
# Algorithm

http://algorithm.yuanbin.me

# Table of Contents

# Data Structure and Algorithm/leetcode/lintcode


- English via Data Structure and Algorithm notes
- 简体中文请戳 数据结构与算法/leetcode/lintcode题解
- 繁體中文請瀏覽 資料結構與演算法/leetcode/lintcode題解

## 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, 七牛 CDN(中国大陆用户适用) - Recommended for iPhone/iPad/MAC
  2. PDF: GitHub, Gitbook, 七牛 CDN(中国大陆用户适用) - Recommended for Desktop
  3. MOBI: GitHub, Gitbook, 七牛 CDN(中国大陆用户适用) - 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
- 简体中文 is maintained by @billryan, @Shaunwei
- 繁體中文 is maintained by @CrossLuna

Other contributors can be found in Contributors to algorithm-exercise

### Donation

本项目接受捐赠，形式不限，可以买书，可以寄明信片，也可以金额打赏 :)

## 邮寄明信片

@billryan 喜欢收集各种明信片，来者不拒~ 邮寄的话可以邮寄至 `上海市闵行区上海交通大学闵行校区电院群楼5号楼307` ，收件人： `袁斌` 。

## 送书

除了邮寄明信片，你还可以买本书送给各位贡献者，@billryan 的地址见上节。

## 支付宝

账户名：yuanbin2014(at)gmail.com 金额随意

## Wechat

金额随意

## PayPal

账户名：yuanbin2014(at)gmail.com 金额随意，付款时选择 friends and family

隐私考虑，以下名单隐去了部分个人信息，有些名单若没来得及添加，可私下联系我加上，有些信息和金额因为时间久远可能有误，欢迎指正。

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

所得捐款用于七牛 CDN 流量付费/激励 Contributors 写出更好的内容/购买书籍/西瓜/饮料

# To Do

- [ ] add multiple languages support, currently 繁體中文, 简体中文 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
- 繁體中文請移步 貢獻指南
- 简体中文请移步 贡献指南

# 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) # ['s', 'h', 'a', 'u', 'n', 'w', 'e', 'i']
# 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.

# Linked List

> https://en.wikipedia.org/wiki/Linked_list
>
> https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm

Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.

Linked list is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. It is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.

The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while an array has to be declared in the source code, before compiling and running the program.

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing.

# Code

## Python

> http://interactivepython.org/courselib/static/pythonds/BasicDS/ImplementinganUnorderedListLinkedLists.html

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None
```

# Basic Linked List Implementation

## Reverse a linked list

## One-way reverse a linked list

The basic structure of the linked list is: `1 -> 2 -> 3 -> null`, after reversing it becomse `3 -> 2 -> 1 -> null`. Note that

- one will need to check whether curt is null or not when visiting the node curt.next;
- need to point the last node after reversing (the first node before reversing) to null.

## Python

> http://www.geeksforgeeks.org/write-a-function-to-reverse-the-nodes-of-a-linked-list/

```
# Python program to reverse a linked list
# Time Complexity : O(n)
# Space Complexity : O(1)
```

```
# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to reverse the linked list
    def reverse(self):
        prev = None
        current = self.head
        while(current is not None):
            next = current.next
            current.next = prev
            prev = current
            current = next
        self.head = prev

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next

# Driver program to test above functions
llist = LinkedList()
llist.push(20)
llist.push(4)
llist.push(15)
llist.push(85)

print "Given Linked List"
llist.printList()
llist.reverse()
print "\nReversed Linked List"
llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)


# A simple tail recursive method
# Simple and tail recursive Python program to
# reverse a linked list

# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
```

```
    # Function to initialize head
    def __init__(self):
        self.head = None


    def reverseUtil(self, curr, prev):

        # If last node mark it head
        if curr.next is None :
            self.head = curr

            # Update next to prev node
            curr.next = prev
            return

        # Save curr.next node for recursive call
        next = curr.next

        # And update next
        curr.next = prev

        self.reverseUtil(next, curr)


