Ultimate Algorithm Cheatsheet

(By Wentao Lu, momi2020 at uw.edu)

October 17, 2016

1 Sorting

1.1 Comparison Sort

1.1.1 Merge Sort ~ O($NlogN$) running time, O(N) memory

Merging already sorted lists into a new sorted list. It firstly divide the original list into little lists until there are only 2 or less elements in the list. Then the algorithm swap the two elements if they are out of order. Finally the algorithm merges the lists into a sorted list of the original size.

1.1.2 Heapsort ~ O($NlogN$) running time, O(N) memory

Heapsort is based on the heap data structure, which is a priority queue. The queue always has either the maximum item or the minimum item. The insertion takes $O(logN)$ and the extraction takes O(1) therefore the overall running time of heapsort is $O(NlogN)$.

```
class Heap:
        def __init__(self):
                self.array = []

        def max_heapify(self, i):    # top-down
                l = i * 2
                r = i * 2 + 1
                largest = max((l, r, i), key = lambda x: self.array[x])
                if largest != i:
                        self.array[i], self.array[largest] = self.array[largest], self.array[i]
                        self.max_heapify(largest)

        def get_top(self):
                mx = self.array[0]
                self.array[0] = self.array.pop()
                self.max_heapify(0)
                return mx

        def insert(self, val):     # bottom-up
                self.array += val,
                i = len(self.array) - 1
                while i > 0 and self.array[i/2] < self.array[i]:
                        self.array[i/2], self.array[i] = self.array[i], self.array[i/2]
                        i = i/2
```

1.1.3 Quicksort ~ O($NlogN$) running time on average, ~ O(N^2) running time in worst case, O(logN) memory

Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected. The running time depends on the selection of the pivot.

1.2 Distribution Sort

1.2.1 Counting Sort

Counting sort is applicable when each input is known to belong to a particular set, S, of possibilities. The algorithm runs in O(|S| + n) time and O(|S|) memory where n is the length of the input.

1.2.2 Bucket Sort

Bucket sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

Application: controlling difference between elements(index, val, etc.)

If two item are in the same bucket: (something happens)

If two item are in the adjacent bucket: (something happens) ex: find out whether there are two distinct indices i and j in the array such that the difference between nums[i] and nums[j] is at most t and the difference between i and j is at most k.

```
def containsNearbyAlmostDuplicate(self, nums, k, t):
        if t < 0:
                return False
    n = len(nums)
        d = {}
    w = t + 1
        for i in xrange(n):
                m = nums[i] / w
        if m in d:
                return True
        if m - 1 in d and abs(nums[i] - d[m - 1]) < w:
            return True
        if m + 1 in d and abs(nums[i] - d[m + 1]) < w:
            return True
        d[m] = nums[i]
        if i >= k:
                        del d[nums[i - k] / w]
    return False
```

2 Searching

2.1 Binary Search

The concept of binary search is easy to understand. It has a running time of O(logN) therefore it is always a choice when searching in some sorted array.

Low - High: Keep the low and the high index of the current search, update the boundary with mid = (low + high) / 2

Step: Step initially is the half of the length of the full array. Decrease the step by the factor of 2 and determine the direction of the step acoordingly.

2.2 String matching

2.2.1 Rabin-Karp Algorithm

Uses a hash function to accelerate the substring comparison since hash value is usually shorter than the substring. However, the hash function might get a collision therefore a complete comparison is necessary to verify. Luckily, if the hash function is good enough, the collision will be rare therefore the expectation of running time is good. When using a rolling hash, i.e., you can get the sequential hash value with previous hash value, the hash can be done in constant time.

Worst-case: collision every time, O(mn)

2.2.2 KMP Algorithm O(k+n)

Preprocess the **pattern** to accelerate searching in the way that increase index based on existed match when the comparison fails instead of increase by one.

W = ABCDABD S = ABC ABCDAB ABCDABCDABDE Preprocess the pattern W, we can get: T = [0,0,0,0,1,2,0] For a given location i, T[i] is the length of longest string which is a prefix of pattern **ending at** the location i-1.

```
def failure_table(array):
        j, i = 0, 1
        t = [None] * len(array)
        t[0] = 0
        while i < len(array):
                if array[i] == array[j]:
                        t[i] = j + 1
                        i += 1
                        j += 1
                elif j != 0:
                        j = t[j-1]
                else:
                        t[i] = 0
                        i += 1
        return t

def kmpAllMatches(pattern, text):
        ans = []
        t = failure_table(pattern)
        i, j = 0, 0
        while i < len(text):
                if text[i] == pattern[j]:
                        i += 1
                        j += 1
                        if j == len(pattern):
                                ans += i - j,
                                j = t[j-1]
                elif j != 0:
```

```
                j = t[j-1]
            else:
                i += 1
    return ans
```

2.2.3 Z Algorithm (optional)

Z value: the length of the longest substring of P that starts at i > 1 and matches a prefiex of P.

