

github.com/algorithmshms/Algo-Quicksheet

Author: idf@github

Algorithm Quicksheet

Classical equations, diagrams and tricks in algorithm

November 11, 2015

©2015 github.com/idf

Except where otherwise noted, this document is licensed under a BSD 3.0 license (opensource.org/licenses/BSD-3-Clause).

This book is dedicated to all Software Engineers.

Preface

INTRODUCTION

This quicksheet contains many classical equations and diagrams for algorithm, which helps 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 are unable to reconstruct the algorithm from the hints and the important clues.

At GitHub, June 2015

github.com/idf

Contents

Contents	v	6 Tree	10
Notations	ix	6.1 Binary Tree	10
1 Time Complexity	1	6.1.1 Introductions	10
1.1 Basic Counts	1	6.1.2 Morris Traversal	10
1.2 Solving Recurrence Equations	1	6.2 Binary Search Tree (BST)	11
1.2.1 Master Theorem	1	6.2.1 Rank	11
1.3 Useful Math Equations	1	6.2.2 Range search	12
2 Memory Complexity	3	6.3 Binary Index Tree (BIT)	12
2.1 Introduction	3	6.3.1 Introduction	12
2.1.1 Memory for Data Type	3	6.3.2 Implementation	12
2.1.2 Example	3	6.4 Segment Tree	13
3 Basic Data Structures	4	6.4.1 Introduction	13
3.1 Introduction	4	6.4.2 Operations	14
3.2 Stack	4	6.5 Trie	15
3.2.1 Stack and Recursion	4	6.5.1 Basic	15
3.2.2 Usage	4	6.5.2 Advanced	15
3.2.3 Applications	4	6.5.3 Applications	15
3.2.4 All nearest smaller values	5	7 Balanced Search Tree	16
3.3 Map	5	7.1 2-3 Search Tree	16
3.3.1 Math relations	5	7.1.1 Insertion	16
3.3.2 Operations	5	7.1.2 Splitting	16
4 Linked List	6	7.1.3 Properties	16
4.1 Operations	6	7.2 Red-Black Tree	17
4.1.1 Fundamentals	6	7.2.1 Properties	17
4.1.2 Basic Operations	6	7.2.2 Operations	17
4.1.3 Combined Operations	6	7.3 B-Tree	18
4.2 Combinations	6	7.3.1 Basics	18
4.2.1 LRU	6	7.3.2 Operations	18
5 Heap	8	7.4 AVL Tree	18
5.1 Introduction	8	7.5 Cartesian Tree	19
5.2 Operations	8	7.5.1 Basics	19
5.2.1 Sink (sift_down)	8	7.5.2 Treap	19
5.2.2 Swim (sift_up)	8	8 Sort	20
5.2.3 Heapify	8	8.1 Introduction	20
5.3 Implementation	8	8.2 Algorithms	20
5.3.1 General	8	8.2.1 Quick Sort	20
5.3.2 Python Heapq	9	8.2.2 Merge Sort	21
5.3.3 Java Priority Queue	9	8.3 Properties	21
		8.3.1 Stability	21
		8.3.2 Sort Applications	21
		8.3.3 Considerations	21
		8.3.4 Summary	21
		8.4 Partial Quicksort	22
		8.4.1 Find m smallest	22
		8.4.2 Find k -th	22

8.5	Inversion	22	14	Arithmetic	35
8.5.1	MergeSort & Inversion Pair	22	14.1	Big Number	35
8.5.2	Binary Index Tree & Inversion Count	22	14.2	Polish Notations	35
8.5.3	Segment Tree & Inversion Count	23	14.2.1	Convert in-fix to post-fix (RPN)	35
8.5.4	Reconstruct Array from Inversion Count	23	14.2.2	Evaluate post-fix expressions	35
9	Search	25	14.2.3	Convert in-fix to pre-fix (PN)	35
9.1	Binary Search	25	14.2.4	Evaluate pre-fix (PN) expressions	36
9.1.1	idx equal or just lower	25	15	Combinatorics	37
9.1.2	idx equal or just higher	25	15.1	Basics	37
9.1.3	bisect_left	25	15.1.1	Considerations	37
9.1.4	bisect_right	25	15.1.2	Basic formula	37
9.2	Applications	26	15.1.3	N objects, K ceils	37
9.2.1	Rotation	26	15.1.4	N objects, K types	37
9.3	Combinations	26	15.1.5	Inclusion–Exclusion Principle	37
9.3.1	Extreme-value problems	26	15.2	Combinations with Duplicated Objects	38
9.4	High dimensional search	26	15.2.1	Basic Solution	38
9.4.1	2D	26	15.2.2	Algebra Solution	38
10	Array	28	15.3	Permutation	38
10.1	Circular Array	28	15.3.1	k-th permutation	38
10.1.1	Circular max sum	28	15.4	Catalan Number	39
10.1.2	Non-adjacent cell	28	15.4.1	Math	39
10.1.3	Binary search	28	15.4.2	Applications	39
10.2	Voting Algorithm	28	15.5	Stirling Number	39
10.2.1	Majority Number	28	16	Probability	40
10.3	Two Pointers	29	16.1	Shuffle	40
10.3.1	Interleaving	29	16.1.1	Incorrect naive solution	40
11	String	30	16.1.2	Knuth Shuffle	40
11.1	Palindrome	30	16.2	Expected Value	40
11.1.1	Palindrome anagram	30	16.2.1	Roll dice until expected value	40
11.2	KMP	30	17	Bit Manipulation	41
11.2.1	Prefix suffix table	30	17.1	Concepts	41
11.2.2	Searching algorithm	30	17.1.1	Basics	41
11.2.3	Applications	31	17.1.2	Operations	41
12	Stream	32	17.2	Single Number	41
12.1	Sliding Window	32	17.2.1	Three-time appearance	41
13	Math	33	17.2.2	Two Numbers	42
13.1	Functions	33	17.3	Bitwise operators	42
13.2	Prime Numbers	33	18	Greedy	43
13.2.1	Sieve of Eratosthenes	33	18.1	Introduction	43
13.2.2	Factorization	33	19	Backtracking	44
13.3	Median	34	19.1	Introduction	44
13.3.1	Basic DualHeap	34	19.2	Sequence	44
13.3.2	DualHeap with Lazy Deletion	34	19.3	String	44
			19.3.1	Palindrome	44
			19.4	Math	45
			19.4.1	Decomposition	45
			19.5	Arithmetic Expression	45
			19.5.1	Unidirection	45
			19.5.2	Bidirection	45

19.6	Tree	46	21.2.2	Dual-state dp	52
19.6.1	BST	46	21.3	String	52
20	Graph	47	21.4	Combinatorics	53
20.1	Basic	47	21.4.1	Tree	53
20.2	DFS	47	21.5	Backpack	54
20.3	BFS	47	21.5.1	Classical	54
20.3.1	BFS with Abstract Level	47	21.5.2	Sum	54
20.4	Detect Acyclic	48	21.6	Local and Global Extremes	54
20.4.1	Directed Graph	48	21.6.1	Long and short stocks	54
20.4.2	Undirected Graph	48	21.7	Game theory - multi players	55
20.5	Topological Sorting	48	21.7.1	Coin game	55
20.5.1	Algorithm	48	22	Interval	56
20.5.2	Applications	49	22.1	Introduction	56
20.6	Union-Find	49	22.2	Operations	56
20.6.1	Algorithm	49	22.3	Event-driven algorithm	56
20.6.2	Complexity	50	22.3.1	Introduction	56
20.7	Axis Projection	50	22.3.2	Questions	56
21	Dynamic Programming	51	23	General	58
21.1	Introduction	51	23.1	General Tips	58
21.1.1	Common practice	51	Glossary		59
21.2	Sequence	51	Abbreviations		60
21.2.1	Single-state dp	51			

List of Contributors

Daniel D. Zhang (github.com/idf)

Notations

GENERAL MATH NOTATIONS

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
$\text{floor}(\text{key})$	the largest key \leq the given key
$\text{ceil}(\text{key})$	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
$a \& b$	Bit AND
$a b$	Bit OR
$a \wedge a$	Bit XOR
$\sim a$	Bit NOT
$\ll a$	Bit shift left
$\gg a$	Bit shift right
∞	Infinity
\rightarrow	Tends towards, e.g., $n \rightarrow \infty$
\propto	Proportional to; $y = ax$ can be written as $y \propto x$
$ x $	Absolute value
$\ \mathbf{a}\ $	L_2 distance (Euclidean distance) of a vector; norm-2
$ \mathcal{S} $	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): $0 : n = 0, 1, 2, \dots, n-1$
\approx	Approximately equal to
\sim	Tilde, the leading term of mathematical expressions
$\arg \max_x f(x)$	Argmax: the value x that maximizes f
$\binom{n}{k}$	n choose k , equal to $\frac{n!}{k!(n-k)!}$
$\text{range}(i, j)$	range of number from i (inclusive) to j (exclusive)
$A[i : j]$	subarray consist of $A_i, A_{i+1}, \dots, A_{j-1}$.

Chapter 1

Time Complexity

1.1 BASIC COUNTS

Double for-loops

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

$$\sum_{i=1}^N \sum_{j=i}^N 1 \sim \int_{x=1}^N \int_{y=x}^N dy dx$$

Triple for-loops

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

$$\sum_{i=1}^N \sum_{j=i}^N \sum_{k=1}^N 1 \sim \int_{x=1}^N \int_{y=x}^N \int_{z=y}^N dz dy dx$$

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:

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

, where in the condition it is o rather than O .

Then:

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

Case 2

If:

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

, for some constant $k \geq 0$

Then:

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

, typically $k = 0$ in most cases.

Case 3

If:

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

, where in the condition it is ω rather than Ω .

And with regularity condition:

$$f\left(\frac{n}{b}\right) \leq k f(n)$$

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

Then:

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

1.3 USEFUL MATH EQUATIONS

Euler:

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

Logarithm power:

$$a^{\log_b n} = n^{\log_b a}$$

2

proof:

$$\begin{aligned}a^{\log_b^n} &= n^{\log_b^a} \\ \Leftrightarrow \ln a^{\log_b^n} &= \ln n^{\log_b^a} \\ \Leftrightarrow \frac{\ln n}{\ln b} \ln a &= \frac{\ln a}{\ln b} \ln n\end{aligned}$$

Chapter 2

Memory Complexity

2.1 INTRODUCTION

When discussing memory complexity, need to consider both

1. **Heap**: the declared variables' size.
2. **Stack**: the recursive functions' call stack.

2.1.1 Memory for Data Type

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
T[]	8N+24

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

Type	Bytes
Object overhead	16
Reference	8
Padding	8x

2. Reference includes object reference and inner class reference.
3. T[] only considers reference; if consider underlying data structure, the memory is $8N+24+xN$, where x is the underlying data structure memory for each element.
4. 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 int N; // 4 (int)
    private T[] items; // 8 (reference to array)
                        // 8N+24 (array of Boolean references)
                        // 24N (underlying Boolean objects)
                        // 4 (padding to round up to a multiple)
}
```

Notice the multiple levels of references.

Notice:

1. The reference takes memory of 8 bytes.

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. Halving: 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>`

1. strictly asc/ strictly desc
2. non-desc/ non-asc

3.2.3 Applications

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.

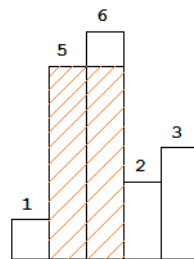


Fig. 3.1: Largest rectangle in histogram

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.

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

Core clues:

1. Maintain the non-decreasing stack. slow performance
2. Popping triggers the calculation of area
3. Calculate the rectangle width by index diff
4. Post-processing in the end

3.2.2 Usage

The core philosophy of using stack is to maintain a relationship invariant among stack element.

The **relationship invariants** can be:

¹ The naming in Python collections is awkward: [discussion](#).

Code:

```
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 array scan, process the dangling stack
    i = n
    ...

    return gmax
```

3.2.4 All nearest smaller values

Nearest smaller. Left neighbor of a value v to be the value that occurs prior to v , is smaller than v , and is closer in position to v than any other smaller value.

For each position in a sequence of numbers, search among the *previous* positions for the last position that contains a smaller value.

Core clues:

1. Maintain a *strictly increasing* stack.
2. If all nearest *larger* values, maintain a *strictly decreasing* stack.