    # This function mainly calls reverseUtil()
    # with previous as None
    def reverse(self):
        if self.head is None:
            return
        self.reverseUtil(self.head, None)


    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print temp.data,
            temp = temp.next


# Driver program
llist = LinkedList()
llist.push(8)
llist.push(7)
llist.push(6)
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)

print "Given linked list"
llist.printList()

llist.reverse()

print "\nReverse linked list"
llist.printList()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

## Doubly linked list

> https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_algorithm.htm
>
> https://en.wikipedia.org/wiki/Doubly_linked_list

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

## Python

> http://www.geeksforgeeks.org/reverse-a-doubly-linked-list/

```
# Program to reverse a doubly linked list

# A node of the doublly linked list
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
     # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function reverse a Doubly Linked List
    def reverse(self):
        temp = None
        current = self.head

        # Swap next and prev for all nodes of
        # doubly linked list
        while current is not None:
            temp = current.prev
            current.prev = current.next
            current.next = temp
            current = current.prev

        # Before changing head, check for the cases like
        # empty list and list with only one node
        if temp is not None:
            self.head = temp.prev

    # Given a reference to the head of a list and an
    # integer,inserts a new node on the front of list
    def push(self, new_data):

        # 1. Allocates node
        # 2. Put the data in it
        new_node = Node(new_data)

        # 3. Make next of new node as head and
        # previous as None (already None)
        new_node.next = self.head

        # 4. change prev of head node to new_node
        if self.head is not None:
            self.head.prev = new_node

        # 5. move the head to point to the new node
        self.head = new_node
```

```
    def printList(self, node):
        while(node is not None):
            print node.data,
            node = node.next

# Driver program to test the above functions
dll = DoublyLinkedList()
dll.push(2);
dll.push(4);
dll.push(8);
dll.push(10);

print "\nOriginal Linked List"
dll.printList(dll.head)

# Reverse doubly linked list
dll.reverse()

print "\n Reversed Linked List"
dll.printList(dll.head)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)
```

### delete a node in a linked list

One will need to know its previous node therefore there must be a pointer points to its previous node. There is another kind of deletion, called ``pseudo deletion''. It means that one copies one node that is identical to the one to be deleted, and then delete. This way one does not need to know its previous node.

All one needs to do is to change `prev -> next = prev -> next -> next`. The head node might change during the above process and one can take care of it via `Dummy Node'.

> http://opendatastructures.org/ods-python/3_2_DLList_Doubly_Linked_Li.html

## Robustness of the linked list

- need to check whether curt is null when visiting curt.next
- after all the operation, one will need to check whether there is a loop. If loop exists, set one end to be null.

## Dummy Node

Dummy node is an important technique in linked list.

The list itself has a head pointer to first node and optionally a tail pointer to last node and/or a count. With a dummy node, the first node previous pointer points to the dummy node and the last node next pointer points to the dummy node. The dummy nodes pointers can point to the dummy node itself or be null.

dummy -> head. Dummy node is usually used in singly linked list without forward pointer questions. It ensures the head wouldn't be lost during deletion. Besides, on some rare case, one uses dummy node to delete the head node. For example, Remove Doublicates From Sorted List II. THe current = current.next can not delete head element. Therefore one puts a dummy node in front of the head.

When the linked list head changes (modified or deleted), using a dummy node can simplify the code. Returning dummy.next is the new linked list.

## Fast/slow pointer

> https://www.quora.com/What-is-a-slow-pointer-and-a-fast-pointer-in-a-linked-list

Slow pointer and fast pointer are simply the names given to two pointer variables. The only difference is that, slow pointer travels the linked list one node at a time where as a fast pointer travels the linked list two nodes at a time. Given below is a basic code snippet for moving slow and fast pointers.

This concept can be used in cases like detecting a loop in a graph, finding the middle node of a linked list (better time complexity), flattening a linked list etc. All these examples use the idea of slow and fast pointers.

