`github.com/algorhythms/Algo-Quicksheet`

github.com/idf

Algorithm Quicksheet

Classical equations, diagrams and tricks in algorithm

October 8, 2015

This book is dedicated to all Software Engineers.

Preface

Introduction

This quicksheet contains many classical equations and diagrams for algorithm, which will help you quickly recall knowledge and ideas in algorithm.

This quicksheet has three significant advantages:

1. Non-essential knowledge points omitted
2. Compact knowledge representation
3. Quick recall

How to Use This Quicksheet

You should not attempt to remember the details of an algorithm. Instead, you should know:

1. What problems this algorithm solves.
2. The benefits of using this algorithm compared to others.
3. The important clues of this algorithm so that you can derive the details of the algorithm from them.

Only dives into the code when you is unable to reconstruct the algorithm from the hits and and the important clues.

At GitHub, June 2015 *github.com/idf*

Contents

List of Contributors

Daniel D. Zhang (`github.com/idf`)

Notations

INTRODUCTION

Algorithm notations.

GENERAL MATH NOTATION

Symbol	Meaning
$\lfloor x \rfloor$	Floor of x, i.e. round down to nearest integer
$\lceil x \rceil$	Ceiling of x, i.e. round up to nearest integer
floor(key)	the largest key $\leq$ the given key
ceil(key)	the smallest key $\geq$ the given key
$\log x$	The base of logarithm is 2 unless otherwise stated
$a \wedge b$	Logical AND
$a \vee b$	Logical OR
$\neg a$	Logical NOT
∞	Infinity
$\rightarrow$	Tends towards, e.g., $n \rightarrow \infty$
$\propto$	Proportional to; $y = ax$ can be written as $y \propto x$
$\lvert x \rvert$	Absolute value
$\lvert\lvert \boldsymbol{a} \rvert\rvert$	L_2 distance (Euclidean distance) of a vector
$\lvert \mathcal{S} \rvert$	Size (cardinality) of a set
$n!$	Factorial function
$\triangleq$	Defined as
$O(\cdot)$	Big-O: roughly means order of magnitude
$\mathbb{R}$	The real numbers
$0:n$	Range (Python convention): $0:n = 0,1,2,...,n-1$
$\approx$	Approximately equal to
$\arg\max_x f(x)$	Argmax: the value x that maximizes f
$\binom{n}{k}$	n choose k , equal to $\frac{n!}{k!(n-k)!}$

Chapter 1
Time Complexity

1.1 Basic Counts

Double for loop

$$\sum_{i=1}^{N}\sum_{j=i}^{N}1=\binom{N}{2}\sim\frac{1}{2}N^2$$

$$\sum_{i=1}^{N}\sum_{j=i}^{N}1\sim\int_{x=1}^{N}\int_{y=x}^{N}\mathrm{d}y\,\mathrm{d}x$$

Triple for loop

$$\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=1}^{N}1=\binom{N}{3}\sim\frac{1}{6}N^3$$

$$\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=1}^{N}1\sim\int_{x=1}^{N}\int_{y=x}^{N}\int_{z=y}^{N}\mathrm{d}z\,\mathrm{d}y\,\mathrm{d}x$$

1.2 Solving Recurrence Equations

Basic recurrence equation solving techniques:

1. Guessing and validation
2. Telescoping
3. Recursion tree
4. Master Theorem

1.2.1 Master Theorem

Recurrence relations:

$$T(n)=a\,T\left(\frac{n}{b}\right)+f(n),\text{ where }a\geq 1,\,b>1$$

Notice that $b>1$ rather than $b\geq 1$.

Case 1

If:

$$f(n)=o(n^{\log_b a})$$

Then:

$$T(n)=\Theta(n^{\log_b a})$$

Notice that in the condition it is o rather than O.

Case 2

If:

$$f(n)=\Theta(n^{\log_b a}\log^k n)$$

for some constant $k\geq 0$

Then:

$$T(n)=\Theta(n^{\log_b a}\log^{k+1} n)$$

Case 3

If:

$$f(n)=\omega(n^{\log_b a})$$

And with regularity condition:

$$f(\frac{n}{b})\leq kf(n)$$

for some constant $k<1$ and sufficiently large n

Then:

$$T\left(n\right)=\Theta\left(f(n)\right)$$

Notice that in the condition it is ω rather than Ω.

1.3 Useful Math Equations

Euler:

$$\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+...+\frac{1}{n}=\ln n$$

Chapter 2
Memory Complexity

2.1 Introduction

2.1.1 Tables

The memory usage is based on Java.

Type	Bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

Type	Bytes
char[]	2N+24
int[]	4N+24
double[]	8N+24

Type	Bytes
char[][]	2MN
int[][]	4MN
double[][]	8MN

Type	Bytes
Object overhead	16
Reference	8
Padding	8x

Reference includes object reference and innner class reference.

Padding is to make the object memory size of 8's multiple.

2.1.2 Example

The generics is passed as Boolean:

```
public class Box<T> {   // 16 (object overhead)
    private in N;       // 4 (int)
    private T[] items;  // 8 (reference to array)
                        // 8N+24 (array of Boolean references)
                        // 24N (Boolean objects)
                        // 4 (padding to round up to a multiple)
}
```

Notice the multiple levels of references.

Chapter 3
Basic Data Structures

3.1 Introduction

Abstract Data Types (ADT):

1. Queue
2. Stack
3. HashMap

Implementation (for both queue and stack):

1. Linked List
2. Resizing Array:
   a. Doubling: when full (100%).
   b. Halfing: when one-quarter full (100%).

Python Library:

1. `collections.deque` [1]
2. `list`
3. `dict, OrderedDict, DefaultDict`

Java Library:

1. `java.util.Stack<E>`
2. `java.util.LinkedList<E>`
3. `java.util.HashMap<K, V>; java.util.TreeMap<K, V>`

3.2 Stack

3.2.1 Stack and Recursion

How a compiler implements a function:

1. Function call: push local environment and return address
2. Return: pop return address and local environment.

Recursive function: function calls itself. It can always be implemented by using an explicit stack to remove recursion.

[1] The naming in collections is awkward: discussion.

3.2.2 Largest Rectangle

Find the largest rectangle in the matrix (histogram). Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

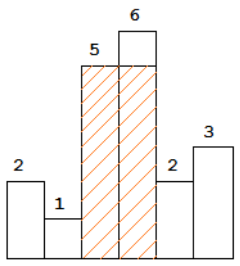


Fig. 3.1: Largest rectangle in histogram

Keep a stack storing the bars in strictly increasing order, then calculate the area by popping out the stack to get the currently lowest bar which determines the height of the rectangle.

Notice:

1. Maintain the non-decreasing stack
2. Calculation of the rectangle width
3. Post-processing in the end

```
def largestRectangleArea(self, height):
    n = len(height)
    gmax = -sys.maxint-1
    stk = []  # store the idx, non-decreasing stack

    for i in xrange(n):
        while stk and height[stk[-1]] > height[i]:
            last = stk.pop()
            if stk:  # calculate area when popping
                area = height[last]*(i-(stk[-1]+1))
            else:
                area = height[last]*i
            gmax = max(gmax, area)

        stk.append(i)

    # after processing all heights, process the remaining stack
    i = n
    ...

    return gmax
```

3.3 Map

3.3.1 Math Relations

OneToOneMap. One map, dual entries.

```
class OneToOneMap(object):
    def __init__(self):
        self.m = {}  # keep a single map

    def set(self, a, b):
        self.m[a] = b
        self.m[b] = a

    def get(self, a):
        return self.m.get(a)
```

3.3.2 Operations

Sorting by value:

```
from operators import itemgetter
sorted(hm.items(), key=itemgetter(1), reverse=True)
```

Chapter 4
Linked List

4.1 Operations

4.1.1 Fundamentals

Get the *pre* reference:

```
dummy = Node(0)
dummy.next = head
pre = dummy
cur = pre.next
```

4.1.2 Basic Operations

1. Get the length
2. Get the *i*-th object
3. Delete a node
4. Reverse

4.1.3 Combined Operations

In $O(n)$ without extra space:

1. Determine whether two lists intersects
2. Determine whether the list is palindrome
3. Determine whether the list is acyclic

Chapter 5
Heap

5.1 Introduction

Heap-ordered. Binary heap is one of the implementations of Priority Queue (ADT).