```
def allNearestSmaller(self, A):
    P = [-1 for _ in A]
    stk = []
    for i, v in enumerate(A):
        while stk and A[stk[-1]] >= v: stk.pop()

        if stk:
            P[i] = stk[-1]
        else:
            P[i] = -1 # no preceding smaller value

        stk.append(i) # store the idx or val

    return P
```

3.3 MAP

3.3.1 Math relations

1-1 Map. Mathematically, full projection. 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. Sort the map entries by values `itemgetter`.

```
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

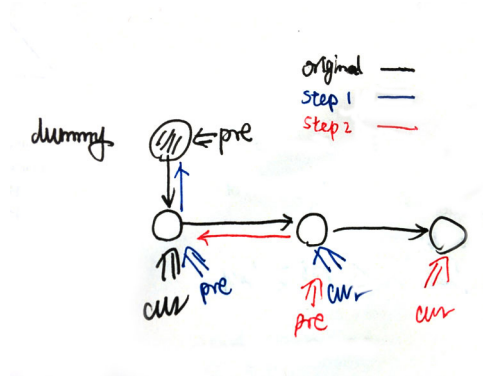


Fig. 4.1: Reverse the linked list

```
def reverseList(self, head):
    dummy = ListNode(0)
    dummy.next = head

    pre = dummy
    cur = pre.next
    while pre and cur:
        pre, cur.next, cur = cur, pre, cur.next
        # incorrect evaluation order:
        # pre, cur, cur.next = cur, cur.next, pre

    dummy.next.next = None # original head
    return pre # new head
```

Notice: the evaluation order for the swapping the nodes and links.

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

4.2 COMBINATIONS

4.2.1 LRU

Core clues:

1. Ensure $O(1)$ find $O(1)$ deletion.
2. Doubly linked list + map.
3. Keep both *head* and *tail* pointer.
4. Operations on doubly linked list are case by case.

```
class Node(object):
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.pre, self.next = None, None

class LRUCache(object):
    def __init__(self, capacity):
        self.cap = capacity
        self.map = {} # key to node
        self.head = None
        self.tail = None

    def get(self, key):
        if key in self.map:
            cur = self.map[key]
            self._elevate(cur)
            return cur.val

        return -1

    def set(self, key, value):
        if key in self.map:
            cur = self.map[key]
```



```

        cur.val = value
        self._elevate(cur)
    else:
        cur = Node(key, value)
        self.map[key] = cur
        self._appendleft(cur)

        if len(self.map) > self.cap:
            last = self._pop()
            del self.map[last.key]

# doubly linked-list operations only
def _appendleft(self, cur):
    """Normal or initially empty"""
    if not self.head and not self.tail:
        self.head = cur
        self.tail = cur
    return

    head = self.head
    cur.next, cur.pre, head.pre = head, None, cur
    self.head = cur

def _pop(self):
    """Normal or resulting empty"""
    last = self.tail
    if self.head == self.tail:
        self.head, self.tail = None, None
    return last

    pre = last.pre
    pre.next = None
    self.tail = pre
    return last

def _elevate(self, cur):
    """Head, Tail, Middle"""
    pre, nxt = cur.pre, cur.next
    if not pre:
        return
    elif not nxt:
        assert self.tail == cur
        self._pop()
    else:
        pre.next, nxt.pre = nxt, pre

    self._appendleft(cur)

```

Chapter 5

Heap

5.1 INTRODUCTION

Heap-ordered. Binary heap is one of the implementations of Priority Queue (ADT). The core relationship of elements in the heap: $A_{2i} \leq A_i \geq A_{2i+1}$.

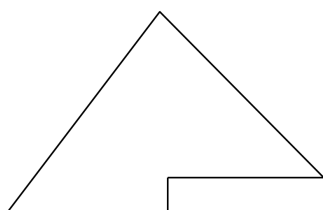


Fig. 5.1: Heap

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 (sift_down)

Core clue: compare parent to the *larger* child.

```
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 (sift_up)

Core clue: compare child to its parent.

```
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

Core clue: bottom-up sink().

```
def heapify(self):
    for i in xrange(self.N/2, 0, -1):
        self.sink(i);
```

Complexity. 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 $O(N \lg N)$. Intuitively, the deeper the level, the more the nodes, but the less the level to sink down.

At most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of any height h .

Proof:

$$\begin{aligned} \because \sum_{i=0}^{+\infty} ix^i &= \frac{x}{(1-x)^2} \\ \therefore \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) &= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) \\ &= O(n) \end{aligned}$$

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**.

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 .

It is possible to have 3-heap. 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 $a[3k+1]$.

5.3.3 Java Priority Queue

```
// min-heap
PriorityQueue<Integer> pq = new PriorityQueue<>(
    (o1, o2) -> o1-o2
);

// max-heap
PriorityQueue<Integer> pq = new PriorityQueue<>(
    (o1, o2) -> o2-o1
);
```

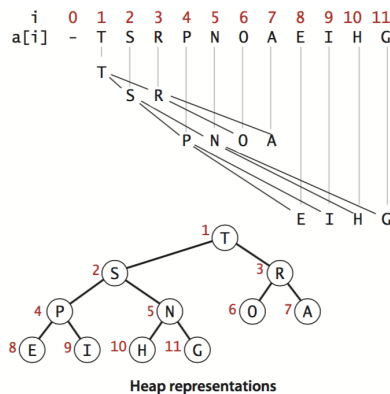


Fig. 5.2: Heap representation

5.3.2 Python Heapq

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

1. Invert the number: 1 becomes -1. (usually the best solution)
2. Wrap the data into another class and override **comparators**: `__cmp__` or `__lt__`

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)
```

Chapter 6

Tree

6.1 BINARY TREE

6.1.1 Introductions

Get parent ref. To get a parent reference (implicitly), *return the Node* of 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;
}
```

Construct path from root to target. To search a node in binary tree (not necessarily BST), use dfs:

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

6.1.2 Morris Traversal

Traversal with $O(1)$ space.²

Time complexity $O(3n)$. - find **pre** twice, **cur** traverse once.

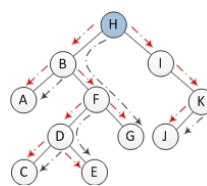


Fig. 6.1: Morris traversal time complexity

6.1.2.1 Inoder

Assign the current node's in-order predecessor's right child to itself (threading). Two ptr **cur**, **pre**.

Process:

1. If no left, *consume cur*, go right
2. If left, find in-order predecessor **pre**
 - a. If no thread (i.e. no **pre** right child), assign it to **cur**; go left
 - b. If thread, *consume cur*, go right. (\equiv no left).

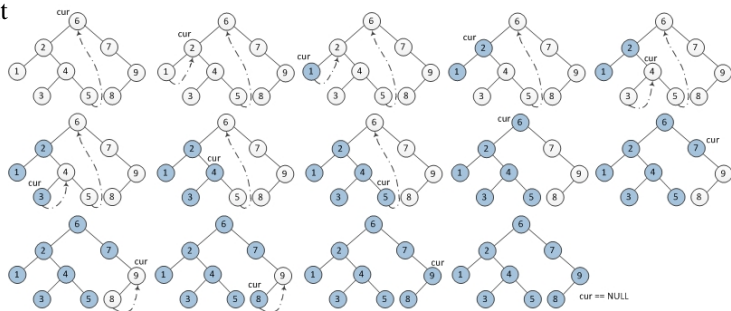


Fig. 6.2: Morris inorder traversal

² ref

Code:

```
def morris_inorder(self, root):
    cur = root
    while cur:
        if not cur.left:
            self.consume(cur)
            cur = cur.right
        else:
            pre = cur.left
            while pre.right and pre.right != cur:
                pre = pre.right

            if not pre.right:
                pre.right = cur
                cur = cur.left
            else:
                pre.right = None
                self.consume(cur)
                cur = cur.right
```

6.1.2.2 Preoder

Similar to inorder.

Process:

1. If no left, *consume* **cur**, go right
2. If left, find in-order predecessor **pre**
 - a. If no thread (i.e. no **pre** right child), assign it to **cur**; *consume* **cur**, go left
 - b. If thread, go right. (\equiv no left, but no *consume*, since *consume* before).

6.1.2.3 Postorder

More tedious.

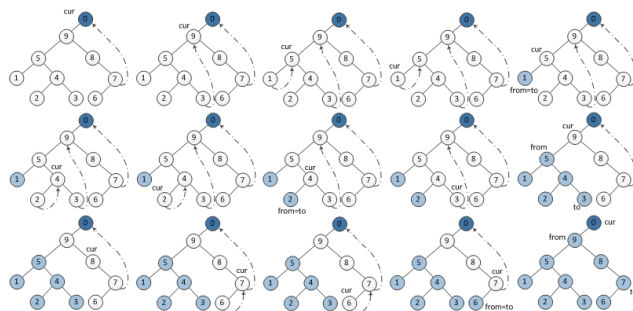


Fig. 6.3: Morris inorder traversal

TODO

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 Rank

Calculates rank.

1. When inserting:
 - a. insert to an existing node: **node.cnt_this** += 1
 - b. insert to left subtree: **node.cnt_left** += 1
 - c. insert to right subtree: do nothing.
2. When querying rank:
 - a. query equals current node: **return node.cnt_left**
 - b. query goes to **left** node: **return rank(node.left, val)**;
 - c. query goes to **right** node: **return node.cnt_left + node.cnt_this + rank(node.right, val)**

Notice that the **rank** calculates a **val**'s rank in a subtree.

Count of smaller number before itself. Given an array A . For each element A_i in the array, count the number of element before this element A_i is smaller than it and return count number array. Average $O(n \log n)$

Clues:

1. Put $A[i+1]$ into a BST; so as to count the rank of $A[i]$ in the BST

Codes:

```
class Node(object):
    def __init__(self, val):
        """Records the left subtree size"""
        self.val = val
        self.cnt_left = 0
        self.cnt_this = 0
        self.left, self.right = None, None

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

    def insert(self, root, val):
        """
        :return: subtree's root after insertion
        """
        if not root:
            root = Node(val)

        if root.val == val:
            root.cnt_this += 1
        elif val < root.val:
            root.cnt_left += 1
            root.left = self.insert(root.left, val)
        else:
            root.right = self.insert(root.right, val)

        return root
```

```
def rank(self, root, val):
    """
    Rank in the root's subtree
    :return: number of items smaller than val
    """
    if not root:
        return 0
    if root.val < val:
        return (root.cnt_this+root.cnt_left+
                self.rank(root.right, val))
    elif root.val == val:
        return root.cnt_left
    else:
        return self.rank(root.left, val)

class Solution(object):
    def countOfSmallerNumberII(self, A):
        tree = BST()
        ret = []
        for a in A:
            tree.root = tree.insert(tree.root, a)
            ret.append(tree.rank(tree.root, a))

        return ret
```

Notice: if worst case $O(n \log n)$ is required, need to use Red-Back Tree - Section 7.2. However, there is a more elegant way using Segment Tree - Section 8.5.3.

