a hash function, but here is an essential difference. A hash function satisfies rule 1 but might not satisfy rule 2. But your mapping function should satisfy both of them.

8.3.1 Summary

Here are some takeaways about how to design the key for you:

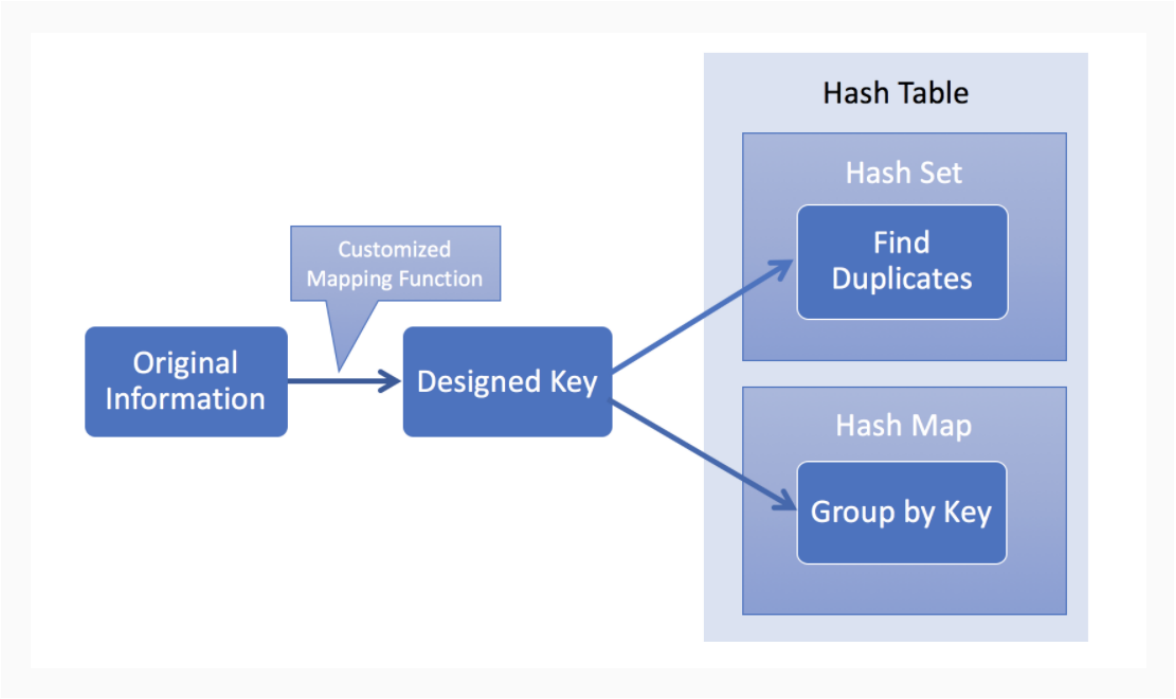
1. When the order of each element in the string/array doesn't matter, you can use the **sorted string/array** as the key.
2. If you only care about the offset of each value, usually the offset from the first value, you can use the **offset** as the key.
3. In a tree, you might want to directly use the **TreeNode** as key sometimes. But in most cases, the **serialization of the subtree** might be a better idea.
4. In a matrix, you might want to use **the row index** or **the column index** as key.
5. In a Sudoku, you can combine the row index and the column index to identify which **block** this element blongs to.
6. Sometimes, in a matrix, you might want to aggregate the values in the same **diagonal line**. $(i, j) \to i + j$, $(i, j) \to i - j$

8.4 Conclusion

A typical thinking process to solve problems by hash table is show in Figure 8.2:

What's more, we will meet more complicated problems sometimes. We might need to:

- ♠ use several hash tables together
- ♠ combine the hash table with other data structure
- ♠ combine the hash table with other algorithms
- ♠ ...



