github.com/algorhythms/Algo-Quicksheet

Author: idf@github

Algorithm Quicksheet

Classical equations, diagrams and patterns in algorithms

November 8, 2024

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

High-level abstraction is the key. 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.

The codes are just the details of implementation. Remembering them is simply unproductive and non-scalable. Only dive into the codes when you are unable to reconstruct the algorithm from the hits and clues.

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

Contents

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
$\log x$	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
$a \mid b$	Bit OR
$a \wedge b$	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 $y \propto x$
$\lvert x \rvert$	Absolute value
$\lvert\lvert \vec{a} \rvert\rvert$	L_2 distance (Euclidean distance) of a vector; norm-2
$\lvert \mathcal{S} \rvert$	Size (cardinality) of a set
$n!$	Factorial function
$\triangleq$	Defined as
$O(\cdot)$	Big-O notation, complexity upper bound
$\mathbb{R}$	The real numbers
$0:n$	Range (Python convention): $0:n = 0, 1, 2, ..., n-1$
$\approx$	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 $\frac{n!}{k!(n-k)!}$
$\mathrm{range}(i, j)$	Range of number from i (inclusive) to j (exclusive)
$A[i:j]$	Subarray consist of $A_i, A_{i+1}, ..., A_{j-1}$.

Chapter 1
Time Complexity

1.1 BASIC COUNTS

Double for-loops

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

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

Triple for-loops

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

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

1.2 SOLVING RECURRENCE EQUATIONS

Basic recurrence equation solving techniques:

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

1.2.1 Master Theorem

Recurrence relations:

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

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

Case 1, dominated by the 1st term

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, non-dominance

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, dominated by the 2nd term

If:

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

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

And with regularity condition:

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

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

Then:

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

1.3 USEFUL MATH EQUATIONS

Euler.

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

Logarithm power.

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

Proof:

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

Discrete to continuous. if $f(x)$ is monotonously decreasing, then

$$\int_1^{+\infty} f(x)\,\mathrm{d}x \le \sum_{i=1}^{+\infty} f(i) \le f(1) + \int_1^{+\infty} f(x)\,\mathrm{d}x$$

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

Table 2.1: for primitive types

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

Table 2.2: for one-dimensional arrays

Notice:

1. The reference takes memory of 8 bytes.
2. Reference includes object reference and innner 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 as a 8's multiple.

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

Table 2.3: for two-dimensional arrays

Type	Bytes
Object overhead	16
Reference	8
Padding	8x

Table 2.4: for objects

2.1.2 Example

The generics is passed as Boolean:

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

Notice the multiple levels of references.

Chapter 3
Basic Data Structures

3.1 Introduction

Abstract Data Types (ADT):

1. Queue
2. Stack
3. HashMap

Implementation (for both queue and stack):

1. Linked List
2. Resizing Array:

   a. Doubling: when full (100%).
   b. Halfing: when one-quarter full (25%).

Python Library:

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

Java Library:

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

3.2 Python Library

```
from collections import defaultdict
counter = defaultdict(int)
G = defaultdict(list)  # graph of v -> neighbors
G = defaultdict(dict)  # G[s][e] -> weight
```

```
from collections import deque
path = deque()
path.appendleft("a")
path.append("b")
path.popleft()
path.pop()
path = deque(sorted(path))
```

```
import heapq
l = []  # list
heapq.heappush(l, "a")
heapq.heappop(l)
heapq.heapify(l)
heapq.heappushpop(l, "b")
heapq.nsmallest(3, l)
```

[1] The lowercase and uppercase naming in Python collections are awkward: discussion.

3.3 Stack

3.3.1 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.

Stack can convert recursive DFS to iterative.