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);
}
```

Closest value Find the value in BST that is closet to the target.

Clues:

1. Find the value just \leq the target.
2. Find the value just \geq the target.

Code for finding either the lower value or higher value:

```
def find(self, root, target, ret, lower=True):
    """ret: result container"""
    if not root: return

    if root.val == target:
        ret[0] = root.val
        return

    if root.val < target:
        if lower:
            ret[0] = max(ret[0], root.val)

        self.find(root.right, target, ret, lower)
    else:
```

```
        if not lower:
            ret[0] = min(ret[0], root.val)

        self.find(root.left, target, ret, lower)
```

Closest values Find k values in BST that are closet to the target.

Clues:

1. Find the predecessors $\triangleq \{node | node.value \leq target\}$. Store in the stack.
2. Find the successors $\triangleq \{node | node.value \geq target\}$. Store in the stack.
3. Merge the predecessors and successors as in merge in MergeSort to get the k values.

Code for finding the predecessors:

```
def predecessors(self, root, target, stk):
    if not root: return

    self.predecessors(root.left, target, stk)
    if root.val <= target:
        stk.append(root.val)
    self.predecessors(root.right, target, stk)
```

6.3 BINARY INDEX TREE (BIT)

6.3.1 Introduction

Compared to Segment Tree 6.4, BIT is shorter and more elegant. BIT can do most of things that Segment Tree can do and it is easier to code. BIT updates and queries

$$i \rightarrow prefixSum$$

in $O(\log n)$ time; however, BIT CANNOT query

$$prefixSum \rightarrow i$$

6.3.2 Implementation

Given an array A of length n starting from 1. prefix sum $s[i] \triangleq A_1 + \dots + A_i$. BIT uses binary to maintain the array of prefix sum for querying and updating. For i -th node in the BIT,

$$N[i] = A_{j+1} + \dots + A_i$$

, where $j = i - lowbit(i)$, i.e. set i 's lowest bit 1 to 0. $lowbit(i)$ can be defined as `return i & -i`, using 2's com-

plement. Notice that the summation ends with A_i since easier to **set**.

For the range, we use $(j, i]$ here instead of $[j, i)$ since more elegant for **get**(i) and **set**(i)

Clues:

1. Binary
2. Low bit
3. BIT uses array index starting from 1, because 0 doesn't have *lowbit*.

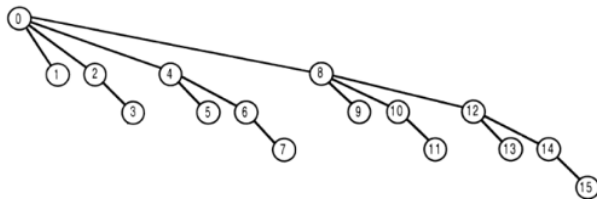


Fig. 6.4: Binary Indexed Tree *get* Operation

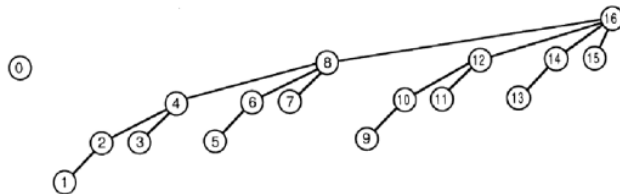


Fig. 6.5: Binary Indexed Tree *set* Operation

Time complexity, longest update is along the leftmost branch, which takes $O(\log_2 n)$ (e.g. 1, 10, 100, 1000, 10000); longest query is along a branch starting with node with all 1's (e.g. 1111, 1110, 1100, 1000), which also takes $O(\log_2 n)$.

Code:

```
class BIT(object):
    def __init__(self, n):
        """BIT uses index starting from 1"""
        self.N = [0 for _ in xrange(n+1)]

    def lowbit(self, i):
        return i & -i

    def get(self, i):
        ret = 0
        while i > 0:
            ret += self.N[i]
            i -= self.lowbit(i)

        return ret

    def set(self, i, val):
        while i < len(self.N):
            self.N[i] += val
            i += self.lowbit(i)
```

6.4 SEGMENT TREE

6.4.1 Introduction

Segment Tree is specially built for *range queries*.

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:

```
# a Count Segment Tree
[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

Segment Tree does a decent job for range queries.

Components in Segment Tree operations:

1. Build
2. Query
3. Modify
4. Search

Notice:

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

Code: Notice the code has abstracted out segment tree functions of sum, min/max or count, by abstracting the subtree combine function to `lambda`.

```
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)
        Either check s+1==e or have root.right
        only if have root.left
        """
        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
```

Concrete example - Count Segment Tree [8.5.4](#).

6.5 TRIE

6.5.1 Basic

Trie is aka radix tree, prefix tree.

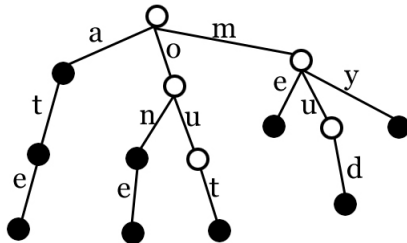


Fig. 6.6: Trie

Notice:

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

Codes:

```
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
```

6.5.2 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):
        """Implicit storage"""
        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
```

6.5.3 Applications

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

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.

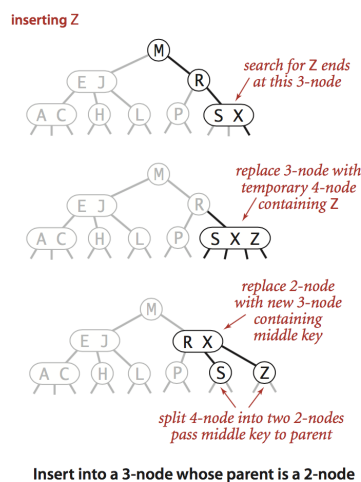


Fig. 7.1: Insertion 1

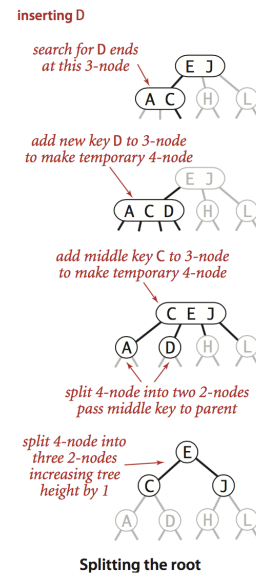


Fig. 7.2: insert 2

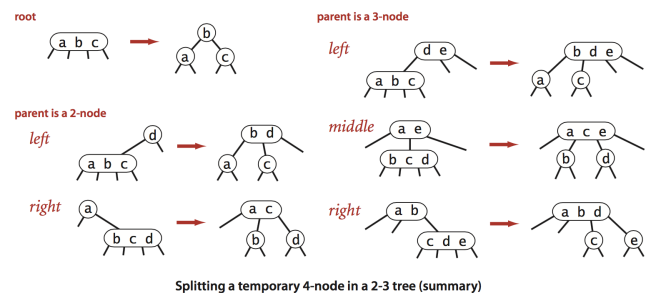


Fig. 7.3: Splitting temporary 4-node summary

7.1.2 Splitting

Summary of splitting the tree.

7.1.3 Properties

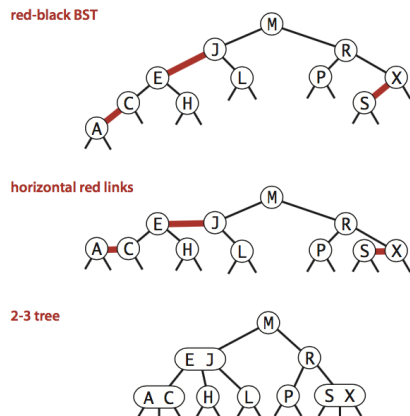
When inserting a new key into a 2-3 tree, under which one of the following scenarios must the height of the 2-3

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 height of the RB-tree is at most



1-1 correspondence between red-black BSTs and 2-3 trees

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

$2\lg N$ where alternating red and black links. Red is the special link while black is the default link.

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

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:

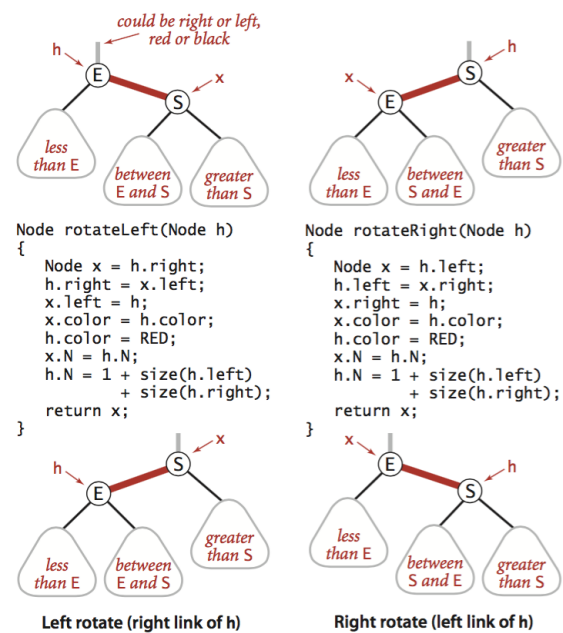


Fig. 7.5: Rotate left/right

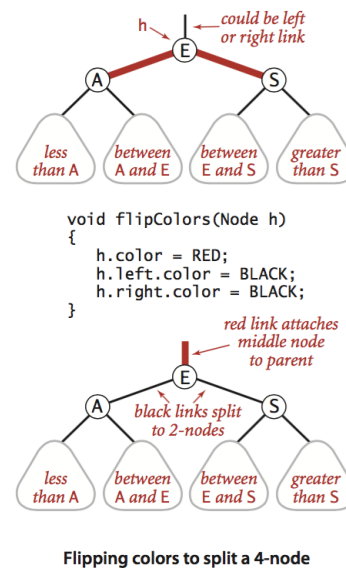


Fig. 7.6: Flip colors

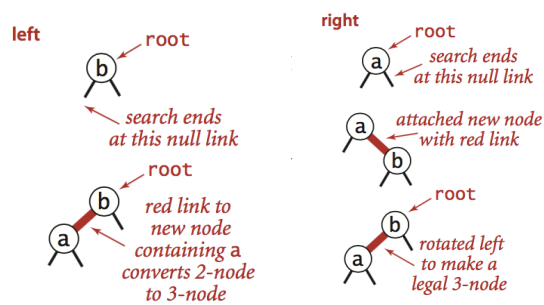


Fig. 7.7: (a) smaller than 2-node (b) larger than 2-nod

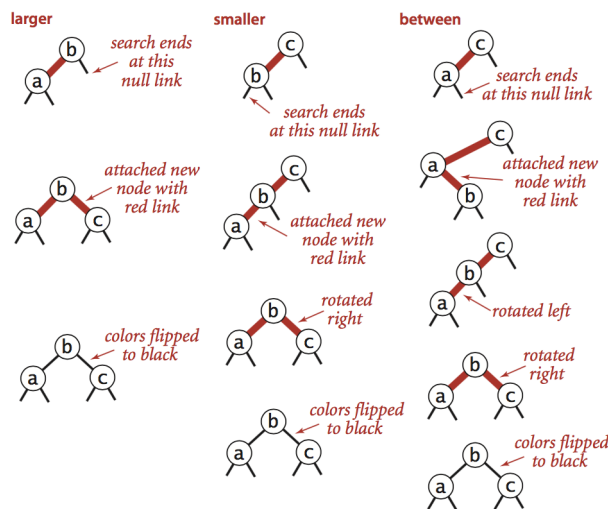


Fig. 7.8: (a) larger than 3-node (b) smaller than 3-node (c) between 3-node.

```

Node put(Node h, Key key, Value val) {
    if (h == null) // std red insert (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.

Illustration of cases. Insert into a single 2-node: Figure-7.7. Insert into a single 3-node: Figure-7.8

Deletion. Deletion is more complicated.

7.3 B-TREE

B-tree is the generalization of 2-3 tree.

