Algorithm

http://algorithm.yuanbin.me

Table of Contents

- Preface 1.1
- FAQ 1.2
  - Guidelines for Contributing 1.2.1
  - Contributors 1.2.2
- Part I - Basics 1.3
- Basics Data Structure 1.4
  - String 1.4.1
  - Linked List 1.4.2
  - Binary Tree 1.4.3
  - Huffman Compression 1.4.4
  - Queue 1.4.5
  - Heap 1.4.6
  - Stack 1.4.7
  - Set 1.4.8
  - Map 1.4.9
  - Graph 1.4.10
- Basics Sorting 1.5
  - Bubble Sort 1.5.1
  - Selection Sort 1.5.2
  - Insertion Sort 1.5.3
  - Merge Sort 1.5.4
  - Quick Sort 1.5.5
  - Heap Sort 1.5.6
  - Bucket Sort 1.5.7
  - Counting Sort 1.5.8
  - Radix Sort 1.5.9
- Basics Algorithm 1.6
  - Divide and Conquer 1.6.1
  - Binary Search 1.6.2
  - Math 1.6.3
    - Greatest Common Divisor 1.6.3.1
    - Prime 1.6.3.2
  - Knapsack 1.6.4
  - Counting Problem 1.6.5
  - Probability 1.6.6
    - Shuffle 1.6.6.1
  - Bitmap 1.6.7

Basics Misc 1.7
- Bit Manipulation 1.7.1

Part II - Coding 1.8

String 1.9
- strStr 1.9.1
- Two Strings Are Anagrams 1.9.2
- Compare Strings 1.9.3
- Anagrams 1.9.4
- Longest Common Substring 1.9.5
- Rotate String 1.9.6
- Reverse Words in a String 1.9.7
- Valid Palindrome 1.9.8
- Longest Palindromic Substring 1.9.9
- Space Replacement 1.9.10
- Wildcard Matching 1.9.11
- Length of Last Word 1.9.12
- Count and Say 1.9.13

Integer Array 1.10
- Remove Element 1.10.1
- Zero Sum Subarray 1.10.2
- Subarray Sum K 1.10.3
- Subarray Sum Closest 1.10.4
- Recover Rotated Sorted Array 1.10.5
- Product of Array Exclude Itself 1.10.6
- Partition Array 1.10.7
- First Missing Positive 1.10.8
- 2 Sum 1.10.9
- 3 Sum 1.10.10
- 3 Sum Closest 1.10.11
- Remove Duplicates from Sorted Array 1.10.12
- Remove Duplicates from Sorted Array II 1.10.13
- Merge Sorted Array 1.10.14
- Merge Sorted Array II 1.10.15
- Median 1.10.16
- Partition Array by Odd and Even 1.10.17
- Kth Largest Element 1.10.18

Binary Search 1.11
- Binary Search 1.11.1
- Search Insert Position 1.11.2
- Search for a Range 1.11.3

- First Bad Version 1.11.4
- Search a 2D Matrix 1.11.5
- Search a 2D Matrix II 1.11.6
- Find Peak Element 1.11.7
- Search in Rotated Sorted Array 1.11.8
- Search in Rotated Sorted Array II 1.11.9
- Find Minimum in Rotated Sorted Array 1.11.10
- Find Minimum in Rotated Sorted Array II 1.11.11
- Median of two Sorted Arrays 1.11.12
- Sqrt x 1.11.13
- Wood Cut 1.11.14

Math and Bit Manipulation 1.12

- Single Number 1.12.1
- Single Number II 1.12.2
- Single Number III 1.12.3
- O1 Check Power of 2 1.12.4
- Convert Integer A to Integer B 1.12.5
- Factorial Trailing Zeroes 1.12.6
- Unique Binary Search Trees 1.12.7
- Update Bits 1.12.8
- Fast Power 1.12.9
- Hash Function 1.12.10
- Count 1 in Binary 1.12.11
- Fibonacci 1.12.12
- A plus B Problem 1.12.13
- Print Numbers by Recursion 1.12.14
- Majority Number 1.12.15
- Majority Number II 1.12.16
- Majority Number III 1.12.17
- Digit Counts 1.12.18
- Ugly Number 1.12.19
- Plus One 1.12.20

Linked List 1.13

- Remove Duplicates from Sorted List 1.13.1
- Remove Duplicates from Sorted List II 1.13.2
- Remove Duplicates from Unsorted List 1.13.3
- Partition List 1.13.4
- Add Two Numbers 1.13.5
- Two Lists Sum Advanced 1.13.6
- Remove Nth Node From End of List 1.13.7