3.3.2 Usage

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

The **relationship invariants** can be:

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

3.3.3 Unpaired Parentheses

Longest Valid Parentheses. Given a string containing just the characters ‘(’ and ‘)’, find the length of the longest valid (well-formed) parentheses substring. Core clues:

1. **Invariant**: Stack holds the INDEX of UNPAIRED brackets, either (or).
2. Thus, `stk[-1]` stores the last unpaired bracket.
3. The length of the well-formed parentheses is: if currently **valid**, current scanning index `idx` minus the last **invalid** index of bracket `stk[-1]`

```
def longestValidParentheses(self, s):
```

```
stk = []
maxa = 0
for idx, val in enumerate(s):
  if val == ")" and stk and s[stk[-1]] == "(":
    stk.pop()
    if not stk:
      maxa = max(maxa, idx+1)
    else:
      maxa = max(maxa, idx-stk[-1])
  else:
    stk.append(idx)

return maxa
```

3.3.4 Nearest smaller value

Nearest smaller value for each. 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. Nearest $\equiv$ spatial locality.
2. **Invariant**: Maintain a *strictly increasing* stack.
3. If the question asks for 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
```

3.3.5 Non-Decreasing Stack

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.

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.

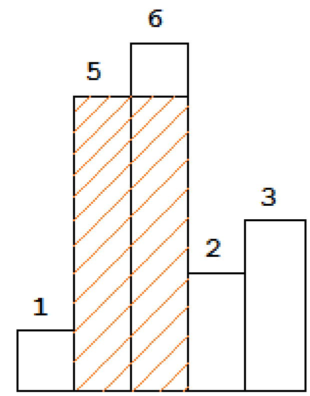


Fig. 3.1: Largest rectangle in histogram
$i \rightarrow$ height 2, *cur* $\rightarrow$ height 6, 5, 1

Core clues:

1. **Invariant**: Maintain the non-decreasing stack, using INDEX.
2. Popping triggers the calculation of area
3. Calculate the rectangle width by index diff
4. Post-processing in the end

Code:

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

    for i in range(n):
        while stk and height[stk[-1]] > height[i]:
            cur = stk.pop()
            # calculate area when popping
            if stk:
                # lower bound stk[-1] + 1
                # higher bound i
                l = i-(stk[-1]+1)
            else:
                l = i

            area = height[cur]*l
            gmax = max(gmax, area)

        stk.append(i)

    # after array scan, process the dangling stack
    i = n
    ...

    return gmax
```

3.4 MAP

3.4.1 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)
```

3.4.2 Operations

Sorting by value. Sort the map entries by values `itemgetter`.

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

In majority case, we need a reference to pre.

4.1.2 Basic Operations

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

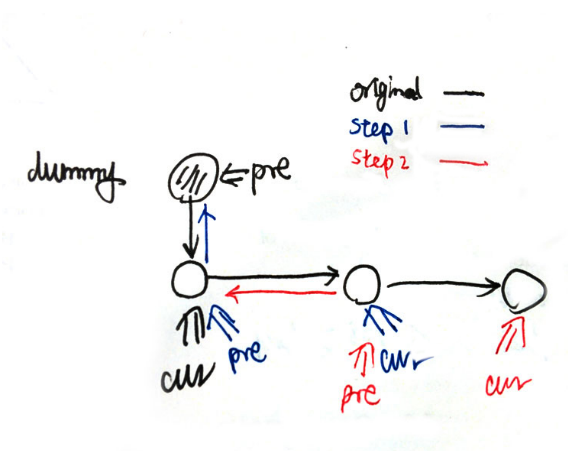


Fig. 4.1: Reverse the linked list

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

    pre = dummy
    cur = head  # ... = dummy.next, not preferred
    while pre and cur:
        assert pre.next == cur
        nxt = cur.next
        # op
        cur.next = pre

        pre = cur
        cur = nxt

        # original head pointing to dummy
    head.next = None  # dummy.next.next = ..., not preferred
    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
```

```python
        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  # safe
        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  # safe

        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: $A_{2i} \leq A_i \geq A_{2i+1}$.

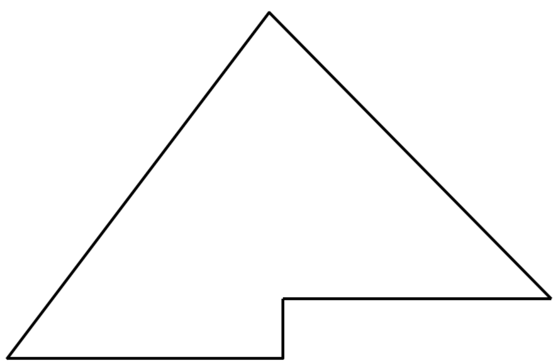


Fig. 5.1: Heap

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 (because we want to maintain the heap invariant).