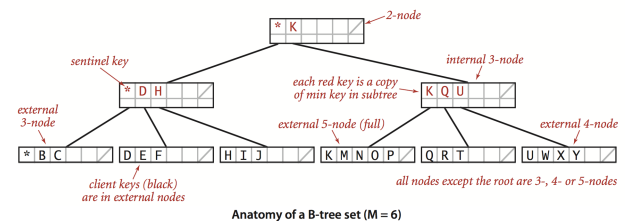


Fig. 7.9: B-Tree

7.3.1 Basics

Half-full principle:

Attrs Non-leaf Leaf

Ptrs $\lceil \frac{n+1}{2} \rceil$ $\lfloor \frac{n+1}{2} \rfloor$

7.3.2 Operations

Core clues

1. **Invariant:** children balanced or left-leaning
2. **Split:** split half, thus invariant.
3. **Leaf-Up:** no delete, recursively move up the right node's first child; thus invariant.
4. **Nonleaf-Up:** delete and recursively move up the left's last if left-leaning or right's first if balanced; thus invariant.

7.4 AVL TREE

TODO

7.5 CARTESIAN TREE

7.5.1 Basics

Also known as max tree (or min tree). The root is the maximum number in the array. The left subtree and right subtree are the max trees of the subarray divided by the root number.

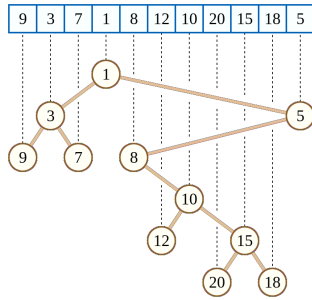
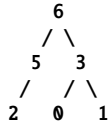


Fig. 7.10: Cartesian Tree

Given [2, 5, 6, 0, 3, 1], the max tree is



Construction algorithm. Similar to all nearest smaller (or larger) values problem - Section 3.2.4.

Core clues:

1. Use stack to maintain a *strictly decreasing* stack, similar to find the all nearest large elements. Maintain the tree for currently scanning A_i with the subarray $A[:i]$.
 - a. **Left tree.** For each currently scanning node A_i , if $stk_{-1} \leq A_i$, then stk_{-1} is the left subtree of A_i . Then pop the stack and iteratively look at stk_{-1} again (previously stk_{-2}). Notice that the original left subtree of A_i should become the right subtree of stk_{-1} , because the original left subtree appears later and satisfies the decreasing relationship.
 - b. **Right tree.** In this stack, $stk_{-1} < stk_{-2}$ and stk_{-1} appears later than stk_{-2} ; thus stk_{-1} is the right subtree of stk_{-2} . The strictly decreasing relationship of stack will be processed when popping the stack.

$O(n)$ since each node on the tree is pushed and popped out from stack once.

```
def maxTree(self, A):
    stk = []
    for a in A:
        cur = TreeNode(a)
        while stk and stk[-1].val <= cur.val:
            pre = stk.pop()
            pre.right = cur.left
            cur.left = pre
        stk.append(cur)

    pre = None
    while stk:
        cur = stk.pop()
        cur.right = pre
        pre = cur

    return pre
```

Usually, min tree is more common.

7.5.2 Treap

Randomized Cartesian tree. Heap-like tree. It is a Cartesian tree in which each key is given a (randomly chosen) numeric priority. As with any binary search tree, the in-order traversal order of the nodes is the same as the sorted order of the keys.

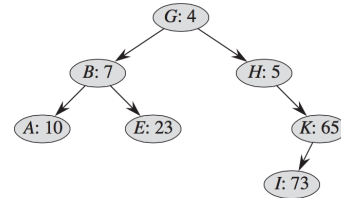


Fig. 7.11: Treap. Each node x is labeled with $x.key$: $x.priority$.

Construct a Treap for an array A with index as the $x.key$ randomly chosen priority $x.priority$ $O(n)$. Thus support search, insert, delete into array (i.e. Treap) $O(\log n)$ on average.

Insertion and deletion - need to perform *rotations* to maintain the min-treap property.

Chapter 8

Sort

8.1 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. $|A_p| \leq \dots |..unseen..| \geq \dots$ 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.

8.2 ALGORITHMS

8.2.1 Quick Sort

8.2.1.1 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 go $O(N^2)$ 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.

8.2.1.2 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
```

8.2.1.3 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

```

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 09:00:03
Houston 09:00:13	Chicago 09:21:05	Chicago 09:00:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:00:59
Houston 09:01:10	Chicago 09:19:32	Chicago 09:01:10
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:03:13
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:10:11
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:10:25
Phoenix 09:14:25	Houston 09:01:10	Phoenix 09:14:25
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Houston 09:00:13
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:00:03
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:14:25
Seattle 09:22:54	Seattle 09:10:25	Phoenix 09:37:44
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:11
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:10:25
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:43
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:22:54
		Seattle 09:36:14

Fig. 8.1: Stable sort vs. unstable sort

8.2.2 Merge Sort

TODO

8.3 PROPERTIES

8.3.1 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.

8.3.2 Sort Applications

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

8.3.3 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 $O(N)$.

8.3.4 Summary

	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		$N^2/2$	$2 N \lg N$	$N \lg N$	$N \lg N$ probabilistic guarantee fastest in practice
3-way quick	x		$N^2/2$	$2 N \lg N$	N	improves quicksort in presence of duplicate keys
merge		x	$N \lg N$	$N \lg N$	$N \lg N$	$N \lg N$ guarantee, stable
heap	x		$2 N \lg N$	$2 N \lg N$	$N \lg N$	$N \lg N$ guarantee, in-place

Fig. 8.2: Sort summary

8.4 PARTIAL QUICKSORT

8.4.1 Find m smallest

Heap-based solution. $O(n \log m)$

Partial Quicksort Then the $A[:m]$ is sorted m smallest. The algorithm recursively sort the $A[i:j]$

The average time complexity is

$$F(n) = \begin{cases} F(\frac{n}{2}) + O(n) & \text{// if } \frac{n}{2} \geq m \\ 2F(\frac{n}{2}) + O(n) & \text{// otherwise} \end{cases}$$

Therefore, the complexity is $O(n + m \log m)$.

```
def partial_qsort(self, A, i, j, m):
    if i >= j: return

    p = self.pivot(A, i, j)
    self.partial_qsort(A, i, p, m)
    if p+1 >= m: return
    self.partial_qsort(A, p+1, j, m)
```

8.4.2 Find k -th

Use partial quick sort to find k -th smallest element in the unsorted array. The algorithm recursively sort the $A[i:j]$

The average time complexity is

$$\begin{aligned} F(n) &= F(n/2) + O(n) \\ &= O(n) \end{aligned}$$

```
def find_kth(self, A, i, j, k):
    if i >= j: return

    p = self.pivot(A, i, j)
    if p == k: return A[p]
    if p > k: return self.find_kth(A, i, p, k)
    else: return self.find_kth(A, p+1, j, k)
```

Pivoting see section - 8.2.1.1.

8.5 INVERSION

If $a_i > a_j$ but $i < j$, then this is considered as 1 Inversion. That is, for an element, the count of other elements that are *larger* than the element but appear *before* it. This is the default definition.

There is also an alternative definition: for an element, the count of other elements that are *smaller* than the element but appear *after* it.

8.5.1 MergeSort & Inversion Pair

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

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

    return ret
```

```
def merge_sort(a):
    n = len(a)
    if n == 1:
        return 0

    a1 = a[:n/2]
    a2 = a[n/2:]

    ret1 = merge_sort(a1)
    ret2 = merge_sort(a2)
    # merge not merge_sort
    ret = ret1+ret2+merge(a1, a2, a)
    return ret
```

8.5.2 Binary Index Tree & Inversion Count

Given A , calculate each element's inversion number.

Construct a BIT (6.3) with length $\max(A) + 1$. Let BIT maintains the index of values. Scan the element from left to right (or right to left depends on the definition of inversion number), and set the index equal val to 1. Use the prefix sum to get the inversion number.

get(end) - **get(a)** get the count of number that appears *before* a (i.e. already in the BIT) and also *larger* than a .

Possible to extend to handle duplicate number.

Core clues:

1. BIT maintains **index of values** to count the number of at each value.

2. `get(end) - get(a)` to get the inversion count of a .

```
def inversion(self, A):
    bit = BIT(max(A)+1)
    ret = []
    for a in A:
        bit.set(a, 1) # += 1 if possible duplicate
        inversion = bit.get(max(A)+1) - bit.get(a)
        ret.append(inversion)

    return ret
```

8.5.3 Segment Tree & Inversion Count

Compared to BIT, Segment Tree can process queries of both $idx \rightarrow sum$ and $sum \rightarrow idx$; while BIT can only process $idx \rightarrow sum$.

Core clues:

1. Segment Tree maintains **index of values** to count the number of at each value.
2. `get(root, end) - get(root, a)` to get the inversion count of a .

```
class SegmentTree(object):
    def __init__(self):
        self.root = None

    def build(self, root, lo, hi):
        if lo >= hi: return
        if not root: root = Node(lo, hi)

        root.left = self.build(root.left, lo, (lo+hi)/2)
        if root.left:
            root.right = self.build(root.right, (lo+hi)/2, hi)

        return root

    def set(self, root, i, val):
        if root.lo == i and root.hi-1 == root.lo:
            root.cnt_this += val
        elif i < (root.lo+root.hi)/2:
            root.cnt_left += val
            self.set(root.left, i, val)
        else:
            self.set(root.right, i, val)

    def get(self, root, i):
        if root.lo == i and root.hi-1 == root.lo:
            return root.cnt_left
        elif i < (root.lo+root.hi)/2:
            return self.get(root.left, i)
        else:
            return (
                root.cnt_left + root.cnt_this +
                self.get(root.right, i)
            )
```

```
class Solution(object):
    def _build_tree(self, A):
        st = SegmentTree()
        mini, maxa = min(A), max(A)
```

```
st.root = st.build(st.root, mini, maxa+2)
# maxa+1 is the end dummy
return st
```

```
def countOfLargerElementsBeforeElement(self, A):
    st = self._build_tree(A)
    ret = []
    end = max(A)+1
    for a in A:
        ret.append(
            st.get(st.root, end) - st.get(st.root, a)
        )
        st.set(st.root, a, 1)

    return ret
```

8.5.4 Reconstruct Array from Inversion Count

Given a *sorted* numbers with their associated inversion count (# larger numbers before this element). $A[i].val$ is the value of the number, $A[i].inv$ is the inversion number. Reconstruct the original array R that consists of each $A[i].val$.

Brute force can be done in $O(n^2)$. Put the $A[i].val$ into R at an index/slot s.t. the # *empty* slots before it equals to $A[i].inv$.

BST. Possible to use BST to maintain the empty slot indexes in the original array. Each node's rank indicates the count of empty indexes in its left subtree. But need to maintain the deletion.

Segment Tree. Use a segment tree to maintain the size of empty slots. Each node has a *start* and a *end* s.t slot indexes $\in [start, end)$. Go down to find the target slot, go up to decrement the size of empty slots.

Reconstruction of array cannot use BIT since there is no map of $prefixSum \rightarrow i$.