Figures 8.2: Thinking process by hash table

8.5 Examples

Here are some examples on Github

Part III

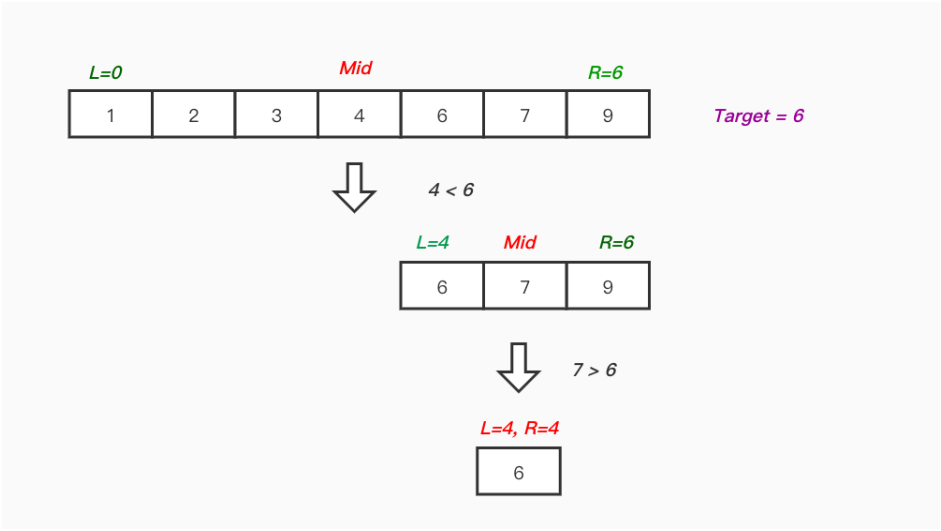
Algorithm

Chapter 9

Binary Search

Binary Search is one of the most fundamental and useful algorithms in Computer Science. It describes the process of searching for a specific value in an **ordered** collection.

9.1 General Idea



Figures 9.1: Binary Search

The general process is as follows (if the collection is increase sorted):

1. Compute the `mid` index.
2. If the value at `mid` is the `target`, search ends.
3. If the `target` is great than the value at `mid`, search the right part of the `mid` else the left.
4. Repeat the previous process till the `target` is found or search end (no more elements to search for).

9.2 Templates

9.2.1 Template 1

```python
def binarySearch(nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: int
    """
    if len(nums) == 0:
        return -1

    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    # End Condition: left > right
    return -1
```

Template 1 is the most basic and elementary form of Binary Search. It is used to search for an element or condition which can be determined by accessing **a single index** in the array.

9.2.2 Template 2

```python
def binarySearch(nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: int
    """
    if len(nums) == 0:
        return -1

    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid

    # Post-processing:
    # End Condition: left == right
```

```
    if left != len(nums) and nums[left] == target:
        return left
    return -1
```

Template 2 is an advanced form of Binary Search. It is used to search for an element or condition which requires accessing **the current index and its immediate right neighbor's index** in the array.

9.2.3 Template 3

```
def binarySearch(nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: int
    """
    if len(nums) == 0:
        return -1

    left, right = 0, len(nums) - 1
    while left + 1 < right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid
        else:
            right = mid

    # Post-processing:
    # End Condition: left + 1 == right
    if nums[left] == target: return left
    if nums[right] == target: return right
    return -1
```

Template 3 is another form of Binary Search. It is used to search for an element or condition which requires accessing **the current index and its immediate left and right neighbor's index** in the array.

9.3 Time and Space Complexity

9.3.1 Time Complexity

Logarithmic Time: $O(\log n)$

Because Binary Search operates by applying a condition to the value in the middle of our search space and thus cutting the search space in half, in the worse case, we will have to make $O(\log n)$ comparisons, where n is the number of elements in our collection.

9.3.2 Space Complexity

Constant Space: $O(1)$

Although Binary Search does require keeping track of 3 indices, the iterative solution does not typically require any other additional space and can be applied directly to the collection itself, therefore warrants $O(1)$ or constant space.

9.4 Examples

Here are some examples on Github.

Chapter 10

Sorting

In computer science, we have formal definitions of sorting with respect to ordering relations.

An ordering relation has two key properties:

1. Given two elements a and b, exactly one of the following must be true: $a < b, a = b$ or $a > b$ (law of trichotomy)
2. If $a < b$ and $b < c$ then $a < c$ (law of transitivity)

A **sort** is formally defined as a rearrangement of a sequence of elements that puts all elements into a non-decreasing order based on the ordering relation.

An important concept in sorting is **inversions**. An inversion in a sequence is defined as a pair of elements that are out of order with respect to the ordering relation. For example, $1, 2, 4, 6, 5, 3$. The inversions are $(4, 3), (6, 5), (6, 3), (5, 3)$. There are 4 inversions in the previous list.