5.2 Operations

Assume the root **starts** at $a[1]$ rather than $a[0]$.
Basic operations:

1. sink()/ sift_down() - recursive
2. swim()/ sift_up() - recursive
3. build()/ heapify() - bottom-up sink()

5.2.1 Sink

```
def sink(self, idx):
    while 2*idx <= self.N:
        c = 2*idx
        if c+1 <= self.N and self.less(c, c+1):
            c += 1
        if not self.less(idx, c):
            return

        self.swap(idx, c)
        idx = c
```

5.2.2 Swim

```
def swim(self, idx):
    while idx > 1 and self.less(idx/2, idx):
        pi = idx/2
        self.swap(pi, idx)
        idx = pi
```

5.2.3 Heapify

Worst case: Heapifying **a sorted array** is the worst case for heap construction, because the root of each subheap considered sinks all the way to the bottom. The worst case complexity $\sim 2N$.

Building a heap is O(N) rather than $O(N\lg N)$.

Proof:

$$\because \sum_{i=0}^{+\infty} ix^i = \frac{x}{(1-x)^2}$$

$$\therefore \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$\therefore \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n)$$

5.3 Implementation

5.3.1 General

The self-implemented binary heap's index usually starts at 1 rather than 0.

The array representation of heap is in **level-order**.

A 3-heap is an array representation (using 1-based indexing) of a complete 3-way tree. The children of $a[k]$ are $a[3k-1]$, $a[3k]$, and $a[3k+1]$.

The main reason that we can use an array to represent the heap-ordered tree in a binary heap is because the tree is **complete**.

Suppose that we represent a BST containing N keys using an array, with $a[0]$ empty, the root at $a[1]$. The two children of $a[k]$ will be at $a[2k]$ and $a[2k+1]$. Then, the length of the array might need to be as large as 2^N.

5.3.2 Python Heapq

Python only has built in min-heap. To use max-heap, you can:

1. Revert the number: 1 becomes -1.
2. Wrap the data into another class and override **comparators**: __cmp__ or __lt__

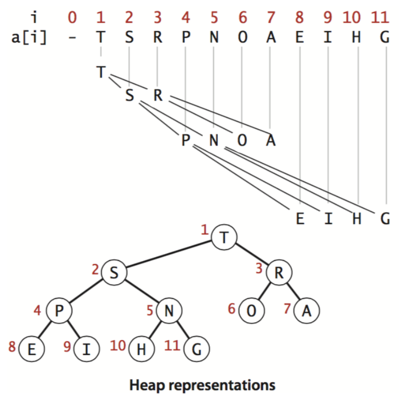


Fig. 5.1: Heap representation

The following code presents the wrapping method:

```
class Value(object):
    def __init__(self, val):
        self.val = val
        self.deleted = False  # lazy delete

    def __cmp__(self, other):
        # Reverse order by height to get max-heap
        assert isinstance(other, Value)
        return other.val - self.val
```

Normally the deletion by value in Python is $O(n)$, to achieve $O(\lg n)$ we can use **lazy deletion**. Before take the top of the heap, we do the following:

```
while heap and heap[0].deleted:
    heapq.heappop(heap)
```

5.4 Event-Driven Algorithm

The core philosophy of event-driven algorithm:

1. The definition of **event**; the event are sorted by time of appearance.
2. The definition of **heap meaning**
3. The definition of the **transition functions** among events impacting the heap.

Overlapping Interval. Given a list of number intervals, find max number of overlapping intervals.

Every new start of an interval is an **event**. Put the ending time into heap, and pop the ending time earlier than the new start time from heap.

```
def max_overlapping(intervals):
    maxa = 0

    intervals.sort(key=operator.attrgetter("start"))
    end_heap = []
    for itvl in intervals:
        heapq.heappush(end_heap, itvl.end)
        while end_heap and end_heap[0] <= itvl.start:
            heapq.heappop(end_heap)

        maxa = max(maxa, len(end_heap))

    return maxa
```

Chapter 6
Tree

6.1 Binary Tree

6.1.1 Introductions

Get parent ref. To get a parent reference (implicitly), return the current recursion function to its parent to maintain the path. Sample code:

```
Node deleteMin(Node x) {
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.count = 1+size(x.left)+size(x.right);
    return x;
}
```

Search. To search a node in binary tree (not necessarily BST), use dfs:

```
def dfs(self, root, t, path, found):
    if not root or found[0]:  # post-call check
        return

    path.append(root)
    if root == t:
        found[0] = True

    self.dfs(root.left, t, path, found)
    self.dfs(root.right, t, path, found)
    if not found[0]:
        path.pop()  # 1 pop() corresponds to 1 append()
```

The 'found' is a wrapper for boolean to keep it referenced by all calling stack.

Lowest common ancestor. In BST, the searching is straightforward. In normal binary tree, construct the path from root to $node_1$ and $node_2$ respectively, and **diff** the two paths.

Find all paths. Find all paths from root to leafs. For every currently visiting node, add itself to path; search left, search right and pop itself. Record current result when reaching the leaf.

```
def dfs_path(self, cur, path, ret):
    if not cur:
        return

    path.append(cur)
    if not cur.left and not cur.right:
        ret.append("->".join(map(lambda x: str(x.val), path)))

    self.dfs_path(cur.left, path, ret)
    self.dfs_path(cur.right, path, ret)
    path.pop()
```

6.2 Binary Search Tree (BST)

Array and BST. Given either the **preorder** or **postorder** (but not inorder) traversal of a BST containing N distinct keys, it is possible to reconstruct the shape of the BST.

6.2.1 Operations

Rank.

6.2.2 Range search

```
int size(Key lo, Key hi) {
    if (contains(hi)) return rank(hi)-rank(lo)+1;
    else              return rank(hi)-rank(lo);
}
```

1-d range search

6.3 Interval Search Tree

TODO

6.4 Segment Tree

6.4.1 Introduction

The structure of **Segment Tree** is a binary tree which each node has two attributes start and end denote an segment/interval. Notice that by practice, the interval is normally $[start, end)$ but sometimes it can be $[start, end]$, which depends on the question definition.

Structure:

```
                   [0,  4,  count=3]
                   /               \
        [0,2,  count=1]                 [2,4,  count=2]
        /            \                  /             \
[0,1,  count=1] [1,2,  count=0] [2,3,  count=1],  [3,4,  count=1]
```

Variants:

1. Sum Segment Tree.
2. Min/Max Segment Tree.
3. Count Segment Tree.

For a **Maximum Segment Tree**, which each node has an extra value max to store the maximum value in this node's interval.

6.4.2 Operations

Components in Segment Tree:

1. Build
2. Query
3. Modify

Notice:

1. Only build need to change the start and end recursively.
2. Pre-check is preferred in recursive calls.

Code:

```python
DEFAULT = 0
f = lambda x, y: x+y


class Node(object):
    def __init__(self, start, end, m):
        self.start, self.end, self.m = start, end, m
        self.left, self.right = None, None


class SegmentTree(object):
    def __init__(self, A):
        self.A = A
        self.root = self.build_tree(0, len(self.A))

    def build_tree(self, s, e):
        """
        segment: [s, e)
        """
        if s >= e:
            return None

        if s+1 == e:
            return Node(s, e, self.A[s])

        left = self.build_tree(s, (s+e)/2)
        right = self.build_tree((s+e)/2, e)
        val = DEFAULT
        if left: val = f(val, left.m)
        if right: val = f(val, right.m)
        root = Node(s, e, val)
        root.left = left
        root.right = right

        return root

    def query(self, root, s, e):
        """
        :type root: Node
        """
        if not root:
            return DEFAULT

        if s <= root.start and e >= root.end:
            return root.m

        if s >= root.end or e <= root.start:
            return DEFAULT

        l = self.query(root.left, s, e)
        r = self.query(root.right, s, e)
        return f(l, r)

    def modify(self, root, idx, val):
        """
        :type root: Node
        """
        if not root or idx >= root.end or idx < root.start:
            return

        if idx == root.start and idx == root.end-1:
            root.m = val
            self.A[idx] = val
            return

        self.modify(root.left, idx, val)
        self.modify(root.right, idx, val)
        val = DEFAULT
        if root.left: val = f(val, root.left.m)
        if root.right: val = f(val, root.right.m)
        root.m = val
```

Chapter 7
Balanced Search Tree

7.1 2-3 Search Tree

7.1.1 Insertion

Insertion into a 3-node at bottom:

1. Add new key to the 3-node to create a temporary 4-node.
2. Move middle key of the 4-node into the parent (including root's parent).
3. Split the modified 4-node.
4. Repeat recursively up the trees as necessary.