- Linked List Cycle 1.13.8
- Linked List Cycle II 1.13.9
- Reverse Linked List 1.13.10
- Reverse Linked List II 1.13.11
- Merge Two Sorted Lists 1.13.12
- Merge k Sorted Lists 1.13.13
- Reorder List 1.13.14
- Copy List with Random Pointer 1.13.15
- Sort List 1.13.16
- Insertion Sort List 1.13.17
- Palindrome Linked List 1.13.18
- Delete Node in the Middle of Singly Linked List 1.13.19
- LRU Cache 1.13.20
- Rotate List 1.13.21
- Swap Nodes in Pairs 1.13.22
- Remove Linked List Elements 1.13.23

Binary Tree 1.14

- Binary Tree Preorder Traversal 1.14.1
- Binary Tree Inorder Traversal 1.14.2
- Binary Tree Postorder Traversal 1.14.3
- Binary Tree Level Order Traversal 1.14.4
- Binary Tree Level Order Traversal II 1.14.5
- Maximum Depth of Binary Tree 1.14.6
- Balanced Binary Tree 1.14.7
- Binary Tree Maximum Path Sum 1.14.8
- Lowest Common Ancestor 1.14.9
- Invert Binary Tree 1.14.10
- Diameter of a Binary Tree 1.14.11
- Construct Binary Tree from Preorder and Inorder Traversal 1.14.12
- Construct Binary Tree from Inorder and Postorder Traversal 1.14.13
- Subtree 1.14.14
- Binary Tree Zigzag Level Order Traversal 1.14.15
- Binary Tree Serialization 1.14.16

Binary Search Tree 1.15

- Insert Node in a Binary Search Tree 1.15.1
- Validate Binary Search Tree 1.15.2
- Search Range in Binary Search Tree 1.15.3
- Convert Sorted Array to Binary Search Tree 1.15.4
- Convert Sorted List to Binary Search Tree 1.15.5
- Binary Search Tree Iterator 1.15.6

- Exhaustive Search 1.16
  - Subsets 1.16.1
  - Unique Subsets 1.16.2
  - Permutations 1.16.3
  - Unique Permutations 1.16.4
  - Next Permutation 1.16.5
  - Previous Permuation 1.16.6
  - Permutation Index 1.16.7
  - Permutation Index II 1.16.8
  - Permutation Sequence 1.16.9
  - Unique Binary Search Trees II 1.16.10
  - Palindrome Partitioning 1.16.11
  - Combinations 1.16.12
  - Combination Sum 1.16.13
  - Combination Sum II 1.16.14
  - Minimum Depth of Binary Tree 1.16.15
  - Word Search 1.16.16
- Dynamic Programming 1.17
  - Triangle 1.17.1
  - Backpack 1.17.2
  - Backpack II 1.17.3
  - Minimum Path Sum 1.17.4
  - Unique Paths 1.17.5
  - Unique Paths II 1.17.6
  - Climbing Stairs 1.17.7
  - Jump Game 1.17.8
  - Word Break 1.17.9
  - Longest Increasing Subsequence 1.17.10
  - Palindrome Partitioning II 1.17.11
  - Longest Common Subsequence 1.17.12
  - Edit Distance 1.17.13
  - Jump Game II 1.17.14
  - Best Time to Buy and Sell Stock 1.17.15
  - Best Time to Buy and Sell Stock II 1.17.16
  - Best Time to Buy and Sell Stock III 1.17.17
  - Best Time to Buy and Sell Stock IV 1.17.18
  - Distinct Subsequences 1.17.19
  - Interleaving String 1.17.20
  - Maximum Subarray 1.17.21
  - Maximum Subarray II 1.17.22

- Longest Increasing Continuous subsequence 1.17.23
- Longest Increasing Continuous subsequence II 1.17.24
- Egg Dropping Puzzle 1.17.25
- Maximal Square 1.17.26

Graph 1.18

- Find the Connected Component in the Undirected Graph 1.18.1
- Route Between Two Nodes in Graph 1.18.2
- Topological Sorting 1.18.3
- Word Ladder 1.18.4
- Bipartial Graph Part I 1.18.5

Data Structure 1.19

- Implement Queue by Two Stacks 1.19.1
- Min Stack 1.19.2
- Sliding Window Maximum 1.19.3
- Longest Words 1.19.4
- Heapify 1.19.5
- Kth Smallest Number in Sorted Matrix 1.19.6

Problem Misc 1.20

- Nuts and Bolts Problem 1.20.1
- String to Integer 1.20.2
- Insert Interval 1.20.3
- Merge Intervals 1.20.4
- Minimum Subarray 1.20.5
- Matrix Zigzag Traversal 1.20.6
- Valid Sudoku 1.20.7
- Add Binary 1.20.8
- Reverse Integer 1.20.9
- Gray Code 1.20.10
- Find the Missing Number 1.20.11
- N Queens 1.20.12
- N Queens II 1.20.13
- Minimum Window Substring 1.20.14
- Continuous Subarray Sum 1.20.15
- Continuous Subarray Sum II 1.20.16
- Longest Consecutive Sequence 1.20.17