There are two main applications:

- quickly find the moddle node set up two pointer `*fast` and `*slow`. They both point to the head. `*fast` is 2 times faster than `*slow`. When `*fast` points to the end node, `*slow` is right in the middle.
- detecting a loop in the singly linked list same as before, set up two pointer `*fast` and `*slow` and they both pointed to the singly linked list head node. `*fast` is 2 times faster than `*slow`. If `*fast = NULL`, it means this singly linked list ends with `NULL` and there is no loop. If `*fast = *slow`, then the fast points catches up the slow one and there is a loop.

## Python

```python
class NodeCircle:
    def __init__(self, val):
        self.val = val
        self.next = None

    def has_circle(self, head):
        slow = head
        fast = head
        while (slow and fast):
            fast = fast.next
            slow = slow.next
            if fast:
                fast = fast.next
            if fast == slow:
                break
        if fast and slow and (fast == slow):
            return True
        else:
            return False
```

# Binary Tree

> http://www.geeksforgeeks.org/binary-tree-set-1-introduction/

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

The ith level of a binary tree has at most $$2^{i-1}$$ nodes; the depth k binary tree has at most $$2^k-1$ nodes; for any binary tree T, if

> http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/FullvsComplete.html

A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

# Code

## Python

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left, self.right = None, None
```

# String

String related topics are discussed in this chapter.

> **Note** 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.
>
> When dealing the strings, one usually starts editing from the end of the string and moves to the front. The reason for this is that there are usually enough space at the end of the string. One can modify directly without worrying about covering the previous data.

# Implement strStr

Tags: Two Pointers, String, Easy

# Question

- leetcode: 28-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(object):
    def strStr(self, haystack, needle):
        """
        :type haystack: str
        :type needle: str
        :rtype: int
        """
        # https://discuss.leetcode.com/topic/29848/my-answer-by-python/6
        # The time complexity is definitely not O(n), it is O((n-m)*m).
        # Since checking haystack[i:i+len(needle)] == needle is O(m) done O(n-m) times.
        # n - length of haystack m - length of needle
```

```
        if haystack is None or needle is None:
            return -1
        for i in range(len(haystack) - len(needle) + 1):
            if haystack[i:i+len(needle)] == needle:
                return i
        return -1

if __name__ == '__main__':
    print Solution().strStr('a', "")
```

## 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.

# What is KMP?

> https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

# Two Strings Are Anagrams

Tags: String, Cracking The Coding Interview, Easy

# Question

- lintcode: Two Strings Are Anagrams

## Problem Statement

Write a method `anagram(s,t)` to decide if two strings are anagrams or not.

**Clarification**

What is **Anagram**?
- Two strings are anagram if they can be the same after change the order of characters.

**Example**

Given s = `"abcd"` , t = `"dcab"` , return `true` .
Given s = `"ab"` , t = `"ab"` , return `true` .
Given s = `"ab"` , t = `"ac"` , return `false` .

**Challenge** **

O(n) time, O(1) extra space

# Solution 1 - hashmap string frequency

To check if two strings are anagrams or not, considering upper/lower cases and empty characters, one can check if the frequency of different characters are the same for two strings. For questions of comparing the number of characters, the usual way is to loop through two strings, get the frequencies. If they are not equal, then return `false` . Many simply string interview questions are based on this.

## Python

```
class Solution:
    """
    @param s: The first string
    @param b: The second string
    @return true or false
    """
    def anagram(self, s, t):
        return collections.Counter(s) == collections.Counter(t)
```

# Solution 2 - order the strings

Alternative solution is to order the strings first. If the ordered strings are identical, then they are anagrams.

## Python

```
class Solution:
    """
    @param s: The first string
    @param b: The second string
    @return true or false
    """
    def anagram(self, s, t):
        return sorted(s) == sorted(t)
```

# Source code analysis

Order s and t strings and then compare.

# Reference

- *CC150 Chapter 9.1* Chinese Edition p109

# Compare Strings

Tags: Basic Implementation, String, LintCode Copyright, Easy

# Question

- lintcode: Compare Strings

## Problem Statement

Compare two strings A and B, determine whether A contains all of the characters in B.