The next important concept in sorting is the **stability** of sorting algorithms. The key feature of a stable sorting algorithm is that it will preserve the order of equal elements. For example, [“hello”, “world”, “we”, “are”, “learning, “sorting”]. We define an ordering relation based on the length of the string. There are two valid sorts for this sequence:

1. “we”, “are”, “hello”, “world”, “sorting”, “learning”
2. “we”, “are”, “world”, “hello”, “sorting”, “learning”

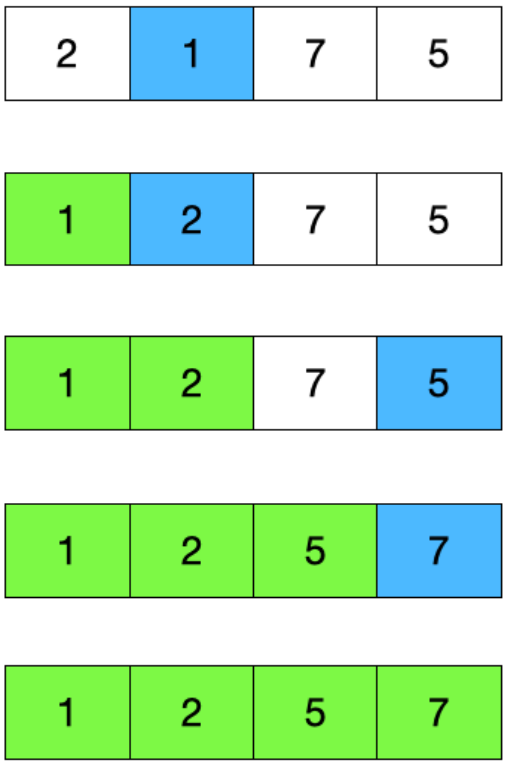
We consider (1) to be a stable sort since the equal elements “hello” and “world” are kept in the same relative order as the original sequence.

10.1 Comparison Based Sort

Comparison based sorts are sorting algorithms that require a direct method of comparison defined by the ordering relation. In a sense, they are the most natural sorting algorithms since, intuitively, when we think about sorting elements, we instinctively think about comparing elements to each other.

10.1.1 Selection Sort

Suppose we had a collection of elements where every element is an integer. Selection sort will build up the sorted list by repeatedly **selecting** the minimum element in that list and moving it to the front of the list through a swap. It will proceed to swap elements appropriately until the entire list is sorted.



Figures 10.1: Selection sort

It's time complexity is $O(n^2)$ and space complexity is $O(1)$. It is not a stable sorting algorithm.

```python
class Solution:
    def selection_sort(self, lst: List[int]) -> None:
        """
        Mutates lst so that it is sorted via selecting the minimum element
            and swapping it with the corresponding index.
        """
        for i in range(len(lst)):
            min_index = i
            for j in range(i + 1, len(lst)):
                # Update minimum index
                if lst[j] < lst[min_index]:
                    min_index = j

            # Swap current index with minimum element in rest of list
            lst[min_index], lst[i] = lst[i], lst[min_index]
```

10.1.2 Bubble Sort

Suppose we have a collection of integers that we want to sort in ascending order. Bubble sort proceeds to consider two adjacent elements at a time. If these two adjacent elements are out of order (in this case, the left element is strictly greater than the right element), bubble sort will swap them. It then proceeds to the next pair of adjacent elements. In the first pass of bubble sort, it will process every set of adjacent elements in the collection once, making swaps as necessary. The core idea of bubble sort is it will repeat this process until no more swaps are made in a single pass, which means the list is sorted.

Bubble sort runtime of $O(n^2)$. The space complexity of bubble sort is $O(1)$. Bubble sort is also a stable sorting algorithm since equal elements will never have swapped places, so their relative ordering will be preserved.

Overall, bubble sort is fairly simple to implement, and it's stable, but outside of that, this algorithm does not have many desirable features. It's fairly slow for most inputs and, as a result, it is rarely used in practice.

```
class Solution:
    def bubble_sort(self, lst: List[int]) -> None:
        """
        Mutates lst so that it is sorted via swapping adjacent elements until
        the entire lst is sorted.
        """
        has_swapped = True
        # if no swap occurred, lst is sorted
        while has_swapped:
            has_swapped = False
            for i in range(len(lst) - 1):
                if lst[i] > lst[i + 1]:
                    # Swap adjacent elements
                    lst[i], lst[i + 1] = lst[i + 1], lst[i]
                    has_swapped = True
```

Bibliography

[1] Aston Zhang et al. *Dive into Deep Learning*. 2021.

[2] Debra Cameron et al. *Learning GNU Emacs*. O'Reilly Media, Inc., 2004.

[3] Ian Goodfellow et al. *Deep Learning*. 2016.

[4] Yann LeCun et al. Gradientbased learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[5] D.E. Knuth. *The TEXbook*. Addison Wesley, 1986.

[6] D.E. Knuth. Typesetting concrete mathematics. *TUGboat*, 10(1):31–36, April 1989.

[7] Adrian Rosebrock. *Practical Python and OpenCV*. PyImageSearch, 2016.

[8] Adrian Rosebrock. *Deep Learning for Computer Vision with Python*. PyImageSearch, 2017.