Part III - Contest 1.21

Google APAC 1.22

- APAC 2015 Round B 1.22.1
  - Problem A. Password Attacker 1.22.1.1
- APAC 2016 Round D 1.22.2

Problem A. Dynamic Grid 1.22.2.1
Microsoft 1.23
Microsoft 2015 April 1.23.1
Problem A. Magic Box 1.23.1.1
Problem B. Professor Q's Software 1.23.1.2
Problem C. Islands Travel 1.23.1.3
Problem D. Recruitment 1.23.1.4
Microsoft 2015 April 2 1.23.2
Problem A. Lucky Substrings 1.23.2.1
Problem B. Numeric Keypad 1.23.2.2
Problem C. Spring Outing 1.23.2.3
Microsoft 2015 September 2 1.23.3
Problem A. Farthest Point 1.23.3.1
Appendix I Interview and Resume 1.24
Interview 1.24.1
Resume 1.24.2
Appendix II System Design 1.25
The System Design Process 1.25.1
Statistics 1.25.2
System Architecture 1.25.3
Scalability 1.25.4
Tags 1.26

Data Structure and Algorithm/leetcode/lintcode

- English via Data Structure and Algorithm notes
- /leetcode/lintcode
- /leetcode/lintcode

Introduction

This work is some notes of learning and practicing data structures and algorithm.

1. Part I is some brief introduction of basic data structures and algorithm, such as, linked lists, stack, queues, trees, sorting and etc.
2. Part II is the analysis and summary of programming problems, and most of the programming problems come from https://leetcode.com/, http://www.lintcode.com/, http://www.geeksforgeeks.org/, http://hihocoder.com/, https://www.topcoder.com/.
3. Part III is the appendix of resume and other supplements.

This project is hosted on https://github.com/billryan/algorithm-exercise and rendered by Gitbook. You can star the repository on the GitHub to keep track of updates. Another choice is to subscribe channel `#github_commit` via Slack https://ds-algo.slack.com/messages/github_commit/. ~~RSS feed is under development.~~

Feel free to access http://slackin4ds-algo.herokuapp.com for Slack invite automation.

You can view/search this document online or offline, feel free to read it. :)

- Online(Rendered by Gitbook): http://algorithm.yuanbin.me
- Offline(Compiled by Gitbook and Travis-CI):
  1. EPUB: GitHub, Gitbook, CDN() - Recommended for iPhone/iPad/MAC
  2. PDF: GitHub, Gitbook, CDN() - Recommended for Desktop
  3. MOBI: GitHub, Gitbook, CDN() - Recommended for Kindle
- Site Search via Google: `keywords site:algorithm.yuanbin.me`
- Site Search via Swiftype: Click `Search this site` on the right bottom of webpages

License

Contribution

- English is maintained by @billryan
- is maintained by @billryan, @Shaunwei
- is maintained by @CrossLuna

Other contributors can be found in Contributors to algorithm-exercise

Donation

:)

@billryan ~ `5307`

@billryan

yuanbin2014(at)gmail.com

Wechat

PayPal

yuanbin2014(at)gmail.com friends and family

- `taoli***@gmail.com`, 20

- `*` , 6.66
- `wen***@126.com` , 20.16
- `she***@163.com` , 10
- `*` , 20
- `*` , 50
- `*` , 20
- `don***@163.com` , 5
- `129***@qq.com` , 50
- `130****9675` , 5
- `Tong W***` , 20 $
- `ee.***@gmail.com` , 6.66

CDN / Contributors ///

To Do

- [ ] add multiple languages support, currently , are available
- [x] explore nice writing style
- [x] add implementations of `Python` , `C++` , `Java` code
- [x] add time and space complexity analysis
- [x] summary of basic data structure and algorithm
- [x] add CSS for online website http://algorithm.yuanbin.me
- [x] add proper Chinese fonts for PDF output

FAQ - Frequently Asked Question

Some guidelines for contributing and other questions are listed here.

How to Contribute?

- Access Guidelines for Contributing for details.

Guidelines for Contributing

- Access English via Guidelines for Contributing
-
-

Part I - Basics

The first part summarizes some of the main aspects of data structures and algorithms, such as implementation and usage.

This chapter consists of the following sections.

Reference

- VisuAlgo - Animated visualizations of data structures and algorithms
- Data Structure Visualizations - An alternative to VisuAlgo
- Sorting Algorithms - Animations comparing various sorting algorithms

Data Structure

This chapter describes the fundamental data structures and their implementations.

String

String-related problems often appear in interview questions. In actual development, strings are also frequently used. Summarized here are common uses of strings in C++, Java, and Python.