We form a new string P$T where $ does not exsit in either pattern or target. If we can compute the z values for all the indices of the string, the indices at which the z value equals to pattern length is a match location.

Compute the Z value using Z boxes:

l,r are the boundaries of the rightmost z box.
We have three cases:
1. if k > r, compute Z directly.
2. if k < r:
k' = k - r + 1 (Since the z box matches a prefix string, this step simply refer the z values of the locations in the prefix)
if k + Z(k') > r : Extends the current z box from r by comparison, move l to k, r to the point of failure.
if not: set z(k) = z(k')

2.2.4 Boyer-Moore Algorithm

The algorithm shift the pattern from the left of the target to the right but the substring comparison starts from the right to the left. Then the algorithm shift the pattern with the maximum distance calculated by the following two rules:

The Bad Character Rule:

When the comparison failed, the pattern will be shifted to the right in a way that the mismatched charactor in the target string will **match the corresponding charactor in the pattern**. If such shift is not possible, i.e., the mismatch does not exist in from the mismatch location to the left of the pattern, then the pattern will shift completely pass the mismatched character in the target string.

The Good Suffix Rule:

When the comparison failed, suppose the pattern's suffix has already matched a substring t in the target string, the pattern will be shifted to the right in a way such that the pattern will match the same substring, but the subsequent character is different from t in pattern. (look up in table L.)
If such substring does not exist in the left part of pattern, shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T.
If no such shift exists, then shift P past t.
If we find a match, shift in a way that a prefix P will match the suffix in T. If not possible, shift past t. (look up in table H)

2.3 Preprocessing

We need two tables for the Good Suffix Rule.

L[i] is the largest position less than n such that the string P[i:n] (inclusive) matches a suffix of P[1:L[i]] (inclusive)

Let$H[i]$ denote the length of the largest suffix of $P[i..n]$ that is also a prefix of P, if one exists. If none exists, let $H[i]$ be zero.

2.4 The Galil Rule

After the shift, we might be able to skip some of the substring comparison. This leads to a O(m+n) running time in worst cases.

3 String

3.1 Sliding window maximum/ mininum (monitonic queue)

```
def slidingWindow(array, size):
        queue = []
        ret = []
        for i, n in enumerate(array):
                while queue and array[queue[-1]] < n: # define your useless item
                        queue.pop()  # clean useless items
                queue += i,
                if d[0] == i - size: # fall out of the window
                        d.pop(0)
                if i >= size - 1:
                        ret += nums[d[0]],
        return ret
```

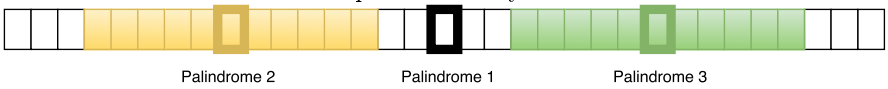
3.2 Palindrome

Palindrome is a kind of string that the string reads forward the same as it reads backward. Palindrome has its center and if a palindrome's first letter and the last letter is removed, it will remain a palindrome.

3.2.1 Manacher's Algorithm O(N)

The algorithm is based on following observations of palindrome.

1. The left side of a palindrome is a mirror image of its right side.
2. For even-length palindromes, the center is at the boundary of the two characters in the middle.
3. Use the first palindrome as a reference. If the second palindrome's center is within the boundary of the first palindrome:
   (a) (Case 1) A third palindrome whose center is within the right side of a first palindrome will have exactly the same length as that of a second palindrome anchored at the mirror center on the left side, if the second palindrome is within the bounds of the first palindrome by at least one character.



   (b) (Case 2) If the second palindrome meets or extends beyond the left bound of the first palindrome, then the third palindrome is guaranteed to have at least the length from its own center to the right outermost character of the first palindrome. This length is the same from the center of the second palindrome to the left outermost character of the first palindrome.