```
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
```

We can return the `idx` at the end to report the final index of the element.

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 range(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 $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of any height h.

Proof:

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

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

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 $a[3k+1]$.

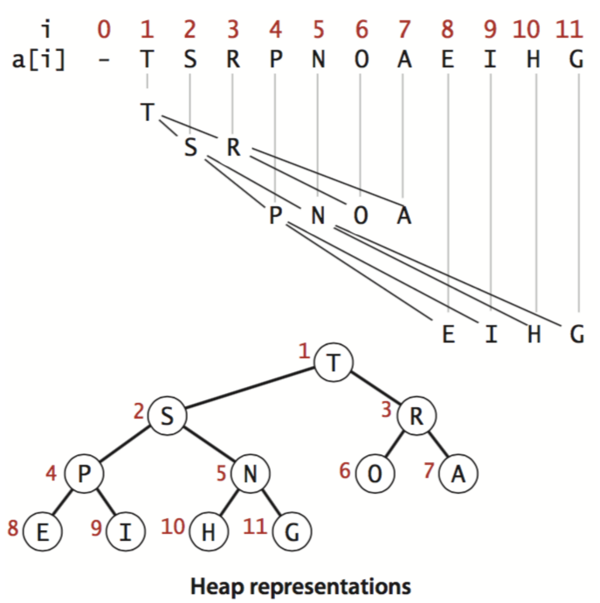


Fig. 5.2: Heap representation

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

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

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

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

5.3.3 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
);
```

5.4 DERIVATIVES

5.4.1 Heap of Linked Lists

Maintain a heap of linked lists, pop the min head, and push the head's next back to the heap.

Chapter 6
Tree

6.1 Binary Tree

6.1.1 Basic Operations

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:

```
# delete minimum node in the BST
def delete_min(x: Node) -> Node:
    if not x.left:
        return x.right
    x.left = delete_min(x.left)
    return x
```

Max depth. DFS solution

```
def probe(self, cur, depth):
    if not root:
        return depth
    else:
        return max(
            self.probe(cur.left, depth+1),
            self.probe(cur.right, depth+1),
      )
```

Min depth. Definition of min depth, lowest depth of leaf node.

Notice, that the additional checks are necessary of missing either right or left child.

```
def probe(self, cur, depth):
    if not cur:
        return depth
    elif cur.right and not cur.left:
        return self.probe(cur.right, depth+1)
    elif cur.left and not cur.right:
        return self.probe(cur.left, depth+1)
    else:
        return min(
            self.probe(root.left, depth+1),
            self.probe(root.right, depth+1),
        )
```

Height. The height of a node is the number of edges from the node to the deepest leaf.

```
def dfs(self, cur):
    if not cur:
        return -1  # leaves index start from 0

    height = 1 + max(
        self.dfs(cur.left),
        self.dfs(cur.right),
    )

    # do something

    return height
```

Application: leaf by leaf traversal by height.

```
def dfs(self, cur, leaves):
    if not cur:
        return -1  # leaves index start from 0

    height = 1 + max(
        self.dfs(cur.left, leaves),
        self.dfs(cur.right, leaves),
    )
    if height >= len(leaves):
        leaves.append([])  # grow

    leaves[height].append(cur.val)
    return height
```

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

```
def dfs(self, cur, target, path, found: List[bool]):
    # post function call check
    if not cur:
      return
    if found[0]:
      return

    path.append(cur)
    if cur == target:
        found[0] = True

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

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

Lowest common ancestor. In BST, the searching is straightforward.

```
def find_lca(self, cur, p, q):
    if p.val > cur.val and q.val > cur.val:
        return self.find_lca(cur.right, p, q)
    if p.val < cur.val and q.val < cur.val:
        return self.find_lca(cur.left, p, q)
    return cur
```

Method 1: In normal binary tree, construct the path from root to $node_1$ and $node_2$ respectively, and **diff** the two paths. Time complexity: $O(\lg n)$, space complexity: $O(\lg n)$.

Method 2: If the parent pointer is provided, it is possible to reduce the space complexity to $O(1)$, by using two pointers:

```
def find_lca(n1, n2):
    if not n1 or not n2:
        return None

    d1, d2 = depth(n1), depth(n2)
    if d2 < d1:
        return find_lca(n2, n1)  # swap

    # move to the same depth
    for _ in range(d2-d1):
        n2 = n2.parent

    while n1 and not n1 == n2:
        n1 = n1.parent
        n2 = n2.parent

    return n1
```

Method 3: In normal bindary tree and O(1) space.