The characters in string A and B are all **Upper Case** letters.

## Notice

The characters of B in A are not necessary continuous or ordered.

**Example**

For `A = "ABCD"`, `B = "ACD"`, return `true`.

For `A = "ABCD"`, `B = "AABC"`, return `false`.

# Solution

Similar to Two Strings Are Anagrams. Now we want to check whether all the characters in B are in A, but not individual character. For example, there are two `A` in B="AABC" but only one `A` in A="ABCD". Therefore the return value should be false. Make sure to check with the interviewer.

It is not suitable to use double loops as in strstr. Note also that the characters in string A and B are all **Upper Case** letters. It is easy to get that one will need to get the frequency by iterating over A and B first, and then compare frequencies. Well, let us use Hash table.

## Python

```
class Solution:
    """
    @param A : A string includes Upper Case letters
    @param B : A string includes Upper Case letters
    @return :  if string A contains all of the characters in B return True else return False
    """
    def compareStrings(self, A, B):
        #letters =   collections.defaultdict(int)
        letters = {}
        for a   in  A:
            if a not in letters:
                letters[a]  =  1
            else:
                letters[a] += 1
        for b in B:
            if b not in letters:
                return False
            else:
                letters[b] -= 1
```

```
            if letters[b] < 0:
                return False
        return True

if __name__ == '__main__':
    print Solution().compareStrings("AABC", "ABCD")
```

or use collections defaultdict

```
import collections
class Solution:
    """
    @param A: A  string  includes  Upper  Case  letters
    @param B: A  string  includes  Upper  Case  letters
    @return : if  string A contains all of the characters in B return  True else return False
    """
    def compareStrings(self,  A,  B):
        letters =  collections.defaultdict(int)
        for a  in  A:
            letters[a]  +=  1
        for b in B:
            letters[b] -= 1
            if letters[b] < 0:
                return False
        return True

if __name__ == '__main__':
    print Solution().compareStrings("AABC", "ABC")
```

## Analysis

Python `dict` is a hash and it is very convinent to use hash. collections is also very useful but more difficult to analysi the complexity.

1, if len(B) > len(A) then `false`, including empty character. 2, use extra space to store the frequency

## complexity

loop through A, loop through B, the time complexity is $$O(n)$$ the worst, space complexity is $$O(1)$$.

# Group Anagrams

Tags: Hash Table, String, Medium

# Question

- leetcode: Group Anagrams
- lintcode: Group Anagrams

## Problem Statement

Given an array of strings, group anagrams together.

For example, given: `["eat", "tea", "tan", "ate", "nat", "bat"]`,
Return:

```
[
  ["ate", "eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

**Note:** All inputs will be in lower-case.

# Solution

In Two Strings Are Anagrams, we introduced the methods using sorting and hashmap to check the anagrams. Here we will use these two methods simutaneously. One can loop through the strings twice, one for sorting and one for saving the final results.

Alternatively, in Python, one can do sorting and hashmap saving in one loop, using `get` method

## Python

```
class Solution(object):
    def groupAnagrams(self, strs):
        """
        :type strs: List[str]
        :rtype: List[List[str]]
        """
        dictStrs = {}
        for str in strs:
            ls = tuple(sorted(str))
            print "ls", ls
#            if not dictStrs.has_key(ls):
#                dictStrs[ls] = []
#            dictStrs[ls].append(str)
#   The method get() returns a value for the given key. If key is not available then returns default value None.

            dictStrs[ls] = dictStrs.get(ls, []) + [str]
            print "dictStrs", dictStrs

#        resStrs = []
#        for group in dictStrs.values():
#            resStrs.append(group)
```

```
#
#         return resStrs
        return dictStrs.values()

if __name__ == '__main__':
    print Solution().groupAnagrams(["eat", "tea", "tan", "ate", "nat", "bat"])
```

# Tags