Python

```
s1 = str()
# in python, `''` and `""` are the same
s2 = "shaunwei" # 'shaunwei'
s2len = len(s2)
# last 3 chars
s2[-3:] # wei
s2[5:8] # wei
s3 = s2[:5] # shaun
s3 += 'wei' # return 'shaunwei'
# list in python is same as ArrayList in java
s2list = list(s3)
# string at index 4
s2[4] # 'n'
# find index at first
s2.index('w')  # return 5, if not found, throw ValueError
s2.find('w') # return 5, if not found, return -1
```

In Python, there's no StringBuffer or StringBuilder. However, string manipulations are fairly efficient already.

Java

```
String s1 = new String();
String s2 = "billryan";
int s2Len = s2.length();
s2.substring(4, 8); // return "ryan"
StringBuilder s3 = new StringBuilder(s2.substring(4, 8));
s3.append("bill");
String s2New = s3.toString(); // return "ryanbill"
// convert String to char array
char[] s2Char = s2.toCharArray();
// char at index 4
char ch = s2.charAt(4); // return 'r'
// find index at first
int index = s2.indexOf('r'); // return 4. if not found, return -1
```

The difference between StringBuffer and StringBuilder is that the former guarantees thread safety. In a single-threaded environment, StringBuilder is more efficient.

String

String related topics are discussed in this chapter.

> In order to re-use most of the memory of an existing data structure, internal implementation of string is immutable in most programming languages(Java, Python). Take care if you want to modify character in place.

strStr

Question

- leetcode: Implement strStr() | LeetCode OJ
- lintcode: lintcode - (13) strstr

Problem Statement

For a given source string and a target string, you should output the **first** index(from 0) of target string in source string.

If target does not exist in source, just return `-1`.

Example

If source = `"source"` and target = `"target"`, return `-1`.

If source = `"abcdabcdefg"` and target = `"bcd"`, return `1`.

Challenge

O(n2) is acceptable. Can you implement an O(n) algorithm? (hint: *KMP*)

Clarification

Do I need to implement KMP Algorithm in a real interview?

- Not necessary. When you meet this problem in a real interview, the interviewer may just want to test your basic implementation ability. But make sure your confirm with the interviewer first.

Problem Analysis

It's very straightforward to solve string match problem with nested for loops. Since we must iterate the target string, we can optimize the iteration of source string. It's unnecessary to iterate the source string if the length of remaining part does not exceed the length of target string. We can only iterate the valid part of source string. Apart from this naive algorithm, you can use a more effective algorithm such as KMP.

Python

```
class Solution:
    def strStr(self, source, target):
        if source is None or target is None:
            return -1

        for i in range(len(source) - len(target) + 1):
            for j in range(len(target)):
                if source[i + j] != target[j]:
                    break
            else:  # no break
                return i
        return -1
```

C

```c
int strStr(char* haystack, char* needle) {
    if (haystack == NULL || needle == NULL) return -1;

    const int len_h = strlen(haystack);
    const int len_n = strlen(needle);
    for (int i = 0; i < len_h - len_n + 1; i++) {
        int j = 0;
        for (; j < len_n; j++) {
            if (haystack[i+j] != needle[j]) {
                break;
            }
        }
        if (j == len_n) return i;
    }

    return -1;
}
```

C++

```cpp
class Solution {
public:
    int strStr(string haystack, string needle) {
        if (haystack.empty() && needle.empty()) return 0;
        if (haystack.empty()) return -1;
        if (needle.empty()) return 0;
        // in case of overflow for negative
        if (haystack.size() < needle.size()) return -1;

        for (int i = 0; i < haystack.size() - needle.size() + 1; i++) {
            string::size_type j = 0;
            for (; j < needle.size(); j++) {
                if (haystack[i + j] != needle[j]) break;
            }
            if (j == needle.size()) return i;
        }

        return -1;
    }
};
```

Java

```java
public class Solution {
    public int strStr(String haystack, String needle) {
        if (haystack == null && needle == null) return 0;
        if (haystack == null) return -1;
        if (needle == null) return 0;

        for (int i = 0; i < haystack.length() - needle.length() + 1; i++) {
            int j = 0;
            for (; j < needle.length(); j++) {
                if (haystack.charAt(i+j) != needle.charAt(j)) break;
            }
            if (j == needle.length()) return i;
        }

        return -1;
    }
}
```

Source Code Analysis

1. corner case: `haystack(source)` and `needle(target)` may be empty string.
2. code convention:
   - space is needed for `==`
   - use meaningful variable names
   - put a blank line before declaration `int i, j;`
3. declare j outside for loop if and only if you want to use it outside.

Some Pythonic notes: 4. More Control Flow Tools section 4.4 and if statement - Why does python use 'else' after for and while loops?

Complexity Analysis

nested for loop, $O((n-m)m)$ for worst case.