```
def count(self, node, p, q):
    if not node:
        return 0

    lcount = self.count(node.left, p, q)
    rcount = self.count(node.right, p, q)
    mcount = 1 if node in (p, q) else 0
    ret = lcount + rcount + mcount
    # lcount == 1 and rcount == 1 #
    # or lcount == 1 and mcount == 1 or ...
    if ret == 2:
        self.ans = node
    return ret
```

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(self, cur, path, ret):
    if not cur:
        return

    path.append(cur)
    if not cur.left and not cur.right:
        ret.append("->".join(map(repr, path)))

    self.dfs(cur.left, path, ret)
    self.dfs(cur.right, path, ret)
    path.pop()  # 1 append 1 pop
```

Leftmost node. Find the leftmost node.

```
def leftmost(self, cur, l):
    """
    :param l: offset from center 0, negative means left side.
    """
    if not cur:
        return l
    return min(
        self.leftmost(cur.left, l-1),
        self.leftmost(cur.right, l+1),
    )
```

Rightmost node can be similarly found.

Diameter of a tree (graph). The diameter of a tree $\equiv$ longest path in the tree. Intuitily, we need to find one end of the edge, and then find the other end.

Core clues:

1. **Find one end of the edge.** Start from any vertex, bfs to reach the farthest leaf.
2. **Find the other end.** Start from this leaf node, bfs to reach the other farthest leaf.

```
_, _, last = self.bfs(0, V)
level, pi, last = self.bfs(last, V)

def bfs(self, s, V) -> int, List[Vertex], Vertex:
    # bfs
    visited = [False for _ in range(len(V))]
    # predecessor
    pi = [-1 for _ in range(len(V))]
    last = s
    level = 0
    q = [s]
    while q:
        l = len(q)
        for i in range(l):
            cur = q[i]
            last = cur
            visited[cur] = True
            for nbr in V[cur]:
                if not visited[nbr]:
                    pi[nbr] = cur
                    q.append(nbr)

        q = q[l:]
        level += 1

    return level, pi, last
```

, where V is the vertices of graph G.

6.1.2 Morris Traversal

Traversal with O(1) space. [2] Three ways of traversal: inorder, pre-order, post-order. Time complexity $O(3n)$. Find `pre` twice, traverse `cur` once.

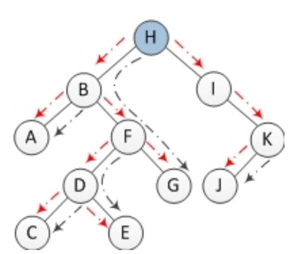


Fig. 6.1: Morris traversal time complexity

[2] ref

Core:

1. Threading from **in-order** predecessor to `cur`.
2. In-order consumes the `cur` when going right, pre-order when going left, post-order consumes the left subtree path when going right.

6.1.2.1 In-order

Given the current node `cur`, we know the next child node. But how does its predecessor knows `cur`? 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. (≡ no left).

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)  # when pop the thread
                cur = cur.right
```

6.1.2.2 Pre-order

Similar to in-order. Pre-order consume the current node when setting the thread rather than removing the thread as in in-order.

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. (≡ no left, but no *consume*, since consume before).

Code:

```
def morris_preorder(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
                self.consume(cur)  # when set the thread
                cur = cur.left
            else:
                pre.right = None
                cur = cur.right
```

6.1.2.3 Post-order

More tedious but solvable. The process is also similar to in-order.

Process:

1. Set a temporary var `dummy.left = root`
2. If no left, go right
3. If left, find the in-order predecessor `pre` in left tree
   a. If no thread, set `right = cur` thread; go left.
   b. If thread, set `right = None`, reversely *consume* the path from `cur.left` to `pre`; go right.

Code:

```
def morris_postorder(self, root):
    dummy = TreeNode(0)
    dummy.left = root
    cur = dummy
    while cur:
        if not cur.left:
            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_path(cur.left, pre)
                cur = cur.right

def _reverse(self, fr, to):
    """Like reversing linked list"""
    if fr == to: return
    cur = fr
    nxt = cur.right
    while cur and nxt and cur != to:
        nxt.right, cur, nxt = cur, nxt, nxt.right

def consume_path(self, fr, to):
```

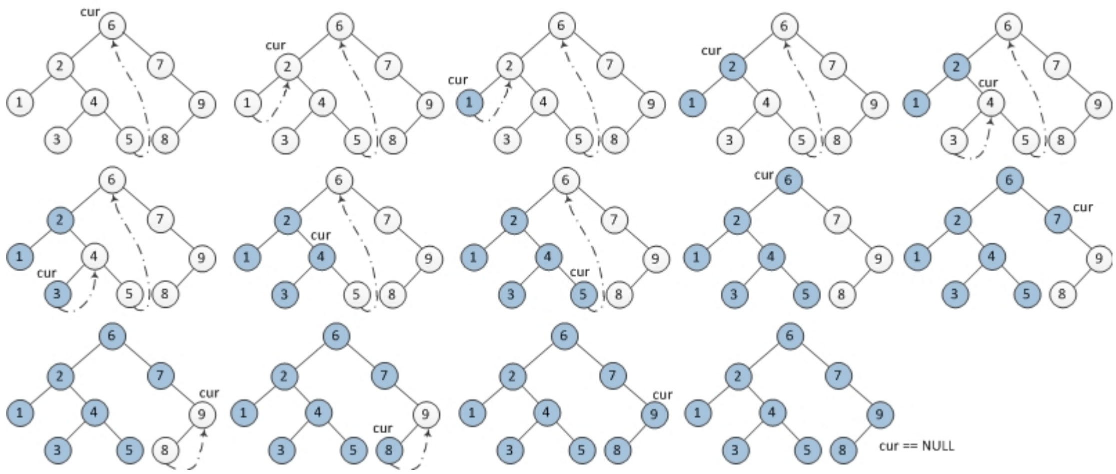


Fig. 6.2: Morris in-order traversal

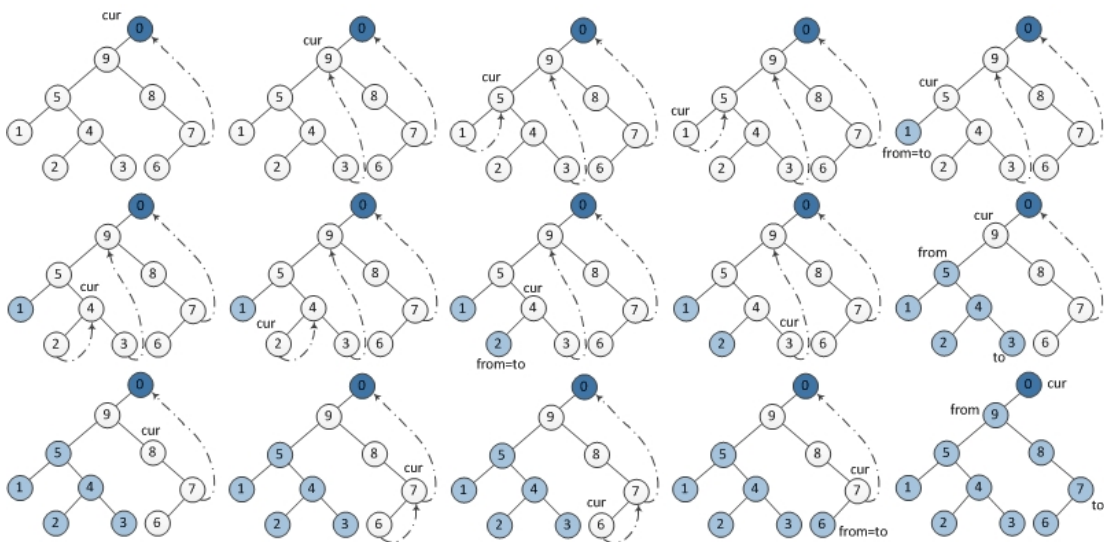


Fig. 6.3: Morris post-order traversal