```

class Node(object):
    def __init__(self, start, end, cnt):
        self.start = start
        self.end = end
        self.cnt = cnt

        self.left = None
        self.right = None

    def __repr__(self):
        return repr("[%d,%d]" % (self.start, self.end))

class SegmentTree(object):
    """empty space"""
    def __init__(self):
        self.root = None

    def build(self, start, end):
        """a node can have right ONLY IF has left"""
        if start >= end:
            return

        root = Node(start, end, end-start)
        root.left = self.build(start, (end+start)/2)
        if root.left:
            root.right = self.build((start+end)/2, end)
        return root

    def find_delete(self, root, val):
        """
        :return: index
        """
        root.cnt -= 1
        if not root.left:
            return root.start
        elif root.left.cnt >= val:
            return self.find_delete(root.left, val)
        else:
            return self.find_delete(root.right,
                                    val - root.left.cnt)

class Solution(object):
    def reconstruct(self, A):
        st = SegmentTree()
        n = len(A)
        st.root = st.build(0, n)
        A = sorted(A, key=lambda x: x[0])
        ret = [0]*n
        for a in A:
            idx = st.find_delete(st.root, a[1]+1)
            ret[idx] = a[0]

        return ret

if __name__ == "__main__":
    A = [(5, 0), (2, 1), (3, 1), (4, 1), (1, 4)]
    assert Solution().reconstruct(A) == [5, 2, 3, 4, 1]

```

Chapter 9

Search

9.1 BINARY SEARCH

Variants:

1. get the idx equal or just lower (floor)
2. get the idx equal or just higher (ceil)
3. `bisect_left`
4. `bisect_right`

9.1.1 idx equal or just lower

Binary search, get the idx of the element equal to or just lower than the target. The returned idx is the $A_{idx} \leq target$. It is possible to return -1 . It is different from the `bisect_left`.

Core clues:

1. To get “equal”, `return mid`.
2. To get “just lower”, `return lo-1`.

$A_{idx} \leq target$.

```
def bin_search(self, A, t, lo=0, hi=None):
    if hi is None: 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
```

9.1.2 idx equal or just higher

$A_{idx} \geq target$.

```
def bin_search(self, A, t, lo=0, hi=None):
    if hi is None: 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
```

9.1.3 bisect_left

Return the index where to insert item x in list A. So if t already appears in the list, `A.insert(t)` will insert just before the *leftmost* t already there.

Core clues:

1. Move `lo` if $A_{mid} < t$
2. Move `hi` if $A_{mid} \geq t$

```
def bisect_left(A, t, lo=0, hi=None):
    if hi is None: hi = len(A)

    while lo < hi:
        mid = (lo+hi)/2
        if A[mid] < t: lo = mid+1
        else: hi = mid

    return lo
```

9.1.4 bisect_right

Return the index where to insert item x in list A. So if t already appears in the list, `A.insert(t)` will insert just after the *rightmost* x already there.

Core clues:

1. Move `lo` if $A_{mid} \leq t$
2. Move `hi` if $A_{mid} > t$

```
def bisect_right(A, t, lo=0, hi=None):
    if hi is None: hi = len(A)

    while lo < hi:
        mid = (lo+hi)/2
        if A[mid] <= t: lo = mid+1
        else: hi = mid

    return lo
```

9.2 APPLICATIONS

9.2.1 Rotation

Find Minimum in Rotated Sorted Array. Three cases to consider:

1. Monotonous
2. Trough
3. Peak

If the elements can be duplicated, need to detect and skip.

```
def findMin(self, A):
    lo = 0
    hi = len(A)
    mini = sys.maxint
    while lo < hi:
        mid = (lo+hi)/2
        mini = min(mini, A[mid])
        if A[lo] == A[mid]: # JUMP
            lo += 1
        elif A[lo] < A[mid] <= A[hi-1]:
            return min(mini, A[lo])
        elif A[lo] > A[mid] <= A[hi-1]: # trough
            hi = mid
        else: # peak
            lo = mid+1

    return mini
```

9.3 COMBINATIONS

9.3.1 Extreme-value problems

Longest increasing subsequence. Array A .

Clues:

1. **MIN**: min of index *last* value of LIS of a particular *len*.
2. **RET**: result table, store the π 's idx (predecessor); (optional, to build the LIS, no need if only needs to return the length of LIS)
3. **bin_search**: For each currently scanning index i , if it smaller (i.e. \neg increasing), to maintain the **MIN**, binary search to find the position to update the min value. The **bin_search** need to find the element \geq to $A[i]$.

```
def LIS(self, A):
    n = len(A)
    MIN = [-1 for _ in xrange(n+1)]

    l = 1
    for i in xrange(1, n):
        if A[i] > A[MIN[l]]:
            l += 1
            MIN[l] = i
        else:
            j = self.bin_search(MIN, A, A[i], 1, l+1)
            MIN[j] = i

    return l
```

If need to return the LIS itself.

```
for i in xrange(1, n):
    if A[i] > A[MIN[l]]:
        l += 1
        MIN[l] = i

    RET[i] = MIN[l-1] # (RET)
else:
    j = self.bin_search(MIN, A, A[i], 1, l+1)
    MIN[j] = i

    RET[i] = MIN[j-1] if j-1 >= 1 else -1 # (RET)

# build the LIS (RET)
cur = MIN[l]
ret = []
while True:
    ret.append(A[cur])
    if RET[cur] == -1: break
    cur = RET[cur]

ret = ret[::-1]
print ret
```

9.4 HIGH DIMENSIONAL SEARCH

9.4.1 2D

2D search matrix I. $m \times n$ mat. Integers in each row are sorted from left to right. The first integer of each row is greater than the last integer of the previous row.

$$\begin{bmatrix} 1 & 3 & 5 & 7 \\ 10 & 11 & 16 & 20 \\ 23 & 30 & 34 & 50 \end{bmatrix}$$

Row column search: starting at top right corner: $O(m+n)$.

Binary search: search rows and then search columns: $O(\log m + \log n)$.

2D search matrix II. $m \times n$ mat. Integers in each row are sorted from left to right. Integers in each column are sorted in ascending from top to bottom.

$$\begin{bmatrix} 1 & 4 & 7 & 11 & 15 \\ 2 & 5 & 8 & 12 & 19 \\ 3 & 6 & 9 & 16 & 22 \\ 10 & 13 & 14 & 17 & 24 \\ 18 & 21 & 23 & 26 & 30 \end{bmatrix}$$

Row column search: starting at top right corner: $O(m + n)$.

Binary search: search rows and then search columns, but upper bound row and lower bound row:

$$O(\min(n \log m, m \log n))$$

Chapter 10

Array

10.1 CIRCULAR ARRAY

This section describes common patterns for solving problems with circular arrays.

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

10.1.1 Circular max sum

Linear problem can be solved linear with dp algorithm for maximum subarray sum - Section 21.2.

The circular sum should use dp.

Problem description: Given an integer array, find a continuous rotate subarray where the sum of numbers is the biggest. Return the index of the first number and the index of the last number.

Core clues:

1. **State definitions:**

Construct left max sum L_i for max sum over the $[0..i]$ with subarray starting at 0 (*forward* starting from the left side).

Construct right max sum R_i for max sum over the indexes $[i+1..n-1]$, with subarray ending at -1 (*backward* starting from the right side).

Notice, for the two max sums, the index ends AT or BEFORE i .

2. **Transition functions:**

$$L_i = \max \left(L_{i-1}, \text{sum}(A[:i]) \right)$$

$$R_i = \max \left(R_{i+1}, \text{sum}(A[i:]) \right)$$

3. **Global result:**

$$\text{maxa} = \max(R_i + L_{i-1}, \forall i)$$

10.1.2 Non-adjacent cell

Maximum sum of non-adjacent cells in an array A .

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 $A[-1]$

and solve them using linear maximum sum of non-adjacent cells separately - Section 21.2.

10.1.3 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 *derive and decide* whether to go to the **sorted half** or the **unsorted half**.

10.2 VOTING ALGORITHM

10.2.1 Majority Number

10.2.1.1 $\frac{1}{2}$ 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 capture 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.

```
def majorityElement(self, nums):
    """
    Algorithm:
    O(n lgn) sort and take the middle one
    O(n) Moore's Voting Algorithm
    """
    mjr = nums[0]
    cnt = 0
    for i, v in enumerate(nums):
        if mjr == v:
            cnt += 1
        else:
            cnt -= 1

        if cnt < 0:
            mjr = v
            cnt = 1

    return mjr
```

10.2.1.2 $\frac{1}{3}$ 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.

10.2.1.3 $\frac{1}{k}$ 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. In this case, we need to generalize the solution to $\frac{1}{2}$ majority number problem.

```
def majorityNumber(self, nums, k):
    """
    Since majority elements appears more
    than ceil(n/k) times, there are at
    most k-1 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]
```

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

raise Exception
```

10.3 TWO POINTERS

10.3.1 Interleaving

Interleaving positive and negative numbers. Given an array with positive and negative integers. Re-range it to interleaving with positive and negative integers.

Input:
[-33, -19, 30, 26, 21, -9]
Output:
[-33, 30, -19, 26, -9, 21]

Core clues:

1. In 1-pass.
2. What (positive or negative) is expected for the current position.
3. Where is the next positive and negative element.

```
def rerange(self, A):
    n = len(A)
    pos_cnt = len(filter(lambda x: x > 0, A))
    pos_expt = True if pos_cnt*2 > n else False

    neg = 0 # next negative
    pos = 0 # next positive
    for i in xrange(n):
        while neg < n and A[neg] > 0: neg += 1
        while pos < n and A[pos] < 0: pos += 1
        if pos_expt:
            A[i], A[pos] = A[pos], A[i]
        else:
            A[i], A[neg] = A[neg], A[i]

        if i == neg: neg += 1
        if i == pos: pos += 1

    pos_expt = not pos_expt
```

Chapter 11

String

11.1 PALINDROME

11.1.1 Palindrome anagram

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

Count palindrome anagram. See Section-15.1.4.

Construct palindrome anagram. Construct all palindrome anagrams given a string s .

Clues:

1. dfs, grow the counter map of s .
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
```

11.2 KMP

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

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

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. 11.1: Prefix-suffix table

In table-building algorithm, similar to dp, let $T[i]$ store the length of matched prefix suffix for $needle[:i]$

Clues:

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

Table-building code:

```
# construct T
T = [0 for _ in xrange(len(needle)+1)]
T[0] = -1
T[1] = 0

cnd = 0
i = 2 # table index
while i < len(needle)+1:
    if needle[i-1] == needle[cnd]: # matched
        T[i] = cnd+1
        cnd += 1
        i += 1
    elif T[cnd] != -1: # fall back
        cnd = T[cnd]
    else: # restart
        T[i] = 0
        cnd = 0
        i += 1
```

11.2.2 Searching algorithm

Notice:

1. index i and j .
2. $T[i-1+1]$ for corresponding previous index in T for current scanning index i .
3. When falling back, the next scanning index is $\text{len}(\text{prefix})$
4. three parts:
 - a. matched

- b. aggressive move and fall back
- c. restart

Search 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: # move and fall back j
            j = j+i-T[i]
            i = T[i]
        else: # restart
            j += 1
            i = 0

return None
```

	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. 11.2: KMP example

11.2.3 Applications

1. Find needle in haystack.
2. Shortest palindrome

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.
→ 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. → Dual heap with lazy deletion - section 13.3.2.

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 be-

ing 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 summarize, 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 prime using 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: Section-19.4.1.1.