Search in Rotated Sorted Array

Question

- leetcode: (33) Search in Rotated Sorted Array
- lintcode: (62) Search in Rotated Sorted Array

Problem Statement

Suppose a sorted array is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

You are given a target value to search. If found in the array return its index, otherwise return -1.

You may assume no duplicate exists in the array.

Example

For `[4, 5, 1, 2, 3]` and `target=1`, return `2`.

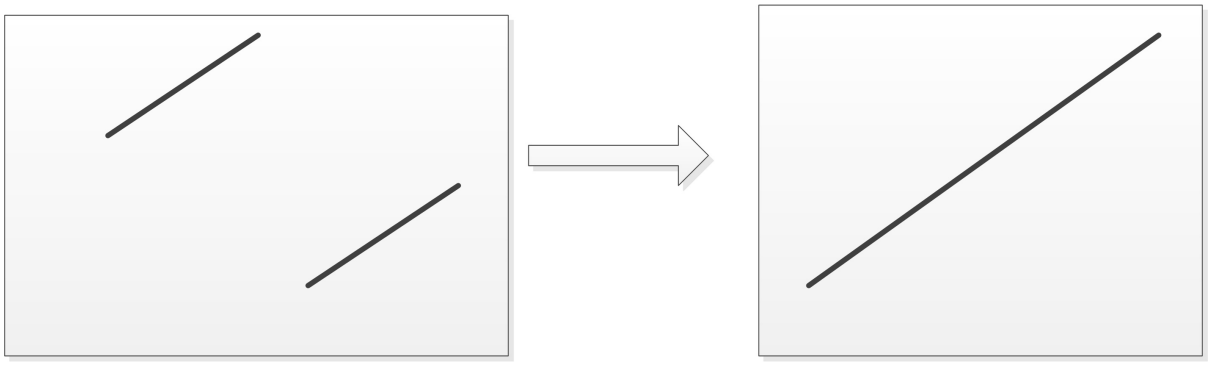
For `[4, 5, 1, 2, 3]` and `target=0`, return `-1`.

Challenge

O(logN) time

Solution1 - work on sorted subarray

Draw it. Rotated sorted array will take one of the following two forms:



Binary search does well in sorted array, while this problem gives an unordered one. Be patient. It is actually a combination of two sorted subarrayss. The solution takes full advantage of this. BTW, another approach can be comparing `target` with `A[mid]`, but dealing with lots of cases is kind of sophisticated.

C++

```
/**
 * fork
 * http://www.jiuzhang.com/solutions/search-in-rotated-sorted-array/
 */
```

```
class Solution {
    /**
     * param A : an integer ratated sorted array
     * param target :  an integer to be searched
     * return : an integer
     */
public:
    int search(vector<int> &A, int target) {
        if (A.empty()) {
            return -1;
        }

        vector<int>::size_type start = 0;
        vector<int>::size_type end = A.size() - 1;
        vector<int>::size_type mid;

        while (start + 1 < end) {
            mid = start + (end - start) / 2;
            if (target == A[mid]) {
                return mid;
            }
            if (A[start] < A[mid]) {
                // situation 1, numbers between start and mid are sorted
                if (A[start] <= target && target < A[mid]) {
                    end = mid;
                } else {
                    start = mid;
                }
            } else {
                // situation 2, numbers between mid and end are sorted
                if (A[mid] < target && target <= A[end]) {
                    start = mid;
                } else {
                    end = mid;
                }
            }
        }

        if (A[start] == target) {
            return start;
        }
        if (A[end] == target) {
            return end;
        }
        return -1;
    }
};
```

Java

```
public class Solution {
    /**
     *@param A : an integer rotated sorted array
     *@param target :  an integer to be searched
     *return : an integer
     */
    public int search(int[] A, int target) {
        if (A == null || A.length == 0) return -1;

        int lb = 0, ub = A.length - 1;
        while (lb + 1 < ub) {
            int mid = lb + (ub - lb) / 2;
            if (A[mid] == target) return mid;

            if (A[mid] > A[lb]) {
                // case1: numbers between lb and mid are sorted
                if (A[lb] <= target && target <= A[mid]) {
```

```
                    ub = mid;
                } else {
                    lb = mid;
                }
            } else {
                // case2: numbers between mid and ub are sorted
                if (A[mid] <= target && target <= A[ub]) {
                    lb = mid;
                } else {
                    ub = mid;
                }
            }
        }

        if (A[lb] == target) {
            return lb;
        } else if (A[ub] == target) {
            return ub;
        }
        return -1;
    }
}
```

Source Code Analysis

1. If `target == A[mid]`, just return.
2. Observe the two sorted subarrays, we can find that the least one of the left is greater than the biggest of the right. So if `A[start] < A[mid]`, then interval [start, mid] will be sorted.
3. Do binary search on `A[start] ~ A[mid]` on condition that `A[start] <= target <= A[mid]`.
4. Or do binary search on `A[mid]~A[end]` on condition that `A[mid] <= target <= A[end]`.
5. If while loop ends and none `A[mid]` hits, then examine `A[start]` and `A[end]`.
6. Return -1 if `target` is not found.