```
self._reverse(fr, to)

cur = to
self.consume(cur)
while cur != fr:
    cur = cur.right
    self.consume(cur)

self._reverse(to, fr)
```

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 Property

∀ node, the node value is larger than the largest value in its left subtree; and is smaller than the smallest value in the righhht subtree:

$$\max(node.left) \leq node.val \leq \min(node.right)$$

Leftmost node is the smallest node of the tree; rightmost node is the largest node of the tree.

Find such info takes $O(n \lg n)$ for all subtrees; and we can cache such info into the following data structure to achieve $O(n)$.

```
class BSTInfo(object):
    def __init__(self, sz, lo, hi):
        self.sz = sz
        self.lo = lo
        self.hi = hi
```

6.2.2 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.

6.2.3 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)
```

Closet 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

A Fenwick tree or binary indexed tree is a data structure that can efficiently update elements and calculate prefix sums in a table of numbers.

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, Segment Tree can but BIT cannot query

$$prefixSum \rightarrow i$$

6.3.2 Implementation

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

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

, 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 complement. 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 $lowbit$. 0 is the dummy root.

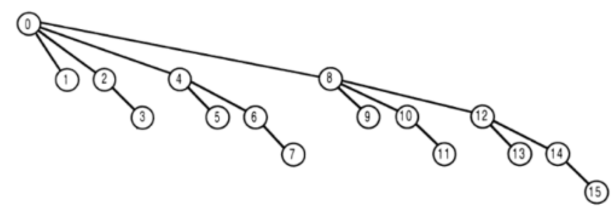


Fig. 6.4: Binary Indexed Tree *get* Operation

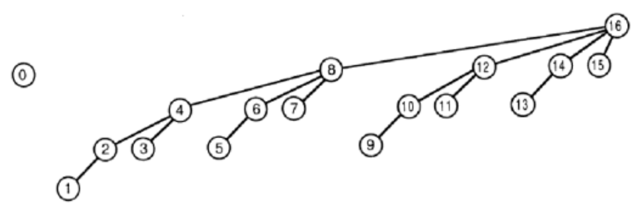


Fig. 6.5: Binary Indexed Tree *set* Operation

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
        0 is the dummy root
```

```
        """
        self.N = [0 for _ in range(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, val):
        self.lo, self.hi, self.val = lo, hi, val
        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, lo, hi):
        """
        Bottom-up build
        segment: [lo, hi)
        Either check lo==hi-1 or have root.right
        only if have root.left
        """
        if lo >= hi:   return None
        if lo == hi-1: return Node(lo, hi, self.A[lo])

        left  = self.build_tree(lo, (lo+hi)/2)
        right = self.build_tree((lo+hi)/2, hi)

        val = DEFAULT
        if left:  val = f(val, left.val)
        if right: val = f(val, right.val)
        root = Node(lo, hi, val)
        root.left  = left
        root.right = right

        return root

    def query(self, root, lo, hi):
        """
        Post-checking
        :type root: Node
        """
        if not root:
            return DEFAULT

        if lo <= root.lo and hi >= root.hi:
            return root.val

        if lo >= root.hi or  hi <= root.hi:
            return DEFAULT
```

```
        l = self.query(root.left,  lo, hi)
        r = self.query(root.right, lo, hi)
        return f(l, r)

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

        if idx == root.lo and idx == root.hi-1:
            root.val = 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.val)
        if root.right: val = f(val, root.right.val)

        root.val = val
```

The above code abstracts out segment tree function using `lambda`. For a concrete example, see Count Segment Tree 8.5.4.

6.5 Trie

6.5.1 Basic

Trie is aka radix tree, prefix tree.

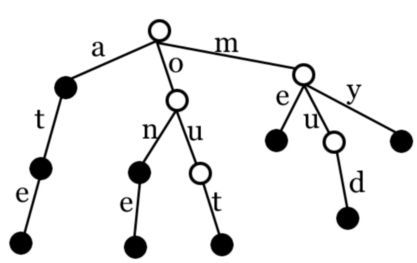


Fig. 6.6: Trie

Notice:

1. Children are stored in HashMap rather than ArrayList.
2. `self.word` to 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):
        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

Storage of words in TrieNode:

1. Implicitly store the current word in the trie with a mark of `is_ended`.
2. Store 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 Simplified Trie

Simplified trie with dict as TrieNode