   (c) To find the length of the third palindrome under Case 2, the next character after the right outermost character of the first palindrome would then be compared with its mirror character about the center of the third palindrome, until there is no match or no more characters to compare.
4. If the second palindrome's center is out of the boundary of the first palindrome, the two palindromes we have give us no information about the next palindrome.
5. Update the reference palindrone to a palindrone so we can have the right most charactor we had in a palindrone so far. In the case 1, the reference is not updated. In the case 2, the reference is updated to palindrome 3.
6. Regarding the time complexity of palindromic length determination for each character in a string, we only do charactor comparison when we are extending the right most character we have seen in all the palindromes so far. Therefore, the right most character can be only extended N times, since the string has a length of N. However, the liner performance here is only considering the number of comparisions. If the are more than N palindromes in the string (consider the string "aaaaaaaaaaaaaaaaaaaaaaa"), the algorithm is linear regarding the number of palindromes in the string (m) and the string length, since we do N comparisions and possibly O(m) referal. Thus the complexity is O (m+n).

```
def Manacher_algorithm(s):
        newS = '#' + '#'.join(s) + '#'
        center, rightMost, p = 0, 0, [0] * len(newS)
        for in xrange(1, len(newS)-1):  # exclude boundaries
                p[i] = (rightMost > i) and min(rightMost - i, p[2 * center - i])
                while newS[i + 1 + p[i]] == T[i - 1 - p[i]]:
                        p[i] += 1
                if i + p[i] > rightMost:
                        center, rightMost = i, i + p[i]
        return p
```

4 Linked List

5 Graph

5.1 Union-Find

```
def find(x):
        if parent[x] != x:
                parent[x] = find(parent[x])
        return parent[x]

parent = range(n)
for s, e in edges:
        x = find(s)
        y = find(e)
        if x == y:
                return False
        parent[x] = y
return len(edges) == n - 1
```

5.2 Tree

5.2.1 Morris Inorder/Preorder Traversal

```
cur, prev = root, None
while cur:
        if not cur.left:
                print cur.val
        cur = cur.right
        else:
                prev = cur.left
                while prev.right and prev.right != cur:
                        prev = prev.right
                if not prev.right:
                        # pre_order print here
                        prev.right = cur
                        cur = cur.left
                else:
                        # in_order print here
                        prev.right = None
                        cur = cur.right
```

5.2.2 Binary Indexed Tree

```
# Note in BIT the index starts from 1!
# Python 0,1,2,3,4,5,6
# BIT     1,2,3,4,5,6,7

def read(index, array):
        sum = 0
        while index:
                sum += array[index]
                index -= (index & - index) # least non-zero significant bit
        return sum

def update(index, array, diff):
        while index < len(array):
                array[index] += diff # diff = new - old
                index += index & - index
```

6 Bitwise

6.1 Word char hash

```
def h(word):
        ret = 0
        for c in h:
                ret |= 1 << (ord(c) - 97)
        return ret
```

7 Math

7.1 Boyer-Moore Voting for items appear more than n/k times in an array

```
def voting(k, array): # O(n) time, O(k) space
        dic = {}
        for item in array:
                if item in dic:
                        dic[item] += 1
                elif item not in dic and len(dic) < k - 1:
                        dic[item] = 1
                else:
                        dic = {item: dic[item] - 1 for item in dic if dic[item] - 1 > 0} # O(k) bu
        return [item for item in dic if array.count(item) > len(array) / k] # verification
```

8 Method

8.1 Dynamic Programming

8.2 Two Pointers

Two pointers can have two applications:

8.2.1 Slow-fast in linked list.

8.2.2 Left-right in sorted array.

N-sum: Given an array of numbers, find the combination of N indices where numbers sum up to a target value.

Sort the array first. This step takes NlogN time but can eliminate a lot of recursion. Dynamic programming can reduce such problem to two sum by changing the target regarding the number at the current index and update the result array. (Similar to DFS) Solve two sum at the base case: if smaller than target, increase left; if larger than target, decrease right; if equals to target, append previous recursion route and the current two sum result.

8.3 Deterministic Finite Automaton

It is extremely useful when dealing with somehow complex to write otherwise. If a situation has finite number of states which can be represented as a graph and the graph requires a lot of conditional statement to jump between blocks, the DFA can be a clear and easier way to write it.

8.4 Three-way partitioning:

Similar to color sort, we use two pointers to track the end of color 0 and the start of color 2.

If the cur pointer hit color 1, then we simply move to the next

If the cur pointer hit color 0, then we swap the val at cur and the pointer for 0

If the cur pointer hit color 2, then we swap the val at cur and the pointer for 2

8.5 Backtracking

```
class Solution(object):
    def combine(self, n, k):
        ans = []
        stack = []
        x = 1
        while True:
            l = len(stack)
            if l == k:
                ans.append(stack[:])
            if l == k or x > n - k + l + 1:
                if not stack:
                    return ans
                x = stack.pop() + 1
            else:
                stack.append(x)
                x += 1
```