13.3 MEDIAN

13.3.1 Basic DualHeap

DualHeap to keep track the median when a method to find median is called multiple times.

```
import heapq

class DualHeap(object):
    def __init__(self):
        self.min_h = []
        self.max_h = [] # need to negate the value

    def insert(self, num):
        if not self.min_h or num > self.min_h[0]:
            heapq.heappush(self.min_h, num)
        else:
            heapq.heappush(self.max_h, -num)
        self.balance()

    def balance(self):
        l1 = len(self.min_h)
        l2 = len(self.max_h)
        if l1-l2 > 1:
            heapq.heappush(self.max_h,
                           -heapq.heappop(self.min_h))
            self.balance()
        elif l2-l1 > 1:
            heapq.heappush(self.min_h,
                           -heapq.heappop(self.max_h))
            self.balance()
        return

    def get_median(self):
        """Straightforward"""
```

13.3.2 DualHeap with Lazy Deletion

Clues:

1. Wrap the value and wrap the heap
2. When delete a value, mark it with tombstone.
3. When negate the value, only change the value, not the reference.
4. When heap pop, clean the op first.

```
import heapq
from collections import defaultdict

class Value(object):
    def __init__(self, val):
        self.val = val
        self.deleted = False

    def __neg__(self):
        """negate without creating new instance"""
        self.val = -self.val
        return self

    def __cmp__(self, other):
```

```
        assert isinstance(other, Value)
        return self.val - other.val

    def __repr__(self):
        return repr(self.val)

class Heap(object):
    def __init__(self):
        self.h = []
        self.len = 0

    def push(self, item):
        heapq.heappush(self.h, item)
        self.len += 1

    def pop(self):
        self._clean_top()
        self.len -= 1
        return heapq.heappop(self.h)

    def remove(self, item):
        """lazy delete"""
        item.deleted = True
        self.len -= 1

    def __len__(self):
        return self.len

    def _clean_top(self):
        while self.h and self.h[0].deleted:
            heapq.heappop(self.h)

    def peek(self):
        self._clean_top()
        return self.h[0]

class DualHeap(object):
    def __init__(self):
        self.min_h = Heap() # represent right side
        self.max_h = Heap() # represent left side
        # others similar as the previous section's above DualHeap
```

Chapter 14

Arithmetic

14.1 BIG NUMBER

Plus One. Given a non-negative number represented as an array of digits, plus one to the number.

```
def plusOne(self, digits):
    for i in xrange(len(digits)-1, -1, -1):
        digits[i] += 1
        if digits[i] < 10:
            return digits
        else:
            digits[i] -= 10

    # if not return within the loop
    digits.insert(0, 1)
    return digits
```

14.2 POLISH NOTATIONS

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

Reverse Polish notation (RPN) is a mathematical notation in which every operator follows all of its operands (i.e. operands are followed by operators). RPN should be treated as the orthogonal expression.

Polish notation (PN) is a mathematical notation in which every operator is followed by its operands.

14.2.1 Convert in-fix to post-fix (RPN)

ret stores the final result of reverse polish notation. **stk** stores the temporary result in strictly increasing order.

In-fix

$5 + ((1 + 2) * 4) - 3$

can be written as

$5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$

Core clues:

1. **Stack.** The stack temporarily stores the operators of *strictly increasing precedence order*.

2. **Precedence.** Digits have the highest precedence, followed by $*$, $/$, $+$, $-$. Notice that $()$ operator itself has the *lowest* precedence.

3. **Bracket.** Match the brackets.

Code:

```
def infix2postfix(self, lst):
    stk = []
    ret = [] # post fix result
    for elt in lst:
        if elt.isdigit():
            ret.append(elt)
        elif elt == "(":
            stk.append(elt)
        elif elt == ")":
            while stk and stk[-1] != "(":
                ret.append(stk.pop())
            stk.pop() # pop "("
        else:
            while stk and precdn(elt) <= precdn(stk[-1]):
                ret.append(stk.pop())
            stk.append(elt)

    while stk: # clean up
        ret.append(stk.pop())

    return ret
```

14.2.2 Evaluate post-fix expressions

Consider:

In-fix

$5 + ((1 + 2) * 4) - 3$

Post-fix

$5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$

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

14.2.3 Convert in-fix to pre-fix (PN)

PN is the *reverse* of RPN, thus, scan the expression from right to left; and **stk** stores the temporary result in non-decreasing order.

In-fix

36

5 + ((1 + 2) * 4) - 3

can be written as

3 4 2 1 + * 5 + -

reverse as

- + 5 * + 1 2 4 3

```
def infix2prefix(self, lst):
    """starting from right the left"""
    stk = []
    pre = []
    for elt in reversed(lst):
        if elt.isdigit():
            pre.append(elt)
        elif elt == ")":
            stk.append(elt)
        elif elt == "(":
            while stk and stk[-1] != ")":
                pre.append(stk.pop())
            stk.pop()
        else:
            # < rather than <=
            while stk and precdn(elt) < precdn(stk[-1]):
                pre.append(stk.pop())
            stk.append(elt)

    while stk:
        pre.append(stk.pop())

    pre.reverse()
    return pre
```

14.2.4 Evaluate pre-fix (PN) expressions

Consider:

In-fix

5 + ((1 + 2) * 4) - 3

Pre-fix

- + 5 * + 1 2 4 3

reverse as

3 4 2 1 + * 5 + -

Put into *stack*, similar to evaluating post-fix [14.2.2](#), but pay attention to operands order, which should be reversed when hitting a operator.

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

$$\begin{aligned}\binom{n}{k} &= \frac{n!}{k!(n-k)!} \\ \binom{n}{k} &= \binom{n}{n-k} \\ \binom{n}{k} &= \binom{n-1}{k} + \binom{n-1}{k-1}\end{aligned}$$

15.1.3 N objects, K ceils

When $N = 10, K = 3$:

$$x_1 + x_2 + x_3 = 10$$

is equivalent to

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

, notice that * are non-order.

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:

$$\begin{aligned}ret &= \frac{A_N^N}{\prod_{k=1}^K A_{sz(k)}^{sz(k)}} \\ &= \frac{N!}{\prod_k sz[k]!}\end{aligned}$$

15.1.5 Inclusion–Exclusion Principle

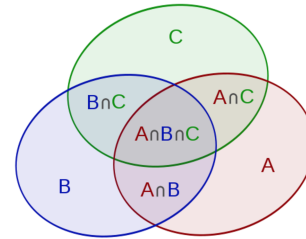


Fig. 15.1: Inclusion–exclusion principl

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

Generally,

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

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. $P_B \dots 4B$. $P_C \dots 5C$.

The result is:

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

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.

15.3 PERMUTATION

15.3.1 k -th permutation

Given n and k , return the k -th permutation sequence. $k \in [1, n!]$. $O(nk)$ in time complexity is easy, can you do it in $O(n^2)$ or less?

Reversed Cantor Expansion

Core clues:

1. $A = [1, 2, \dots, n]$
Suppose for n element, the k -th permutation is:
 $\text{ret} = [a_0, a_1, a_2, \dots, a_{n-1}]$
2. **Basic case.** Since $[a_1, a_3, \dots, a_{n-1}]$ has $(n-1)!$ permutations, if $k < (n-1)!, a_0 = A_0$ (first element in array), else $a_0 = A_{k/(n-1)!}$
3. Recursively, (or iteratively), calculate the values at each position. Similar to Radix.
 - a. $a_0 = A_{k_0/(n-1)!}$, where $k_0 = k$
 - b. $a_1 = A_{k_1/(n-2)!}$, where $k_1 = k_0 \% (n-1)!$ in the remaining array A
 - c. $a_2 = A_{k_2/(n-3)!}$, where $k_2 = k_1 \% (n-2)!$ in the remaining array A

def getPermutation(**self**, **n**, **k**):

k -= 1 # start from 0

A = range(1, n+1)

k %= math.factorial(n)

ret = []

for i **in** xrange(n-1, -1, -1):

idx, **k** = divmod(k, math.factorial(i))

ret.append(A.pop(idx))

return "".join(map(str, ret))

15.4 CATALAN NUMBER

15.4.1 Math

Definition.

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1} \binom{2n}{n} \quad \text{for } n \geq 0$$

Proof. Proof of Catalan Number $C_n = \binom{2n}{n} - \binom{2n}{n+1}$. Objective: count the number of paths in $n \times n$ grid without exceeding the main diagonal.

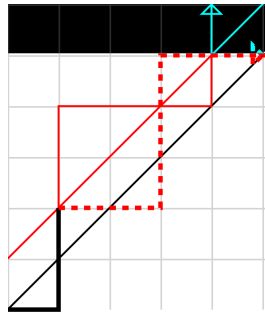


Fig. 15.2: Monotonic Paths

- monotonic paths - n right, n up

$$\binom{2n}{n}$$

- flip at the line just above the diagonal line - $n-1$ right, $n+1$ up

$$\binom{n-1+n+1}{n-1}$$

- thus, the number of path without *exceedance* (i.e. passing the diagonal line) is:

$$\begin{aligned} C_n &= \binom{2n}{n} - \binom{2n}{n-1} \\ &= \binom{2n}{n} - \binom{2n}{n+1} \end{aligned}$$

15.4.2 Applications

The paths in Figure 15.2 can be abstracted to anything that at any time $\#right \geq \#up$.

#Parentheses. Number of different ways of adding parentheses. At any time, $\#(\geq \#)$.

#BSTs. Number of different BSTs. Consider it as a set of same binary operators with their operands. Reduce this problem to #Parentheses.

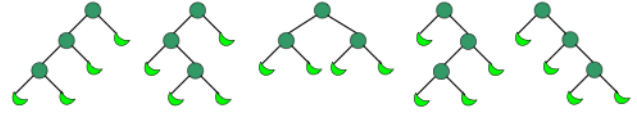


Fig. 15.3: #BSTs. Circles are operators; crescents are operands.

15.5 STIRLING NUMBER

a Stirling number of the second kind (or Stirling partition number) is the number of ways to partition a set of n objects into k non-empty subsets and is denoted by $S(n, k)$ or $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$.

$$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n.$$

Chapter 16

Probability

16.1 SHUFFLE

Equal probability shuffle algorithm.

16.1.1 Incorrect naive solution

Swap current card A_i with a random card from the deck.

```
for (int i = 0; i < N; i++) {  
    int j = (int) Math.random()*N;  
    swap(a[i], a[j]);  
}
```

The easiest proof that this algorithm does not produce a uniformly random permutation is that it generates 27 possible outcomes, but there are only $3! = 6$ permutations. Since $27\%3 \neq 0$, there must be some permutation is that is picked too much, and some that is picked to little.

16.1.2 Knuth Shuffle

Knuth (aka Fisher-Yates) shuffling algorithm guarantees to rearrange the elements in uniformly random order.

Core clues:

1. choose index uniformly $\in [i, N)$

```
public void shuffle(Object[] a) {  
    int N = a.length;  
    for (int i = 0; i < N; i++) {  
        // choose index uniformly in [i, N)  
        int j = i + (int) (Math.random() * (N - i));  
        swap(a[i], a[j]);  
    }  
}
```

16.2 EXPECTED VALUE

16.2.1 Roll dice until expected value.

Chapter 17

Bit Manipulation

17.1 CONCEPTS

17.1.1 Basics

1. Bit value: bit0, bit1.
2. BitSet/Bits
3. Bit position (bit interchangeably)
4. 32-bit signed range: $[-2^{31}, 2^{31} - 1]$. 0 is like positive number without complement.

17.1.2 Operations

Mask.

1. Masking to 1: to mask a single bit position, $bit \mid 1$
2. Masking to 0: to mask a single bit position, $bit \& 0$
3. Querying a bit position value: to query a single bit position, $bit \& 0010$
4. Toggling bit values: to toggle a single bit position, $bit \wedge 1$

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

Check 2's power

$$x \& (x - 1)$$

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

1. Left extended with 1's:

$$x \wedge (-x)$$

2. Left extended with 0's:

$$x \& (-x)$$

Negation and index We can use tilde notation for the index accessing a string or an array

```
i ~i
0 -1
1 -2
2 -3
3 -4
4 -5
5 -6
```

$$\sim i = -i + 1$$

To determine whether a string is palindrome:

```
def is_palindrome(s):
    return all(s[i] == s[~i] for i in xrange(len(s)/2))
```

17.2 SINGLE NUMBER

17.2.1 Three-time appearance

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 &) 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:

$$bitSet^{(i)} = (bitSet^{(i-1)} \& num) \mid (bitSet^{(i)} \& \sim num)$$

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.