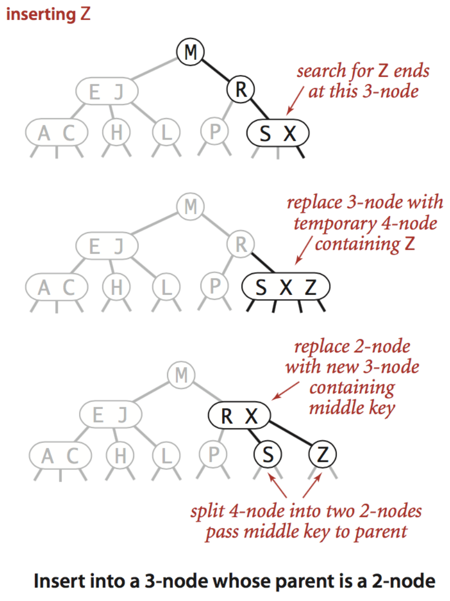

inserting Z
search for Z ends at this 3-node
replace 3-node with temporary 4-node containing Z
replace 2-node with new 3-node containing middle key
split 4-node into two 2-nodes pass middle key to parent
Insert into a 3-node whose parent is a 2-node

Fig. 7.1: Insertion 1

7.1.2 Splitting

Summary of splitting the tree.

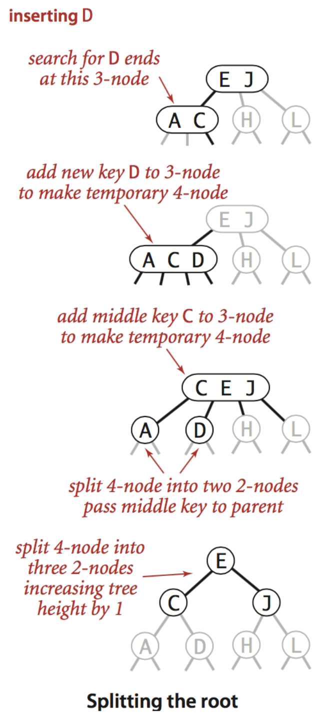

inserting D
search for D ends at this 3-node
add new key D to 3-node to make temporary 4-node
add middle key C to 3-node to make temporary 4-node
split 4-node into two 2-nodes pass middle key to parent
split 4-node into three 2-nodes increasing tree height by 1
Splitting the root

Fig. 7.2: insert 2

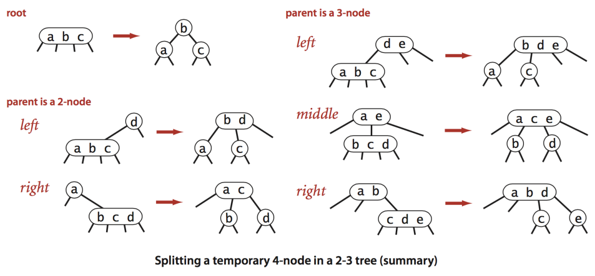


Fig. 7.3: Splitting temporary 4-ndoe summary

7.1.3 Properties

When inserting a new key into a 2-3 tree, under which one of the following scenarios must the height of the 2-3 tree increase by one? When every node on the search path from the root is a 3-node

7.2 Red-Black Tree

7.2.1 Properties

Red-black tree is an implementation of 2-3 tree using **leaning-left red link**. The hight of the RB-tree is at most

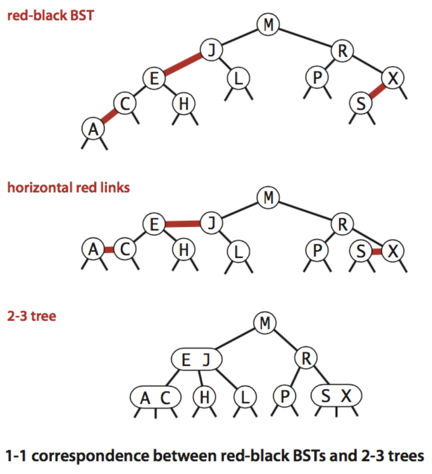


Fig. 7.4: RB-tree and 2-3 tree

$2\lg N$ where alternating red and black links.

Perfect black balance. Every path from root to null link has the same number of black links.

7.2.2 Operations

Elementary operations:

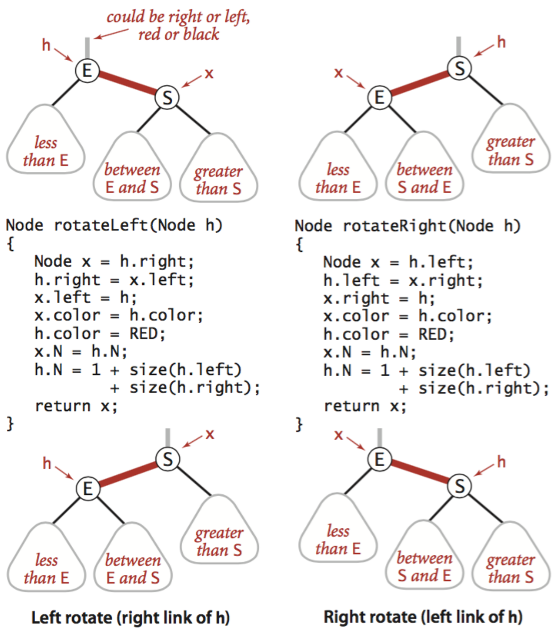
1. Left rotation: orient a (temporarily) right-leaning red link to lean left.
2. Right rotation: orient a (temporarily) left-leaning red link to lean right.
3. Color flip: Recolor to split a (temporary) 4-node.

Insertion. When doing insertion, from the child's perspective, need to have the information of current leaning direction and parent's color. Or from the parent's perspective - need to have the information of children's and grandchildren's color and directions.

For every new insertion, the node is always attached with red links.

The following code is the simplest version of RB-tree insertion:

```
Node put(Node h, Key key, Value val) {
```

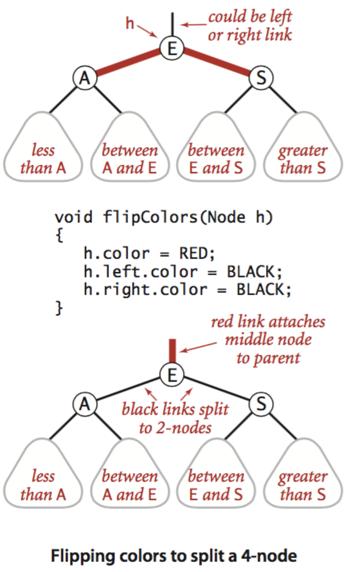


could be right or left, red or black
h
x
less than E
between E and S
greater than S
Left rotate (right link of h)
Right rotate (left link of h)

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

Fig. 7.5: Rotate left/right



h
could be left or right link
less than A
between A and E
between E and S
greater than S
red link attaches middle node to parent
black links split to 2-nodes
Flipping colors to split a 4-node

```
void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

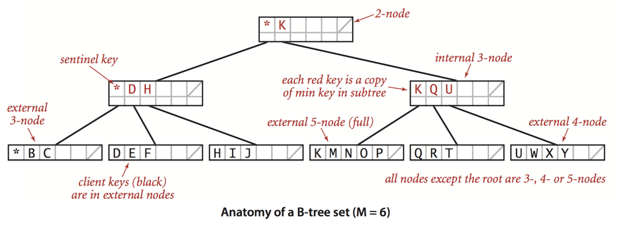
Fig. 7.6: Flip colors

```
    if (h == null)  // std insert, with red link to parent.
      return new Node(key, val, 1, RED);
    int cmp = key.compareTo(h.key);
    if      (cmp < 0) h.left  = put(h.left,  key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else h.val = val; // pass
    if (isRed(h.right) && !isRed(h.left))    h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right))     flipColors(h);
    h.N = 1+size(h.left)+size(h.right);
    return h;
}
```

Rotate left, rotate right, then flip colors.

Deletion. Deletion is more complicated.

7.3 B-Tree



2-node
sentinel key
internal 3-node
each red key is a copy of min key in subtree
external 3-node
external 5-node (full)
external 4-node
client keys (black) are in external nodes
all nodes except the root are 3-, 4- or 5-nodes
Anatomy of a B-tree set (M = 6)

Fig. 7.7: B-Tree

7.4 AVL Tree

TODO

Chapter 8
Trie

8.1 Data Structure

8.1.1 Basic

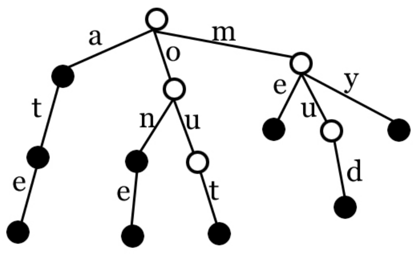


Fig. 8.1: Trie

Notice:

1. Children are stored in HashMap rather than ArrayList.
2. self.word to stores the word and indicates whether a word ends at the current node.

Code:

```
class TrieNode(object):
    def __init__(self, char):
        self.char = char
        self.word = None
        self.children = {}  # map from char to TrieNode


class Trie(object):
    def __init__(self):
        self.root = TrieNode(None)

    def add(self, word):
        word = word.lower()
        cur = self.root
        for c in word:
            if c not in cur.children:
                cur.children[c] = TrieNode(c)
            cur = cur.children[c]
        cur.word = word
```

8.1.2 Advanced

Implicit storage of word in TrieNode:

1. Implicitly stores the current word.
2. Implicitly stores the current char.
3. When insert new word, do not override the existing TrieNode. A flag to indicate whether there is a word ending here.

Code:

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


class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        cur = self.root
        for w in word:
            if w not in cur.children:   # not override
                cur.children[w] = TrieNode()
            cur = cur.children[w]

        cur.ended = True

    def search(self, word):
        cur = self.root
        for w in word:
            if w in cur.children:
                cur = cur.children[w]
            else:
                return False

        if not cur.ended:  # not ended here
            return False

        return True

    def startsWith(self, prefix):
        cur = self.root
        for w in prefix:
```

```
        if w in cur.children:
            cur = cur.children[w]
        else:
            return False

    return True
```

8.1.3 Application

1. Word search in matrix.
2. Word look up in dictionary.

Chapter 9
Sort

9.1 Introduction

List of general algorithms:

1. Selection sort: invariant
   a. Elements to the left of i (including i) are fixed and in ascending order (fixed and sorted).
   b. No element to the right of i is smaller than any entry to the left of i ($A[i] \leq \min(A[i+1:n])$).
2. Insertion sort: invariant
   a. Elements to the left of i (including i) are in ascending order (sorted).
   b. Elements to the right of i have not yet been seen.
3. Shell sort: h-sort using insertion sort.
4. Quick sort: invariant
   a. $|A_p|.. \leq ..|..unseen..|.. \geq ..|$ maintain the 3 subarrays.
5. Heap sort: compared to quick sort it is guaranteed $O(N \lg N)$, compared to merge sort it is $O(1)$ extra space.

9.2 Algorithms

9.2.1 Quick Sort

9.2.1.1 Normal pivoting

The key part of quick sort is pivoting:

```
def pivot(self, A, i, j):
    """
    pivoting algorithm:
    p | closed set | open set |
    | closed set p | open set |
    """
    p = i
    closed = p
    for ptr in xrange(i, j):
        if A[ptr] < A[p]:
            closed += 1
            A[ptr], A[closed] = A[closed], A[ptr]

    A[closed], A[p] = A[p], A[closed]
    return closed
```

Notice that this implementation goes $O(N^2)$ for arrays with all duplicates.

Problem with duplicate keys: it is important to stop scan at duplicate keys (counter-intuitive); otherwise quick sort will goes $O(N^2)$ for the array with all duplicate items, because the algorithm will put all items equal to the $A[p]$ on **a single side**.

Example: quadratic time to sort random arrays of 0s and 1s.

9.2.1.2 Stop-at-equal pivoting

Alternative pivoting implementation with optimization for duplicated keys:

```
def pivot_optimized(self, A, lo, hi):
    """
    Fix the pivot as the 1st element
    Scan from left to right and right to left simultaneously
    Avoid the case that the algo goes O(N^2) with duplicated keys
    """
    p = lo
    i = lo
    j = hi
    while True:
        while True:
            i += 1
            if i >= hi or A[i] >= A[lo]:
                break
        while True:
            j -= 1
            if j < lo or A[j] <= A[lo]:
                break

        if i >= j:
            break

        A[i], A[j] = A[j], A[i]

    A[lo], A[j] = A[j], A[lo]
    return j
```

9.2.1.3 3-way pivoting

3-way pivoting: pivot the array into 3 subarrays:
$|.. \leq ..|.. = ..|..unseen..|.. \geq ..|$

```
def pivot_3way(self, A, lo, hi):
    lt = lo-1  # pointing to end of array LT
    gt = hi  # pointing to the end of array GT (reversed)

    v = A[lo]
    i = lo  # scanning pointer
    while i < gt:
        if A[i] < v:
            lt += 1
            A[lt], A[i] = A[i], A[lt]
            i += 1
        elif A[i] > v:
            gt -= 1
            A[gt], A[i] = A[i], A[gt]
        else:
            i += 1

    return lt+1, gt
```

9.2.2 Stability

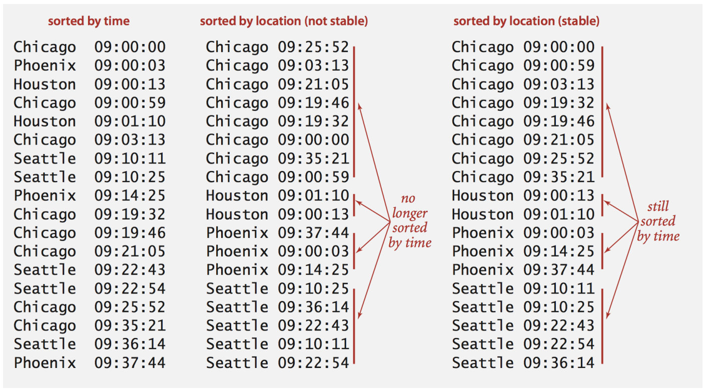
Definition: a stable sort preserves the **relative order of items with equal keys** (scenario: sorted by time then sorted by location).
Algorithms:

1. Stable
   a. Merge sort
   b. Insertion sort
2. Unstable
   a. Selection sort
   b. Shell sort
   c. Quick sort
   d. Heap sort

Long-distance swap operation is the key to find the unstable case during sorting.

9.2.3 Applications

1. Sort


no longer sorted by time
still sorted by time

sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

Fig. 9.1: Stale sort vs. unstable sort

2. Partial quick sort (selection), k-th largest elements
3. Binary search
4. Find duplicates
5. Graham scan
6. Data compression

9.2.4 Considerations

1. Stable?
2. Distinct keys?
3. Need guaranteed performance?
4. Linked list or arrays?
5. Caching system? (reference to neighboring cells in the array?
6. Usually randomly ordered array? (or partially sorted?)
7. Parallel?
8. Deterministic?
9. Multiple key types?

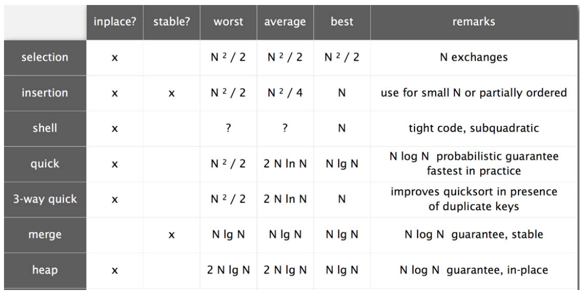
$O(N \lg N)$ is the lower bound of comparison-based sorting; but for other contexts, we may not need $O(N \lg N)$:

1. Partially-ordered arrays: insertion sort to achieve $O(N)$. **Number of inversions**: 1 inversion $=$ 1 pair of keys that are out of order.
2. Duplicate keys
3. Digital properties of keys: radix sort to achieve $O(N)$.

9.2.5 Summary

9.3 REVERSION

If $a_i > a_j$ but $i < j$, then this is considered as 1 reversion.