Complexity

The time complexity is approximately ***O(log n)***.

Solution2 - double binary search

Do binary search twice: first on the given array to find the break point; then on the proper piece of subarray to search for the target.

It may take a small step to see why the given array is binary-searchable. Though a rotated array itself is neither sorted nor monotone, there is implicit monotonicity. All elements on the left of break point are ≥A[0], and those on the right of break point are <A[0]. In a binary search, we keep narrowing the search scope by dropping the left or right half of the sequence, and here in the rotated array, we can do that much similarly.

To formalize, define an array `A'` that `A'[i] = A[i] < A[0] ? true : false`. If `A` is `[4, 5, 6, 7, 0, 1, 2]`, `A'` will be `[false, false, false, false, true, true, true]`. Surely `A'` monotone.

Java

```
public class Solution {
    /**
     *@param A : an integer rotated sorted array
     *@param target :  an integer to be searched
     *return : an integer
     */
```

```
    public int search(int[] A, int target) {
        if (A == null || A.length == 0) {
            return -1;
        }

        int p = findBreakPoint(A);
        if (target >= A[0]) {
            // search in [lo, segPoint]
            return binSearch(A, target, 0, p);
        } else {
            // search in [segPoint, hi]
            return binSearch(A, target, p, A.length - 1);
        }
    }

    private int findBreakPoint(int[] A) {
        // A[index] < A[0], min[index]
        int index;

        int lo = 0, hi = A.length - 1, segValue = A[0];
        while (lo + 1 < hi) {
            int md = lo + (hi - lo)/2;
            if (A[md] > segValue) {
                lo = md;
            } else {
                hi = md;
            }
        }
        index = A[lo] < segValue ? lo : hi;

        return index;
    }

    private int binSearch(int[] A, int target, int lo, int hi) {
        while (lo + 1 < hi) {
            int md = lo + (hi - lo) / 2;
            if (A[md] == target) {
                lo = md;
            } else if (A[md] < target) {
                lo = md;
            } else {
                hi = md;
            }
        }

        if (A[lo] == target) {
            return lo;
        }
        if (A[hi] == target) {
            return hi;
        }
        return -1;
    }
}
```

Complexity

The first binary search costs ***O(log n)*** time complexity, and the second costs no more than ***O(log n)***.