Consider each single bit separately. For the j -th bit in num , if $num_j = 1$, the first part indicates $bitSet^{(i-1)}$ will transit in (since transition); the 2nd part is always 0 (since transition out or initially 0). If $num_j = 0$, the 1st part is always 0 (since no transition); the 2nd part indicates $bitSet^{(i)}$ will remain the same (since no transition).

17.2.2 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: $x = a \wedge b$.
- $a \neq b$; 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.

17.3 BITWISE OPERATORS

Comparison. Write a method which finds the maximum of two numbers a, b . You should not use if- else or any other comparison operator

Clues:

1. check the sign bit s of $a - b$.
2. return $a - s * (a - b)$

Codes:

```
int getMax(int a, int b) {
    int c = a - b;
    int k = (c >> 31) & 0x1;
    int max = a - k * c;
    return max;
}
```

If consider overflow, it raises another level of difficulty.

Chapter 18

Greedy

18.1 INTRODUCTION

Philosophy: choose the best options at the current state without reverting the choice in the future.

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Chapter 19

Backtracking

19.1 INTRODUCTION

Difference between backtracking and dfs. *Backtracking* is a more general purpose algorithm. *Dfs* is a specific form of backtracking related to searching tree structures.

Prune. Backtrack need to think about pruning using the condition *predicate*.

19.2 SEQUENCE

k sum. Given n unique integers, number k and target. Find all possible k integers where their sum is target.

Complexity: $O(2^n)$.

Pay attention to the pruning condition.

```
def dfs(self, A, i, k, cur, remain, ret):
    """self.dfs(A, 0, k, [], target, ret)"""
    if len(cur) == k and remain == 0:
        ret.append(list(cur))
        return

    if (i >= len(A) or len(cur) > k
        or len(A)-i+len(cur) < k):
        return

    self.dfs(A, i+1, k, cur, remain, ret)
    cur.append(A[i])
    self.dfs(A, i+1, k, cur, remain-A[i], ret)
    cur.pop()
```

19.3 STRING

19.3.1 Palindrome

19.3.1.1 Palindrome partition.

Given $s = \text{"aab"}$, return:

$[[\text{"aa"}, \text{"b"}], [\text{"a"}, \text{"a"}, \text{"b"}]]$

Core clues:

1. Expand the search tree **horizontally**.

Search process:

input: "aabb"

```
"a", "abbc"
  "a", "bbc"
    "b", "bc"
      "b", "c" (o)
      "bc" (x)
    "bb", "c" (o)
    "bbc" (x)
  "ab", "bc" (x)
  "abb", "c" (x)
  "abbc" (x)
"aa", "bbc"
  "b", "bc"
    "b", "c" (o)
    "bc" (x)
  "bb", "c" (o)
  "bbc" (x)
"aab", "bc" (x)
"aabb", "c" (x)
```

Code:

```
def partition(self, s):
    ret = []
    self.backtrack(s, [], ret)
    return ret

def backtrack(self, s, cur_lvl, ret):
    """
    Let i be the scanning ptr.
    If s[:i] passes predicate, then backtrack s[i:]
    """
    if not s:
        ret.append(list(cur_lvl))

    for i in xrange(1, len(s)+1):
        if self.predicate(s[:i]):
            cur_lvl.append(s[:i])
            self.backtrack(s[i:], cur_lvl, ret)
            cur_lvl.pop()

def predicate(self, s):
    return s == s[::-1]
```

19.4 MATH

19.4.1 Decomposition

19.4.1.1 Factorize a number

Core clues:

1. Expand the search tree **horizontally**.

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

[4, 4]

Code:

```
def dfs(self, cur, ret):
    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 self.predicate(n, i):
            cur.append(i)
            cur.append(n/i)
            self.dfs(cur, ret)
            cur.pop()

def predicate(self, n, i):
    return n%i == 0
```

Time complexity. The search tree's size is $O(2^n)$ where n is the number of prime factors. Choose i prime factors to combine then, and keep the rest uncombined

$$\sum_i \binom{n}{i} = 2^n$$

19.5 ARITHMETIC EXPRESSION

19.5.1 Unidirection

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 → ["1 + 2 + 3", "1 * 2 * 3"]
"232", 8 → ["2 * 3 + 2", "2 + 3 * 2"]

Clues:

1. Backtracking with *horizontal* expanding
2. Special handling for multiplication - caching the expression *predecessor* for multiplication association.
3. Detect *invalid* number with leading 0's

```
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: # 1st number
                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)
```

19.5.2 Bidirection

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
```

19.6 TREE

19.6.1 BST

19.6.1.1 Generate Valid BST

Generate all valid BST with nodes from 1 to n .

Core clues:

1. Iterate pivot
2. Generate left and right

Code:

```
def generate(self, start, end):
    roots = []
    if start > end:
        roots.append(None)
        return roots

    for pivot in range(start, end+1):
        left_roots = self.generate_cache(start, pivot-1)
        right_roots = self.generate_cache(pivot+1, end)

        for left_root in left_roots:
            for right_root in right_roots:
                root = TreeNode(pivot)
                root.left = left_root
                root.right = right_root

                roots.append(root)

    return roots
```


Chapter 20

Graph

20.1 BASIC

Graph representation. V for a vertex set with a map, mapping from vertex to its neighbors. The mapping relationship represents the edges E .

$V = \text{defaultdict}(\text{list})$

Complexity. Basic complexities:

Algorithm	Time	Space
dfs	$O(E)$	$O(V), O(\text{longest path})$
bfs	$O(E)$	$O(V)$

```
def dfs(self, grid, i, j, visited):
    m = len(grid)
    n = len(grid[0])
    visited[i][j] = True

    for dir in self.dirs:
        I = i+dir[0]
        J = j+dir[1]
        if (0 <= I < m and 0 <= J < n and
            not visited[I][J] and grid[I][J] == "1"):
            self.dfs(grid, I, J, visited)
```

If the islands are constantly updating and the query for number of islands is called multiple times, need to use union-find (Section 20.6) to reduce each query's complexity from $O(mn)$ to $O(\log mn)$.

20.2 DFS

Number of Islands. The most fundamental and classical problem.

```
11000
11000
00100
00011
Answer: 3
```

Clue:

1. Iterative dfs

```
class Solution(object):
    def __init__(self):
        self.dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    def numIslands(self, grid):
        cnt = 0
        visited = [[False for _ in xrange(n)]
                    for _ in xrange(m)]
        for i in xrange(m):
            for j in xrange(n):
                if not visited[i][j] and grid[i][j] == "1":
                    self.dfs(grid, i, j, visited)
                    cnt += 1

        return cnt
```

20.3 BFS

20.3.1 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' 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:

```
self.dirs = ((-1, 0), (1, 0), (0, -1), (0, 1))

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))
```

20.4 DETECT ACYCLIC

1. **marked** is reset after a dfs.
2. **visited** should be updated 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**. $A - B$.
 - b. Excluding neighbors in **visited** is redundant, due to **pi**.

20.4.1 Directed Graph

Detect cycles (any) in directed graph.

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

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

    pathset.remove(v)
    visited.add(v)
    return True
```

20.4.2 Undirected Graph

Detect cycles (any) in undirected graph.

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

    pathset.add(v)
    for nbr in V[v]:
        if nbr != pi:
            if not self.dfs(V, nbr, v, visited, pathset):
                return False

    pathset.remove(v)
    visited.add(v)
    return True
```

20.5 TOPOLOGICAL SORTING

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

20.5.1 Algorithm

Core clues:

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

Notice:

1. Need to check ascending order or descending order.
2. Need to **detect cycle**; thus the dfs need to construct result queue and detect cycle simultaneously, by using two sets: *visited* and *pathset*.

```

from collections import deque

def topological_sort(self, V):
    visited = set()
    ret = deque()

    for v in V.keys():
        if v not in visited:
            if not self.dfs_topo(V, v, visited, set(), ret):
                return [] # contains cycle

    return list(ret)

def dfs_topo(self, V, v, visited, pathset, ret):
    if v in pathset:
        return False

    pathset.add(v)
    for nbr in V[v]:
        if nbr not in visited:
            if not self.dfs_topo(V, nbr, visited, pathset, ret):
                return False

    pathset.remove(v)
    visited.add(v)
    ret.appendleft(v)
    return True

```

20.5.2 Applications

1. Course scheduling problem with pre-requisite.

20.6 UNION-FIND

Improvements:

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

20.6.1 Algorithm

Weighted union-find with path compression.

Core clues.

1. **π array**: an array to store each item's predecessor π . The predecessor are lazily updated to its ancestor. When $x == \pi[x]$, then x is the ancestor (i.e. root).
2. **Size-balanced**: merge the tree according to the size to maintain balance.
3. **Path compression**: Make the ptr in π array to point to its root rather than its immediate parent.

worst-case input

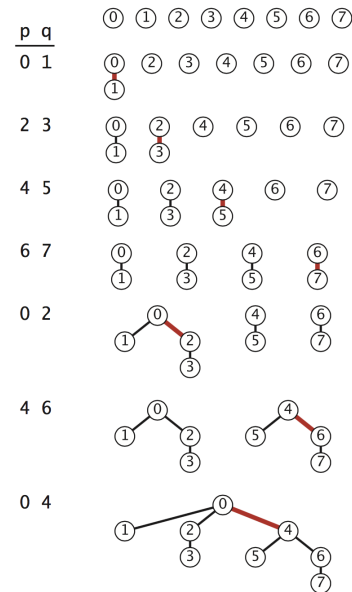


Fig. 20.1: Weighted quick-union traces

```

class UnionFind(object):
    def __init__(self):
        self.pi = {} # item -> pi
        self.sz = {} # root -> size

    def __len__(self):
        """number of unions"""
        return len(self.sz) # only root nodes have size

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

    def root(self, x):
        """path compression"""
        pi = self.pi[x]
        if x != pi:
            self.pi[x] = self.root(pi)
        return self.pi[x]

    def unionize(self, x, y):
        pi1 = self.root(x)
        pi2 = self.root(y)

        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 isunion(self, x, y):
        if x not in self.pi or y not in self.pi:
            return False
        return self.root(x) == self.root(y)

```

20.6.2 Complexity

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

20.7 AXIS PROJECTION

Project the mat dimension from 2D to 1D, using *orthogonal axis*.

Smallest bounding box. Given the location (x, y) of one of the 1's, return the area of the smallest bounding box that encloses 1's.

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

Clues:

1. Project the 1's onto x-axis, binary search for the left bound and right bound of the bounding box.
2. Do the same for y-axis.

Time complexity: $O(m \log n + n \log m)$, where $O(m), O(n)$ is for projection complexity.

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 - the state

1. ends *at* index i
2. ends *before* index i
3. ends *at or 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,

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

should be reduced to

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

Secondly,

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

should be reduced to

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

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

21.2.1 Single-state dp

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.

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

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 $gmax = \max(\{F_{i,j}\})$.

Longest increasing subsequence. Find the longest increasing subsequence of an array A .

let F_i be the LIS length ends at A_i .

$$F_i = \max(F[j] + 1 \cdot \forall j < i) // \text{ if } A_i > A_j$$

Then the global $maxa$ is:

$$maxa = \max(F_i \cdot \forall i)$$

Time complexity: $O(n^2)$

In code, notice the **else**.