	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2/2$	$N^2/2$	$N^2/2$	N exchanges
insertion	x	x	$N^2/2$	$N^2/4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2/2$	2 N ln N	N lg N	N log N probabilistic guarantee fastest in practice
3-way quick	x		$N^2/2$	2 N ln N	N	improves quicksort in presence of duplicate keys
merge		x	N lg N	N lg N	N lg N	N log N guarantee, stable
heap	x		2 N lg N	2 N lg N	N lg N	N log N guarantee, in-place

Fig. 9.2: Sort summary

9.3.1 Calculation

MergeSort to calculate the #reverse-ordered paris. The only difference from a normal merge sort is that - when pushing the 2nd half of the array to the place, you calculate the reversion generated by the element $A_2[i_2]$ compared to $A_1[i_1:]$.

```
def merge(A1, n1, A2, n2, A, n):
    i = i1 = i2 =0
    count = 0
    while i < n:
        if i1 == n1:
            for i2 in xrange(i2, n2):
                A[i] = A2[i2]
                i += 1
        elif i2 == n2:
            for i1 in xrange(i1, n1):
                A[i] = A1[i1]
                i += 1
        else:
            # use array diagram to illustrate
            if A1[i1] > A2[i2]:  # push the A2 to A
                A[i] = A2[i2]
                # number of reverse-ordered pairs
                count += n1 - i1
                i += 1
                i2 += 1
            else:
                A[i] = A1[i1]
                i += 1
                i1 += 1

    return count

def merge_sort(a):
    n = len(a)
    if n == 1:
        return 0
    n1 = n/2
    n2 = n - n1

    a1 = a[:n1]
    a2 = a[n1:]

    count1 = merge_sort(a1)
    count2 = merge_sort(a2)
    count = count1+count2+merge(a1, n1, a2, n2, a, n)

    return count
```

Chapter 10
Search

10.1 Binary Search

Variants:

1. bisect_left
2. bisect_right
3. get the idx equal or just lower
4. get the idx equal or just higher

Binary search, get the idx equal or just lower, which is a very standard binary search:

```
def bisect(self, A, t):
    lo = 0
    hi = len(A)
    while lo < hi:
        mid = (lo+hi)/2
        if A[mid] == t:
            return mid
        elif A[mid] < t:
            lo = mid+1
        else:
            hi = mid
    return lo-1
```

10.2 Looping Root

Iterate the list and make the current element as the root, evaluate the left part and the right part and combine the results (i.e. looping + divide & conquer).

Chapter 11
Array

11.1 Circular Array

Common patterns for solving problems with circular arrays.

Normally, we should solve the linear problem and circular problem differently.

11.1.1 Circular max sum

Linear problem can be solved linear with simple algorithm, but the circular sum should use dp.

1. Construct left max sum for max sum over the $[0..i]$ (**forward** starting from the left side).
   Construct right max sum for max sum over the indexes $[i..n-1]$ (**backward** starting from the right side). Notice that the max sum index ends AT or BEFORE i.
2. The $maxSum = maxSum[i..n-1] + maxSum[0..i]$

11.1.2 Non-adjacent cell

To solve circular non-adjacent array problem in linear way, we should consider 2 cases:

1. Not consider the $A[1]$
2. Not consider the $A[-1]$

11.1.3 Binary search

Searching for an element in a circular sorted array. Half of the array is sorted while the other half is not.

1. If $A[0] < A[mid]$, then all values in the first half of the array are sorted.
2. If $A[mid] < A[-1]$, then all values in the second half of the array are sorted.
3. Then decide whether to got the **sorted half** or the **unsorted half**.

11.2 Voting Algorithm

11.2.1 Majority Number

11.2.1.1 $\frac{1}{2}$ of the Size

Given an array of integers, the majority number is the number that occurs more than half of the size of the array.

Algorithm: Majority Vote Algorithm. Maintain a counter to count how many times the majority number appear more than any other elements before index i and after re-initialization. Re-initialization happens when the counter drops to 0.

Proof: assuming there is a majority number x, if at the index i, the current count is j and the current counter does not capture the majority number, there are less than $\frac{i-j}{2}$ x, thus there are more than $\frac{n-i+j}{2}$ x after the index i. The j x beats against the counter and $\frac{n-i-j}{2}$ x will make it counted by counter.

If the counter captures the majority number, two cases will happen. The one is that the counter continue to capture the majority number till the end; then the counter will captures the correct majority number. The other case is that the majority number counter is beaten by other numbers, which will in turn fall back to the case that the counter does not capture the majority number.

This algorithm needs to re-check the current number being counted is indeed the majority number.

11.2.1.2 $\frac{1}{3}$ of the Size

Given an array of integers, the majority number is the number that occurs more than $\frac{1}{3}$ of the size of the array. This question can be generalized to be solved by $\frac{1}{k}$ case.

11.2.1.3 $\frac{1}{k}$ of the Size

Given an array of integers and a number k, the majority number is the number that occurs more than $\frac{1}{k}$ of the size of the array.

```
class Solution:
```

```
def majorityNumber(self, nums, k):
    """
    Since majority elements appears more
    than ceil(n/k) times, there are at
    most 2 majority number
    """
    cnt = defaultdict(int)
    for num in nums:
        if num in cnt:
            cnt[num] += 1
        else:
            if len(cnt) < k-1:
                cnt[num] += 1
            else:
                for key in cnt.keys():
                    cnt[key] -= 1
                    if cnt[key] == 0:
                        del cnt[key]

    for key in cnt.keys():
        if (len(filter(lambda x: x == key, nums))
            > len(nums)/k):
            return key

    raise Exception
```

Chapter 12
Stream

12.1 SLIDING WINDOW

Sliding Window Maximum. Given an array *nums*, Find the list of maximum in the sliding window of size *k* which is moving from the very left of the array to the very right. $\rightarrow$ double-ended queue.

Invariant: the queue is storing the non-decreasing-ordered elements of current window.

Sliding Window Median. Find the list of median in the sliding window. $\rightarrow$ Dual heap with lazy deletion.

Chapter 13
Math

13.1 Functions

Equals. Requirements for equals

1. Reflexive
2. Symmetric
3. Transitive
4. Non-null

Compare. Requirements for compares (total order):

1. Antisymmetry
2. Transitivity
3. Totality

13.2 Prime Numbers

13.2.1 Sieve of Eratosthenes

13.2.1.1 Basics

To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Starting from p, enumerate its multiples by counting to n in increments of p, and mark them in the list (these will be $2p$, $3p$, $4p$, ... ; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.

When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.

13.2.1.2 Refinements

The main idea here is that every value for p is prime, because we have already marked all the multiples of the numbers less than p. Note that some of the numbers being marked may have already been marked earlier (e.g., 15 will be marked both for 3 and 5).

As a refinement, it is sufficient to mark the numbers in step 3 starting from p^2, because all the smaller multiples of p will have already been marked at that point by the previous smaller prime factor other than p. From p^2, p becomes the smaller prime factor of a composite number. This means that the algorithm is allowed to terminate in step 4 when p^2 is greater than n.

Another refinement is to initially list odd numbers only, (3, 5, ..., n), and count in increments of 2p in step 3, thus marking only odd multiples of p. This actually appears in the original algorithm. This can be generalized with wheel factorization, forming the initial list only from numbers coprime with the first few primes and not just from odds (i.e., numbers coprime with 2), and counting in the correspondingly adjusted increments so that only such multiples of p are generated that are coprime with those small primes, in the first place.

To summarized, the refinements include:

1. Starting from p^2.
2. Preprocessing even numbers and then only process odd numbers; thus the increment becomes $2p$.

13.2.1.3 code

```
def countPrimes(n):
    """
    Find primeusing Sieve's algorithm
    :type n: int
    :rtype: int
    """
    if n < 3:
        return 0

    is_prime = [True for _ in xrange(n)]
    is_prime[0], is_prime[1] = False, False
    for i in xrange(2, int(math.sqrt(n))+1):
        if is_prime[i]:
            for j in xrange(i*i, n, i):
                is_prime[j] = False

    return is_prime.count(True)
```

13.2.2 Factorization

Backtracking:

```
def dfs(self, cur, ret):
    """
    16

    get factors of cur[-1]
    [16]
    [2, 8]
    [2, 2, 4]
    [2, 2, 2, 2]

    [4, 4]
    """
    if len(cur) > 1:
        ret.append(list(cur))

    n = cur.pop()
    start = cur[-1] if cur else 2
    for i in xrange(start, int(sqrt(n))+1):
        if n%i == 0:
            cur.append(i)
            cur.append(n/i)
            self.dfs(cur, ret)
            cur.pop()
```

13.2.2.1 Time Complexity

$O(2^n)$ where n is the number of prime factors. Choose i prime factors to combine then, and keep the rest uncombined

$$\sum_i \binom{n}{i}$$

TODO

Chapter 14
Arithmetic

14.1 DFS

Insert operators. Given a string that contains only digits 0-9 and a target value, return all possibilities to add binary operators (not unary) +, -, or * between the digits so they evaluate to the target value.