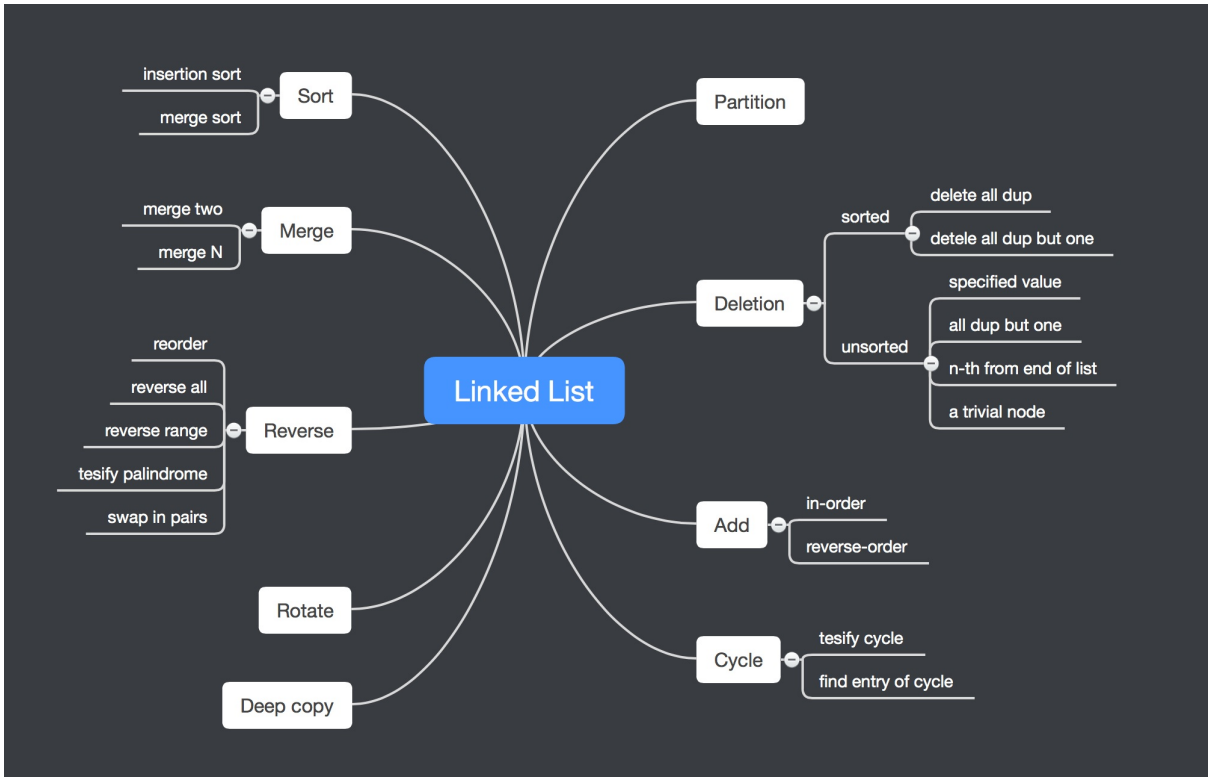
Linked List

This section includes common operations on linked list, such as deletion, insertion, and merging.

Frequently made mistakes:

- Not updating runner-node when traversing linked list
- Not recording head node before traversing
- returning incorrect pointer to node

The image below serves as a summarization.



Reverse Linked List

Question

- leetcode: (206) Reverse Linked List | LeetCode OJ
- lintcode: (35) Reverse Linked List

```
Reverse a linked list.

Example
For linked list 1->2->3, the reversed linked list is 3->2->1

Challenge
Reverse it in-place and in one-pass
```

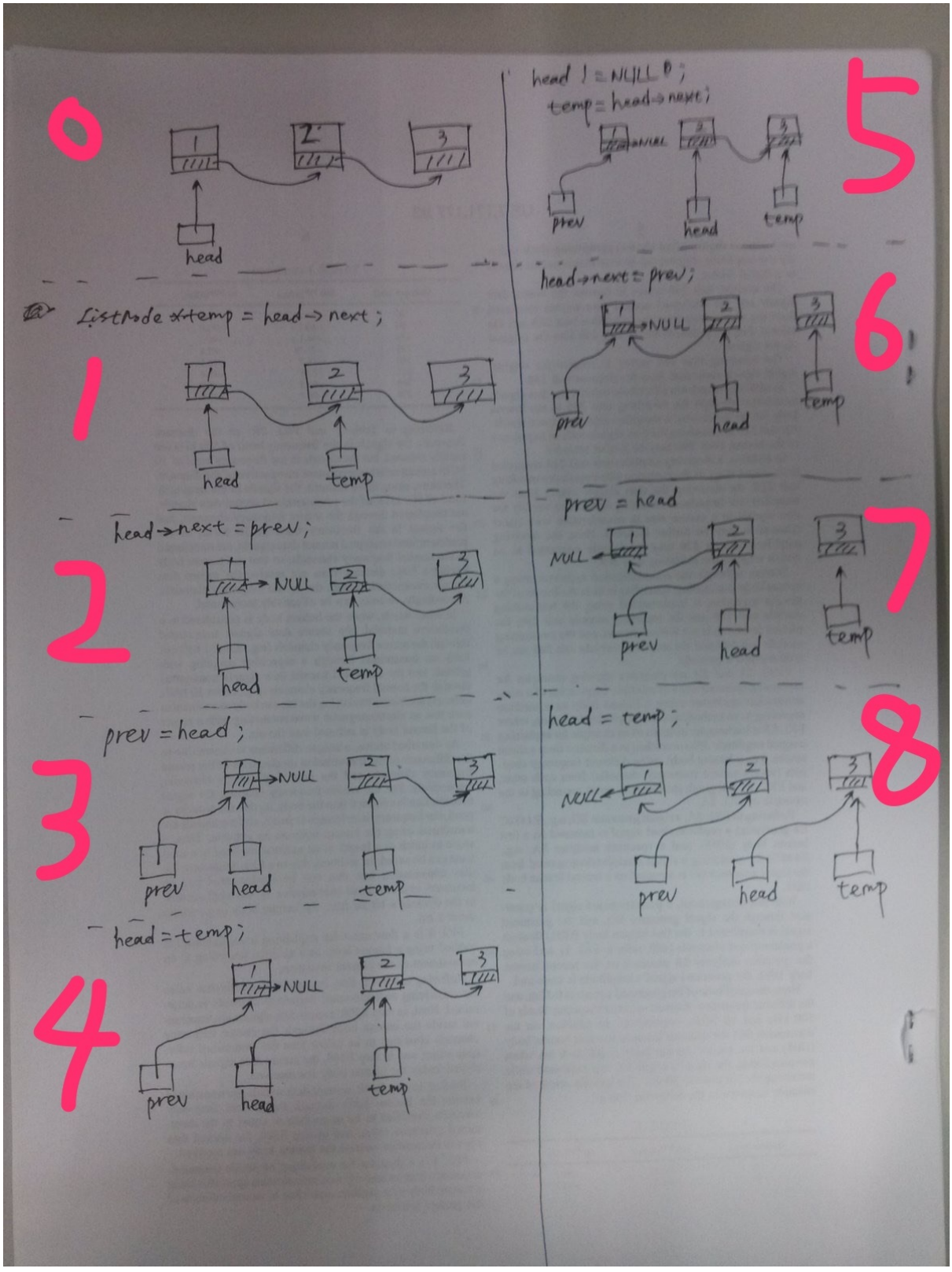
Solution1 - Non-recursively

It would be much easier to reverse an array than a linked list, since array supports random access with index, while singly linked list can ONLY be operated through its head node. So an approach without index is required.