```
F[i] = max(
    F[j] + 1 if A[i] > A[j] else 1
    for j in xrange(i)
)
```

Alternative solution using binary search in $O(n \log n)$ - Section 9.3.1.

Maximum subarray sum. Find the maximum subarray sum of A .

Let F_i be the maximum subarray sum ending at A_i

$$F_i = \max(F_{i-1} + A_i, 0)$$

Then the global *maxa* is:

$$maxa = \max(F_i \cdot \forall i)$$

Maximum sum of non-adjacent cells. Get the maximum sum of non-adjacent cells of an array A .

Let F_i be the maximum sum of non-adjacent cells for $A[:i]$. You have two options: choose A_{i-1} or not.

$$F_i = \max(F_{i-1}, F_{i-2} + A_{i-1})$$

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 $B[:j]$.

$$F_{i,j} = \begin{cases} F_{i-1,j-1} & // \text{ if } a[i] == b[j] \\ \min \begin{pmatrix} F_{i,j-1} + 1, & // \text{ otherwise, insert} \\ F_{i-1,j} + 1, & // \text{ delete} \\ F_{i-1,j-1} + 1 \end{pmatrix} & // \text{ replace} \end{cases}$$

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).

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

21.2.2 Dual-state dp

Maximal product subarray. Find the subarray within an array A which has the largest product.

- Let $small_i$ be the smallest product end with A_i .
- Let $large_i$ be the largest product end with A_i .
- The states can be negative.

$$small_i = \min(A_i, small_{i-1} \cdot A_i, large_{i-1} \cdot A_i)$$

$$large_i = \max(A_i, small_{i-1} \cdot A_i, large_{i-1} \cdot A_i)$$

It can be optimized to use space $O(1)$.

Trapping Rain Water Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.



Fig. 21.1: Trapping Rain Water

Let $maxL_i$ be the $\max(A[:i])$; let $maxR_i$ be the $\max(A[i:n])$. The dp of obtaining max is trivial.

The total volume *vol*:

$$vol = \sum_i (\max(0, \min(maxL_i, maxR_{i+1}) - A[i]))$$

Zigzag subsequence. Find the max length zigzag subsequence which goes up and down alternately within the array A .

Let U_i be the max length of zigzag subsequence end at $A_i \wedge$ going up.

Let D_i be the max length of zigzag subsequence end at $A_i \wedge$ going down.

$$U_i = \max(D_j + 1 \cdot \forall j < i) // \text{ if } A_i > A_j$$

$$D_i = \max(U_j + 1 \cdot \forall j < i) // \text{ if } A_i < A_j$$

Notice in python implementation, don't use list comprehension since two states are interleaved and interdependent.

21.3 STRING

Word break. Given a string s and a dictionary of words $dict$, determine if s can be segmented into a space-separated sequence of $dict$ words.

Let F_i be whether $s[:i]$ can be segmented.

$$F_i = \begin{cases} F_{i-len(w)} & // \text{ if } \exists w \in dict, s[i-len(w):i] == w \\ false & // \text{ otherwise} \end{cases}$$

Return all such possible sentences. In original case, we use a bool array to record whether a dp could be segmented. Now we should use a vector for every dp to record how to construct that dp from another dp.

Let F_i be all possible segmented words ends at $s[i-1]$. F_i is a list. $\exists F_i$ means F_i is not empty.

$$F_i = \begin{cases} F_i + [w] // \forall w \in dict. \\ \quad \text{if } s[i-\text{len}(w):i] = w \wedge \exists F_{i-\text{len}(w)} \\ F_i // \text{otherwise} \end{cases}$$

Reconstruct the sentence from F_i . It is like building path for the tree. Using backtracking:

```
def build(self, dp, i, cur, ret):
    if cur_index == 0:
        ret.append(" ".join(list(cur)))
        return

    # backtracking
    for word in dp[i]:
        cur.appendleft(word)
        self.build(dp, i-len(word), cur, ret)
        cur.popleft()
```

Is palindrome. Given a string s , use an array to determine whether $s[i:j]$.

Let $P_{i,j}$ indicates whether $s[i:j]$ is palindrome. We have one condition - whether the head and the end letter are equal:

$$P_{i,j} = P_{i-1,j+1} \wedge s[i] = s[j-1]$$

The code for palindrome dp is error-prone due to indexing. Notice that $i \in [0, n), j \in [i, n+1]$.

```
n = len(s)
pa = [[False for _ in xrange(n+1)] for _ in xrange(n)]
for i in xrange(n):
    pa[i][i] = True
    pa[i][i+1] = True

for i in xrange(n-2, -1, -1):
    for j in xrange(i+2, n+1):
        pa[i][j] = pa[i+1][j-1] and s[i] == s[j-1]
```

Minimum palindrome cut. Given a string s , partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s .

Let C_i be the min cut for $s[:i]$. We have 1 more cut from previous state to make $S[:i]$ palindrome.

$$C_i = \begin{cases} \min(C[k] + 1 \cdot \forall k < i) // \text{if } s[k:i] \text{ is palindrome} \\ 0 // \text{otherwise} \end{cases}$$

```
def minCut(self, s):
    n = len(s)

    P = [[False for _ in xrange(n+1)] for _ in xrange(n+1)]
    for i in xrange(n+1): # len 0
        P[i][i] = True
    for i in xrange(n): # len 1
        P[i][i+1] = True
```

```
for i in xrange(n, -1, -1): # len 2 and above
    for j in xrange(i+2, n+1):
        P[i][j] = P[i+1][j-1] and s[i] == s[j-1]

C = [i for i in xrange(n+1)] # max is all cut
for i in xrange(n+1):
    if P[0][i]:
        C[i] = 0
    else:
        C[i] = min(
            C[j] + 1
            for j in xrange(i)
            if P[j][i]
        )

return C[n]
```

ab string. Change the char in a str only consists of 'a' and 'b' to non-decreasing order. Find the min number of char changes.

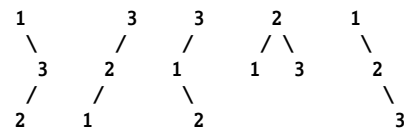
Two-state dp: 'a' \rightarrow 'b' and 'b' \rightarrow 'a'. 1 cut into 2 segment.

abc string. Follow up for ab string. Three-state dp: $chr \neq a, chr \neq b, chr \neq c$. 2 cuts into 3 segments.

21.4 COMBINATORICS

21.4.1 Tree

Number of different BSTs. It can be solved using Catalan number (Section 15.4), but here goes the dp solution.



Let F_i be the #BSTs constructed from i elements. The pattern is:

$$F_3 = F_0 * F_2 + F_1 * F_1 + F_2 * F_0$$

Thus, in general,

$$F_i = \sum (F_j * F_{i-1-j} \cdot \forall j < i)$$

21.5 BACKPACK

21.5.1 Classical

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.

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

Advanced backpack problem³.

21.5.2 Sum

k sum. Given n distinct positive integers, integer k ($k \leq n$) and a number target. Find k numbers where sum is target. Calculate the number of solutions. Since we only need the number of solutions, thus it can be solved using dp. If we need to enumerate all possible answers, need to do dfs instead.

$$\text{sum} \left(\begin{matrix} j \\ i \end{matrix} \right) = v$$

Let $F_{i,j,v}$ means the #ways of selecting i elements from the first j elements so that their sum equals to v . j is the scanning pointer.

You have two options: either select A_{j-1} or not.

$$F_{i,j,v} = F_{i,j-1,v-A_{j-1}} + F_{i,j-1,v}$$

Time complexity: $O(n^2k)$

21.6 LOCAL AND GLOBAL EXTREMES

21.6.1 Long and short stocks

The following formula derives from the question: Best Time to Buy and Sell Stock IV. Say you have an array for which the i -th element is the price of a given stock on day i . Design an algorithm to find the maximum profit. You may complete at most k transactions.

Let $local_{i,j}$ be the max profit with j transactions with last transactions **ended at** day i . Let $global_{i,j}$ be the max profit with transactions **ended at or before** day i with j transactions.

To derive transition function for $local$, for any given day i , you have two options: 1) transact in one day; 2) hold the stock one more day than previous and then transact. The latter option is equivalent to revert yesterday's transaction and instead transact today.

To derive transition function for $global$, for any given day i , you have two options: 1) transact today; 2) don't transact today.

$$local_{i,j} = \max \left(global_{i-1,j-1} + \Delta, local_{i-1,j} + \Delta \right)$$

$$global_{i,j} = \max \left(local_{i,j}, global_{i-1,j} \right)$$

, where Δ is the price change (i.e. profit) at day i .

Notice:

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

Space optimization.

$$local_j = \max \left(global_{j-1} + \Delta, local_j + \Delta \right)$$

$$global_j = \max \left(local_j, global_j \right)$$

Notice,

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

Alternative definitions. Other possible definitions: let $global_{i,j}$ be the max profit with transactions ended at or before day i with **up to** j transactions. Then,

$$local_{i,j} = \max \left(global_{i-1,j-1} + \max(0, \Delta), local_{i-1,j} + \Delta \right)$$

$$global_{i,j} = \max \left(local_{i,j}, global_{i-1,j} \right)$$

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

The complexity of the alternative definitions is the same as the original definitions. The bottom line is that different definitions of states result in different transition functions.

³ [Nine Lectures in Backpack Problem.](#)

21.7 GAME THEORY - MULTI PLAYERS

Assumption: the opponent take the optimal strategy for herself.

21.7.1 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.

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

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.

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

Chapter 22

Interval

22.1 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.

$$| \leftarrow p$$

2. Merge the intermediate intervals with the new interval. Need to mathematically prove it works as expected.

```
def insert(self, itvls, newItvl):
    s, e = newItvl.start, newItvl.end
    left = filter(lambda x: x.end < s, itvls)
    right = filter(lambda x: x.start > e, itvls)
    if len(left)+len(right) != len(itvls):
        s = min(s, itvls[len(left)].start)
        e = max(e, itvls[-len(right)-1].end)

    return left + [Interval(s, e)] + right
```

22.2 OPERATIONS

Merge intervals. Given a collection of intervals, merge all overlapping intervals.

Core clues:

1. Sort the intervals
2. When does the overlapping happens? $[0, 5]$ vs. $[2, 6]$; $[0, 5]$ vs. $[2, 4]$

```
def merge(self, itvls):
    if not itvls:
        return []

    itvls.sort(key=lambda x: x.start)
    ret = [itvls[0]]
    for cur in itvls[1:]:
        pre = ret[-1]
        if cur.start <= pre.end: # overlap
            pre.end = max(pre.end, cur.end)
        else:
            ret.append(cur)

    return ret
```

Insert intervals. Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary). Assume that the intervals were initially sorted according to their start times.

Core clues

1. Partition the original list of intervals to left-side intervals and right-side intervals according to the new interval.

22.3 EVENT-DRIVEN ALGORITHM

22.3.1 Introduction

The core philosophy of event-driven algorithm:

1. **Event:** define *event*; the event are sorted by time of appearance.
2. **Heap:** define *heap meaning*.
3. **Transition:** define *transition functions* among events impacting the heap.

22.3.2 Questions

Maximal overlaps. Given a list of number intervals, find max number of overlapping intervals.

Core clues:

1. **Event:** Every new start of an interval is an event. Scan the sorted intervals (sort the interval by *start*).
2. **Heap meaning:** Heap stores the *end* of the interval.
3. **Transition:** 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"))
    h_end = []
    for itvl in intervals:
        heapq.heappush(h_end, itvl.end)

        while h_end and h_end[0] <= itvl.start:
            heapq.heappop(h_end)

        maxa = max(maxa, len(h_end))

    return maxa
```

Chapter 23

General

23.1 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>`).

Solving unseen problems. Solving unseen problems is like a search problems. You need to explore different options, either with dfs or bfs.

Glossary

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.

subarray Continuous subarray $A[i : j]$

subsequence Non-continuous ordered subsequence that $S \subset A[i : j]$.

invariant An invariant is a condition that can be relied upon to be true during execution of a program. A loop invariant is a condition that is true at the beginning and end of every execution of a loop.

Abbreviations

A Array

idx Index

TLE Time Limit Exceeded

MLE Memory Limit Exceeded

dp Dynamic programming

def Definition

ptr Pointer

len Length

asc Ascending

desc Descending

pred Predecessor

succ Successor

π /**pi** The parent of a child

bfs Breadth-first search

dfs Depth-first search

mat Matrix

ADT Abstract Data Type

aka Also known as