Example:

$$"123", 6 \rightarrow ["1+2+3", "1*2*3"]$$
$$"232", 8 \rightarrow ["2*3+2", "2+3*2"]$$

Clues:

1. DFS
2. Special handling for multiplication - caching
3. Detect invalid number with leading 0's

Code:

```
def addOperators(self, num, target):
  ret = []
  self.dfs(num, target, 0, "", 0, 0, ret)
  return ret

def dfs(self, num, target, pos,
        cur_str, cur_val,
        mul, ret
    ):
  if pos >= len(num):
    if cur_val == target:
      ret.append(cur_str)
  else:
    for i in xrange(pos, len(num)):
      if i != pos and num[pos] == '0':
        continue
      nxt_val = int(num[pos:i+1])

      if not cur_str:
        self.dfs(num, target, i+1,
            "%d"%nxt_val, nxt_val,
            nxt_val, ret)
      else:
        self.dfs(num, target, i+1,
            cur_str+"+%d"%nxt_val, cur_val+nxt_val,
            nxt_val, ret)
        self.dfs(num, target, i+1,
            cur_str+"-%d"%nxt_val, cur_val-nxt_val,
            -nxt_val, ret)
        self.dfs(num, target, i+1,
            cur_str+"*%d"%nxt_val, cur_val-mul+mul*nxt_val,
            mul*nxt_val, ret)
```

Insert parenthesis. Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are +, - and *.

Examples:

$$(2*(3-(4*5))) = -34$$
$$((2*3)-(4*5)) = -14$$
$$((2*(3-4))*5) = -10$$
$$(2*((3-4)*5)) = -10$$
$$(((2*3)-4)*5) = 10$$

Clues: Iterate the operators, divide and conquer - left parts and right parts and then combine result.
Code:

```
def dfs_eval(self, nums, ops):
    ret = []
    if not ops:
        assert len(nums) == 1
        return nums

    for i, op in enumerate(ops):
        left_vals = self.dfs_eval(nums[:i+1], ops[:i])
        right_vals = self.dfs_eval(nums[i+1:], ops[i+1:])
        for l in left_vals:
            for r in right_vals:
                ret.append(self._eval(l, r, op))

    return ret
```

14.2 Polish Notation

Polish Notation is in-fix while Reverse Polish Notation is post-fix.

14.2.1 Evaluate Post-fix Expressions

Straightforward: Use a stack to store the number. Iterate the input, push stack when hit numbers, pop stack when hit operators.

14.2.2 Convert In-fix to Post-fix

TODO

Chapter 15
Combinatorics

15.1 Basics

15.1.1 Considerations

1. Does **order** matter?
2. Are the objects **repeatable**?
3. Are the objects partially **duplicated**?

If order does not matter, you can pre-set the order.

15.1.2 Basic Formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$
$$\binom{n}{k} = \binom{n}{n-k}$$
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

15.1.3 N objects, K Ceils

$$x_1 + x_2 + x_3 = 10$$

is equivalent to

$$*****|**|***$$

, notice that $*$ are duplicated.
then the formula is:

$$\binom{n+r}{r}$$

,where $r = k-1$.
The meaning is to choose r objects from $n+r$ objects to become the $|$.

15.1.4 N objects, K types

What is the number of permutation of N objects with K different types:

$$ret = \frac{A_N^N}{\prod_{i=1}^{K} A_{sz(i)}^{sz(i)}}$$

15.1.5 Inclusion–Exclusion Principle

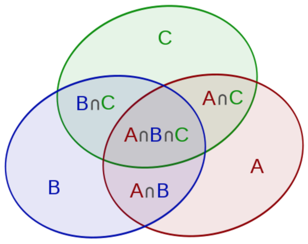


Fig. 15.1: Inclusion–exclusion principl

$$\begin{aligned} |A\cup B\cup C| &= |A| + |B| + |C| \\ &-|A\cap B| - |A\cap C| - |B\cap C| \\ &+|A\cap B\cap C| \end{aligned}$$

Generally,

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{k=1}^{n} (-1)^{k+1} \left(\sum_{1\le i_1<\cdots<i_k\le n} |A_{i_1}\cap\cdots\cap A_{i_k}| \right)$$

15.2 Combinations with Duplicated Objects

Determine the number of combinations of 10 letters (order does not matter) that can be formed from 3A, 4B, 5C.

15.2.1 Basic Solution

If there are no restrictions on the number of any of the letter, it is $\binom{10+2}{2}$; then we get the universal set,

$$|U| = \binom{10+2}{2}$$

Let P_A be the set that a 10-combination has more than 3A. P_B...4B. P_C...5C.

The result is:

$$|3A \cap 4B \cap 5C| =$$
$$|U| - sum(|P_i|) + sum(|P_i \cap P_j|) - sum(|P_i \cap P_j \cap P_k|)$$

To calculate $|P_i|$, take $|P_1|$ as an example. **Pre-set** 4A – if we take any one of these 10-combinations in P_1 and remove 4A we are left with a 6-combination with unlimited on the numbers of letters; thus,

$$|P_1| = \binom{6+2}{2}$$

Similarly, we can get P_2, P_3.

To calculate $|P_i \cap P_j|$, take $|P_1 \cap P_2|$ as an example. **Pre-set** 4A and 5B; thus,

$$|P_1 \cap P_2| = \binom{1+2}{2}$$

Similarly, we can get other $|P_i \cap P_j|$.

Similarly, we can get other $|P_i \cap P_j \cap P_k|$.

15.2.2 Algebra Solution

The number of 10-combinations that can be made from 3A, 4B, 5C is found from the coefficient of x^{10} in the expansion of:

$$(1+x+x^2+x^3)(1+x+x^2+x^3+x^4)(1+x+x^2+x^3+x^4+x^5)$$

And we know:

$$\begin{aligned}1+x+x^2+x^3 &= (1-x^4)/(1-x)\\ 1+x+x^2+x^3+x^4 &= (1-x^5)/(1-x)\\ 1+x+x^2+x^3+x^4+x^5 &= (1-x^6)/(1-x)\end{aligned}$$

We expand the formula, although the naive way of getting the coefficient of x^{10} is tedious.

Chapter 16
Bit Manipulation

16.1 Concepts

16.1.1 Basics

1. Bit value: bit0, bit1.
2. BitSet/Bits
3. Bit position (bit interchangeably)

16.1.2 Operations

Mask.

1. Masking to 1: to mask a single bit position, $bit \vee 1$
2. Masking to 0: to mask a single bit position, $bit \wedge 0$
3. Querying a bit position value: to query a single bit position, $bit \wedge 1$
4. Toggling bit values: to toggle a single bit position, $bit \oplus 1$

This can be extended to do masking operations on multiple bits.

Rightmost bit set. To get the rightmost bit, with the help of 2's complement:

$$rightmost = bits \wedge -bits$$

16.2 Single Number

16.2.1 Appear three times

Given an array of integers, every element appears three times except for one. Find that single one.

Using list. Consider 4-bit numbers:

```
0000
0001
0010
...
1111
```

Add (not $\wedge$) the bit values **vertically**, then result would be *abcd* where a, b, c, d can be any number, not just binary. a, b, c, d can be divided by 3 if the all element appears three times. Until here, you can use a list to hold a, b, c, d. By mod 3, the single one that does not appear 3 times is found.

To generalize to 32-bit `int`, use a list of length 32.

Using bits. To further optimize the space, use bits (bit set) instead of list.

- Since all except one appears 3 times, we are only interested in $0, 1, 2$ (mod 3) count of bit1 appearances in a bit position.
- We create 3 bit sets to represent $0, 1, 2$ appearances of all positions of bits.
- For a bit, there is one and only one bit set containing bit1 in that bit position.
- Transition among the 3 bit sets for every number:

  $$bitSet^{(i)} = (bitSet^{(i-1)} \wedge num) \vee (bitSet^{(i)} \wedge \neg num)$$

  For i appearances, the first part is the bit set **transited from** $(i-1)$ appearances, and the second part is the bit set **transited out** from itself.

16.2.2 Two Numbers

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

- Easily get: $x = a \oplus b$.
- $a \neq b$; thus there are at least one 1-bit in x is different.
- Take an arbitrary 1 bit set in x, and such bit set can classify the elements in the array into two separate groups.

Chapter 17
Greedy

17.1 INTRODUCTION

Queue, Stack

17.1.1 Summarizing properties

TODO

Chapter 18
String

18.1 Palindrome

18.1.1 Palindrome anagram

Test palindrome anagram. Char counter, number of odd count should ≤ 0.