Think about how '1->2->3' can become '3->2->1'. Starting from '1', we should turn '1->2' into '2->1', then '2->3' into '3->2', and so on. The key is how to swap two adjacent nodes.

```
temp = head -> next;
head->next = prev;
prev = head;
head = temp;
```

The above code maintains two pointer, `prev` and `head`, and keeps record of next node before swapping. More detailed analysis:



1. Keep record of next node
2. change `head->next` to `prev`
3. update `prev` with `head`, to keep moving forward
4. update `head` with the record in step 1, for the sake of next loop

Python

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def reverseList(self, head):
        prev = None
        curr = head
        while curr is not None:
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp
        # fix head
        head = prev

        return head
```

C++

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* reverse(ListNode* head) {
        ListNode *prev = NULL;
        ListNode *curr = head;
        while (curr != NULL) {
            ListNode *temp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = temp;
        }
        // fix head
        head = prev;

        return head;
    }
};
```

Java

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
```

```
        while (curr != null) {
            ListNode temp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = temp;
        }
        // fix head
        head = prev;

        return head;
    }
}
```

Source Code Analysis

Already covered in the solution part. One more word, the assignment of `prev` is neat and skilled.

Complexity

Traversing the linked list leads to ***O(n)*** time complexity, and auxiliary space complexity is ***O(1)***.

Solution2 - Recursively

Three cases when the recursion ceases:

1. If given linked list is null, just return.
2. If given linked list has only one node, return that node.
3. If given linked list has at least two nodes, pick out the head node and regard the following nodes as a sub-linked-list, swap them, then recurse that sub-linked-list.

Be careful when swapping the head node (refer as `nodeY`) and head of the sub-linked-list (refer as `nodeX`): First, swap `nodeY` and `nodeX`; Second, assign `null` to `nodeY->next` (or it would fall into infinite loop, and tail of result list won't point to `null`).

Python

```
"""
Definition of ListNode

class ListNode(object):

    def __init__(self, val, next=None):
        self.val = val
        self.next = next
"""
class Solution:
    """
    @param head: The first node of the linked list.
    @return: You should return the head of the reversed linked list.
                  Reverse it in-place.
    """
    def reverse(self, head):
        # case1: empty list
        if head is None:
            return head
        # case2: only one element list
        if head.next is None:
            return head
        # case3: reverse from the rest after head
        newHead = self.reverse(head.next)
```

```
        # reverse between head and head->next
        head.next.next = head
        # unlink list from the rest
        head.next = None

        return newHead
```

C++

```
/**
 * Definition of ListNode
 *
 * class ListNode {
 * public:
 *     int val;
 *     ListNode *next;
 *
 *     ListNode(int val) {
 *         this->val = val;
 *         this->next = NULL;
 *     }
 * }
 */
class Solution {
public:
    /**
     * @param head: The first node of linked list.
     * @return: The new head of reversed linked list.
     */
    ListNode *reverse(ListNode *head) {
        // case1: empty list
        if (head == NULL) return head;
        // case2: only one element list
        if (head->next == NULL) return head;
        // case3: reverse from the rest after head
        ListNode *newHead = reverse(head->next);
        // reverse between head and head->next
        head->next->next = head;
        // unlink list from the rest
        head->next = NULL;

        return newHead;
    }
};
```

Java

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverse(ListNode head) {
        // case1: empty list
        if (head == null) return head;
        // case2: only one element list
        if (head.next == null) return head;
        // case3: reverse from the rest after head
        ListNode newHead = reverse(head.next);
        // reverse between head and head->next
```

```
            head.next.next = head;
            // unlink list from the rest
            head.next = null;

            return newHead;
        }
    }
```

Source Code Analysis

case1 and case2 can be combined.What case3 returns is head of reversed list, which means it is exact the same Node (tail of origin linked list) through the recursion.

Complexity

The depth of recursion: ***O(n)***. Time Complexity: ***O(N)***. Space Complexity (without considering the recursion stack): ***O(1)***.

Reference

- - -
- data structures - Reversing a linked list in Java, recursively - Stack Overflow
- C++ |
- iteratively and recursively Java Solution - Leetcode Discuss

Tags