Count palindrome anagram. 15.1.4.

Construct palindrome anagram. clues:

1. dfs
2. jump parent char

Code:

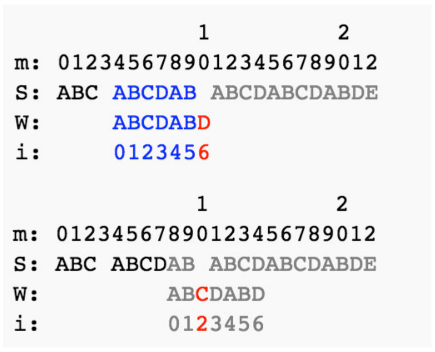
```
def grow(self, s, count_map, pi, cur, ret):
  if len(cur) == len(s):
    ret.append(cur)
    return

  for k in count_map.keys():
    if k != pi and count_map[k] > 0:
      # jump the parent
      for i in xrange(1, count_map[k]/2+1):
        count_map[k] -= i*2
        self.grow(s, count_map, k, k*i+cur+k*i, ret)
        count_map[k] += i*2
```

18.2 KMP

Find string W in string S within complexity of $O(|W|+|S|)$.

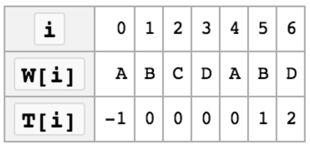
18.2.1 Prefix suffix table

Partial match table (also known as "failure function"). After a failure matching, you know that the matched suffix before the failure point is already matched; therefore when you shift the W, you only need to shift the prefix



```
                  1         2
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:     ABCDABD
i:     0123456

                  1         2
m: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
W:         ABCDABD
i:         0123456
```

Fig. 18.1: KMP example



i	0	1	2	3	4	5	6
W[i]	A	B	C	D	A	B	D
T[i]	-1	0	0	0	0	1	2

Fig. 18.2: Prefix-suffix table

onto the position of the previous suffix. The prefix and suffix must be proper prefix and suffix.

In table-building algorithm. $T[i]$ stores the NEXT prefix index *cnd*.

Notice:

1. dummy at $T[0] = -1$.
2. three parts
   a. matched
   b. fall back
   c. restart

Code:

```
# construct T
T = [0 for _ in xrange(ln+1)]
T[0] = -1
T[1] = 0
pos = 2
cnd = 0
while pos <= ln:
    if needle[pos-1]==needle[cnd]:  # matched
        T[pos] = cnd+1
        cnd += 1
        pos += 1
    elif T[cnd]!=-1:  # fall back
        cnd = T[cnd]
    else:  # restart
        T[pos] = 0
        cnd = 0
        pos += 1
```

18.2.2 Searching algorithm

Notice:

1. index *i* and *s*.
2. $T[i+1-1]$ for corresponding previous index in T for current scanning index *i*.
3. three parts:
    a. matched
    b. fall back
    c. restart

Code:

```
# search
i = 0  # index for needle
j = 0  # index for haystack
while j+i < len(haystack):
    if needle[i] == haystack[j+i]:  # matched
        i += 1
        if i == len(needle):
            return haystack[j:]
    else:
        if T[i] != -1:  # fall back
            j = j+i-T[i]
            i = T[i]
        else:  # restart
            j += 1
            i = 0

return None
```

18.2.3 Applications

1. Find needle in haystack.
2. Shortest palindrome

Chapter 19
Graph

19.1 Basic

Graph Representation. V for a vertex set with a map, mapping from vertex to its neighbors.

```
V = defaultdict(list)
```

19.2 BFS

19.2.1 BFS with Abstract Level

Start BFS with a set of vertices in abstract level, not necessarily neighboring vertices.

Example: -1 obstacles, 0 targets, calculate all other vertices's Manhattan distance to its nearest target:

$$\begin{bmatrix} \infty & -1 & 0 & \infty \\ \infty & \infty & \infty & -1 \\ \infty & -1 & \infty & -1 \\ 0 & -1 & \infty & \infty \end{bmatrix}$$

is calculated as:

$$\begin{bmatrix} 3 & -1 & 0 & 1 \\ 2 & 2 & 1 & -1 \\ 1 & -1 & 2 & -1 \\ 0 & -1 & 3 & 4 \end{bmatrix}$$

Code

```
def wallsAndGates(self, mat):
    q = [(i, j) for i, row in enumerate(mat)
           for j, val in enumerate(row) if val == 0]
    for i, j in q:  # iterator
        for d in self.dirs:
            I, J = i+d[0], j+d[1]
            if (0 <= I < m and  0 <= J < n and
                mat[I][J] > mat[i][j]+1):
                mat[I][J] = mat[i][j]+1
                q.append((I, J))
```

19.3 Detect Acyclic

1. `marked` is reset after a dfs.
2. `visited` should be maintained only in the end of the dfs.
3. For directed graph:
   a. Should dfs for all neighbors except for vertices in `visited`, to avoid revisiting. For example, avoid revisiting A, B when start from C in the graph $C \rightarrow A \rightarrow B$.
   b. Excluding predecessor `pi` is erroneous in the case of $A \leftrightarrow B$
4. For undirected graph:
   a. Should dfs for all neighbors except for the predecessor `pi`.
   b. Excluding neighbors in `visited` is redundant.

19.3.1 Directed Graph

```
def dfs(self, V, k, visited, pathset):
    if k in pathset:
        return False

    marked.add(k)
    for nbr in V[k]:
        if nbr not in visited:
            if not self.dfs(V, nbr, visited, pathset):
                return False

    marked.remove(k)
    pathset.add(k)
    return True
```

19.3.2 Undirected Graph

```
def dfs(self, V, k, pi, visited, marked):
    if k in marked:
        return False

    marked.add(k)
    for neighbor in V[k]:
        if neighbor != pi:
            if not self.dfs(V, neighbor, k,
                visited, marked):
                return False

    marked.remove(k)
    visited.add(k)
    return True
```

19.4 Topological Sorting

For a graph $G = \{V, E\}$, $A \to B$, then A is before B in the ordered list.

19.4.1 Algorithm

General algorithm and notice.

dfs:

1. **Dfs neighbors first**. If the neighbors of current node is $\neg$visited, then dfs the neighbors
2. **Dfs current node**. After visiting all the neighbors, then visit the current node and push it to the result queue.
3. Reverse. Reverse the result queue.

Notice:

1. Need to **reverse** the result queue, since the neighbors (successors) are visited first.
2. Need to **detect cycle**; thus the dfs need to construct result queue and detect cycle simultaneously, by using two sets: *visited* and *marked*.

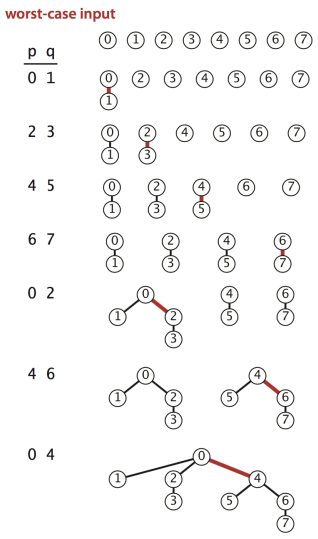


Fig. 19.1: Weighted quick-union traces

```
def topological_sort(self, V):
    visited = set()
    marked = set()
    ret = []

    for k in V.keys():
        if k not in visited:
            if not self.dfs(V, k, visited, marked, ret):
                return []  # contains cycle

    ret.reverse()
    return ret

def dfs(self, V, k, visited, marked, ret):
    if k in marked:
        return False

    marked.add(k)
    for neighbor in V[k]:
        if neighbor not in visited:
            if not self.dfs(V, neighbor, visited, marked, ret):
                return False

    marked.remove(k)
    visited.add(k)
    ret.append(k)
    return True
```

19.4.2 Applications

1. Course scheduling problem with pre-requisite.

19.5 Union-Find

Improvements:

1. Weighting: size-baladnced tree
2. Path Compression.

19.5.1 Algorithm

Weighted union-find with path compression:

1. An array to store each item's predecessor **pi**.
2. Merge the tree according to the **size** to maintain balance.

```
class UnionFind(object):
    """
    Weighted Union Find with path compression
    """
    def __init__(self):
        self.pi = {}  # item -> pi
        self.sz = {}  # root -> size

    def add(self, item):
        if item not in self.pi:
            self.pi[item] = item
            self.sz[item] = 1

    def union(self, a, b):
        pi1 = self.root(a)
        pi2 = self.root(b)

        if pi1 != pi2:
            if self.sz[pi1] > self.sz[pi2]:
                pi1, pi2 = pi2, pi1
                # size balancing

            self.pi[pi1] = pi2
            self.sz[pi2] += self.sz[pi1]
            del self.sz[pi1]

    def root(self, item):
        pi = self.pi[item]
        if item != pi:
            self.pi[item] = self.root(pi)
            # path compression

        return self.pi[item]

    def count(self):
        return len(self.sz)  # only root nodes have size
```

19.5.2 Complexity

m union-find with n objects: $O(n) + mO(\lg n)$

Chapter 20
Interval

20.1 Introduction

Two-way range. The current scanning node as the pivot, need to scan its left neighbors and right neighbors.

$$|\leftarrow p \rightarrow|$$

If the relationship between the pivot and its neighbors is symmetric, since scanning range is $[i-k, i+k]$ and iterating from left to right, only consider $[i-k, i]$ to avoid duplication.

$$|\leftarrow p$$

Chapter 21
Dynamic Programming

21.1 Introduction

The core philosophy of dp:

1. The definition of **states**
2. The definition of the **transition functions** among states

The so called concept dp as memoization of recursion does not grasp the core philosophy of dp.

The formula in the following section are unimportant. Instead, what is important is the definition of dp array and transition function derivation.

21.1.1 Common practice

Dummy. Use dummies to avoid using if-else conditional branch.

1. Use $n+1$ dp arrays to reserve space for dummies.
2. Iteration range is $[1, n+1)$.
3. $n+k$ for k dummies

State definition. Two general sets of state definitions:

1. End AT index i
2. End BEFORE index i

Space optimization. To avoid MLE, we need to carry out space optimization. Let o be other subscripts, f be the transition function.

Firstly,

$$F_{i,o} = f\big(F_{i-1,o'}\big)$$

should be reduced to

$$F_o = f\big(F_{o'}\big)$$

Secondly,

$$F_{i,o} = f\big(F_{i-1,o'}, F_{i-2.o'}\big)$$

should be reduced to

$$F_{i,o} = f\big(F_{(i-1)\%2,o'}, F_{(i-2)\%2.o'}\big)$$

More generally, we can be $(i-b)\%a$ to reduce the space down to a.

Notice:

1. Must iterate o **backward** to un-updated value.

21.2 Sequence

Longest common subsequence. Let $F_{i,j}$ be the LCS at string $a[:i]$ and $b[:j]$. We have two situations: $a[i] == b[j]$ or not.

$$F_{i.j} = \begin{cases} F_{i-1,j-1} + 1 \text{ // if } a[i] == b[j] \\ \max\Big(F_{i-1,j}, \text{ // otherwise} \\ \quad F_{i,j-1}\Big) \end{cases}$$

Longest common substring. Let $F_{i,j}$ be the LCS at string $a[:i]$ and $b[:j]$. We have two situations: $a[i] == b[j]$ or not.

$$F_{i.j} = \begin{cases} F_{i-1,j-1} + 1 \text{ // if } a[i] == b[j] \\ 0 \text{ // otherwise} \end{cases}$$

Because it is not necessary that $F_{i,j} \geq F_{i',j'}, \forall i,j \cdot i > i', j > j'$, the $gmax = max\big(\{F_{i,j}\}\big)$.

Edit distance Find the minimum number of steps required to convert words A to B using inserting, deleting, replacing.

Let $F_{i,j}$ be the minimum number of steps required to convert $A[:i]$ to $B[:j]$.

$$F_{i.j} = \begin{cases} F_{i-1,j-1} \text{ // if } a[i] == b[j] \\ \min\Big(F_{i,j-1} + 1, \text{ //otherwise, insert} \\ \quad F_{i-1,j} + 1, \text{ // delete} \\ \quad F_{i_1,j_1} + 1\Big) \text{ // replace} \end{cases}$$

Maximal square. Find the largest rectangle in the matrix:

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

Let $F_{i,j}$ represents the max square's length ended at $mat_{i,j}$ (lower right corner).

$$F_{i.j} = \begin{cases} \min\big(F_{i-1,j-1}, F_{i-1,j}, F_{i,j-1}\big) + 1 \text{ // if } mat_{i,j} == 1 \\ 0 \text{ // otherwise} \end{cases}$$

21.3 BACKPACK

Given n items with weight w_i and value v_i, an integer C denotes the size of a backpack. What is the max value you can fill this backpack?

Let $F_{i,c}$ be the max value we can carry for index $0..i$ with capacity c. We have 2 choices: take the i-th item or not.

$$F_{i,c} = \max\big(F_{i-1,c}, \\ F_{i-1,c-w_i} + v_i\big)$$

TODO advanced backpack problem.

21.4 LOCAL AND GLOBAL EXTREMES

21.4.1 Long and short stocks

The following formula derives from the question: Best Time to Buy and Sell Stock IV.

Let $local_{i,j}$ be the max profit at day i with j transactions with last transactions ENDED at day i. Let $global_{i,j}$ be the max profit at day i with j transactions.

$$\begin{aligned} local_{i,j} &= \max\Big(global_{i-1.j-1} + \Delta, local_{i-1,j} + \Delta\Big) \\ global_{i,j} &= \max\Big(local_{i,j}, global_{i-1,j}\Big) \end{aligned} \tag{21.1}$$

Notice:

1. Consider opportunity costs.
2. The global min is not $glocal[-1]$ but $\max\big(\{global[i]\}\big)$.
3. You must sell the stock before you buy again (i.e. you can not have higher than 1 in stock position).

21.4.1.1 Space optimization

$$\begin{aligned} local_j &= \max\Big(global_{j-1} + \Delta, local_j + \Delta\Big) \\ global_j &= \max\Big(local_j, global_j\Big) \end{aligned} \tag{21.2}$$

Notice,

1. Must iterate j **backward**; otherwise we will use the updated value.

21.4.1.2 Alternative definitions

Other possible definitions: let $global_{i,j}$ be the max profit at day i with UP TO j transactions. Then,

$$\begin{aligned} local_{i,j} &= \max\Big(global_{i-1.j-1} + \max(0,\Delta), local_{i-1,j} + \Delta\Big) \\ global_{i,j} &= \max\Big(local_{i,j}, global_{i-1,j}\Big) \end{aligned} \tag{21.3}$$

and $global[-1]$ is the global max.

21.5 GAME THEORY - MULTI PLAYERS

Assumption: the opponent take the optimal strategy for herself.

21.5.1 Coin game

Same side There are n coins with different value in a line. Two players take turns to take 1 or 2 coins from left side. The player who take the coins with the most value wins.

let F_i^p represents maximum values he can get for index $i..last$, for the person p. There are 2 choices: take the i-th coin or take the i-th and $(i+1)$-th coin.

$$F_i^p = \max\big(A_i + S[i+1:] - F_{i+1}^{p'}, \\ A_i + A_{i+1} + S[i+2:] - F_{i+2}^{p'}\big)$$

The above equation can be further optimized by merging the sum S.

Dual sides There are n coins in a line. Two players take turns to take a coin from one of the ends of the line until there are no more coins left. The player with the larger amount of money wins.

let $F_{i,j}^p$ represents maximum values he can get for index $i..j$, for the person p. There are 2 choices: take the i-th coin or take the j-th coin.

$$F_{i,j}^p = \max\big(A_i + S[i+1:j] - F_{i+1,j}^{p'}, \\ A_j + S[i:j-1] - F_{i,j-1}^{p'}\big)$$

Chapter 22
General

22.1 General Tips

Information Source. Keep the source information rather than derived information (e.g. keep the array index rather than array element).

Information Transformation. Need you keep the raw information to avoid information loss (e.g. after converting `str` to `list`, you should keep `str`).

Element Data Structure When working with ADT, you should use a more intelligence data structure as type to avoid allocating another ADT to maintain the state (e.g. `java.util.PriorityQueue<E>`).

Glossary

A Array

idx Index

TLE Time Limit Exceeded

MLE Memory Limit Exceeded

dp Dynamic programming

in-place The algorithm takes $\leq c \lg N$ extra space

partially sorted number of inversion in the array $\leq cN$

non-degeneracy Distinct properties without total overlapping

underflow Degenerated, empty, or null case

loitering Holding a reference to an object when it is no longer needed thus hindering garbage collection.