```
root = {}
ends = []
for word in set(words):
    cur = root
    for c in word:
        nxt = cur.get(c, {})
        cur[c] = nxt
        cur = nxt

    ends.append((cur, len(word)))
```

6.5.4 The Most Simplified Trie

```
# constructor
TrieNode = lambda: defaultdict(TrieNode)

# or
class TrieNode:
    def __init__(self):
        self.children = defaultdict(TrieNode)
        self.attr = None  # some attr with default values
        self.word = None  # a word ends here, value or index
```

6.5.5 Extensions

Search for multiple words Search for combination of words e.g. “unitedstates”. When one word ended, start the search again from the root. One trick is to add threads between tails and the root; thus enable the search for multi-word combinations.

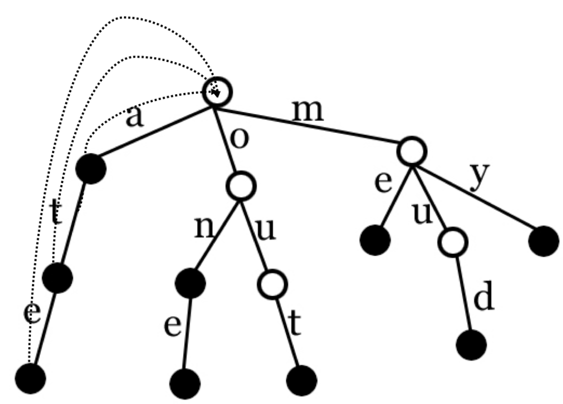


Fig. 6.7: Trie with threads from ending point to root

6.5.6 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.

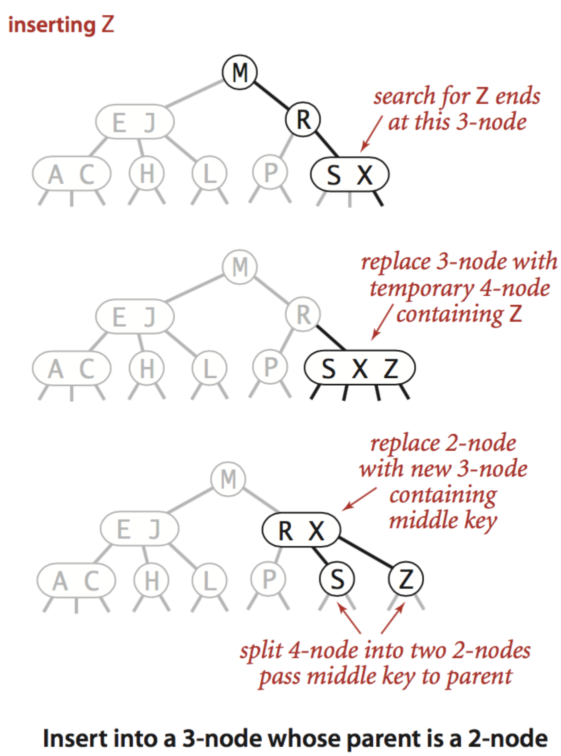


Fig. 7.1: Insertion 1

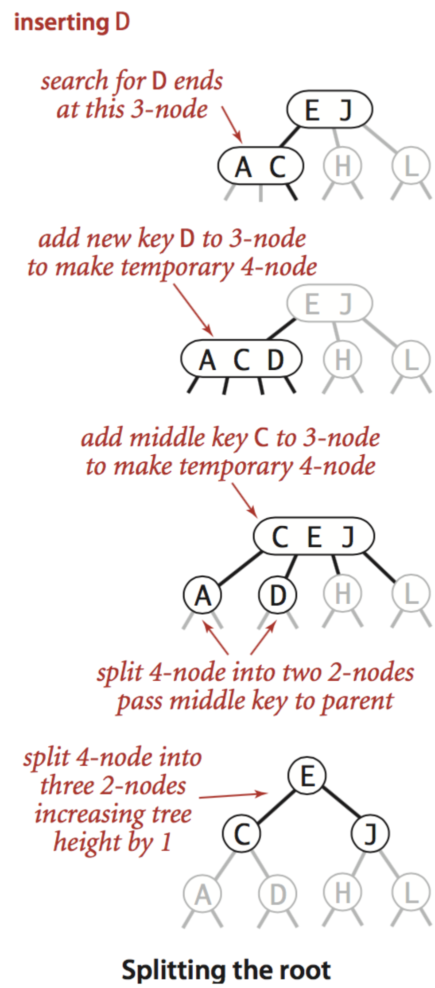


Fig. 7.2: insert 2

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

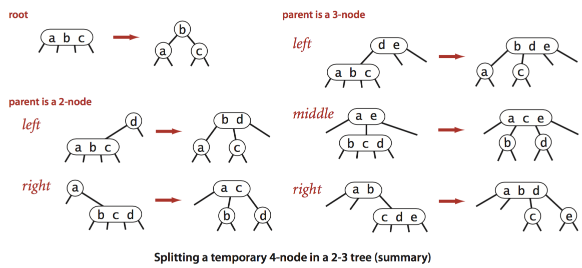


Fig. 7.3: Splitting temporary 4-ndoe summary

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 $2\lg N$ 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.

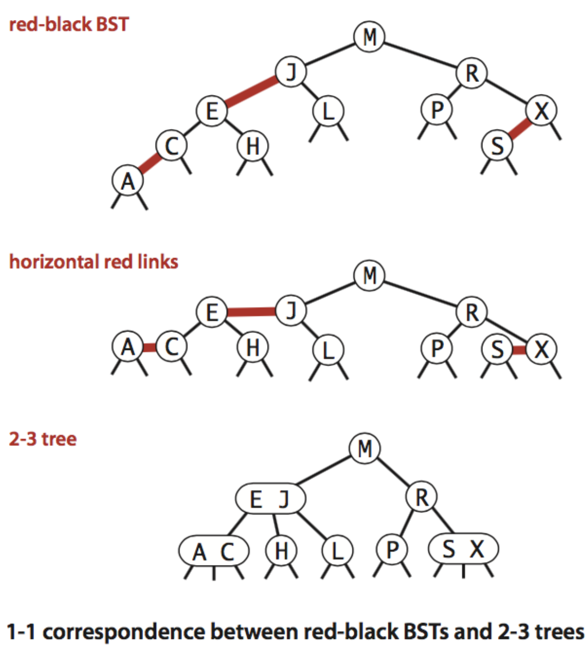


Fig. 7.4: RB-tree and 2-3 tree

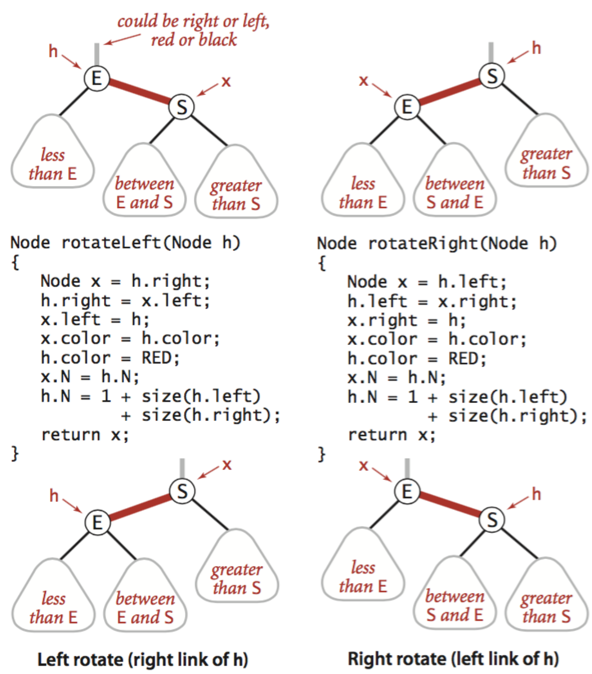


Fig. 7.5: Rotate left/right

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:

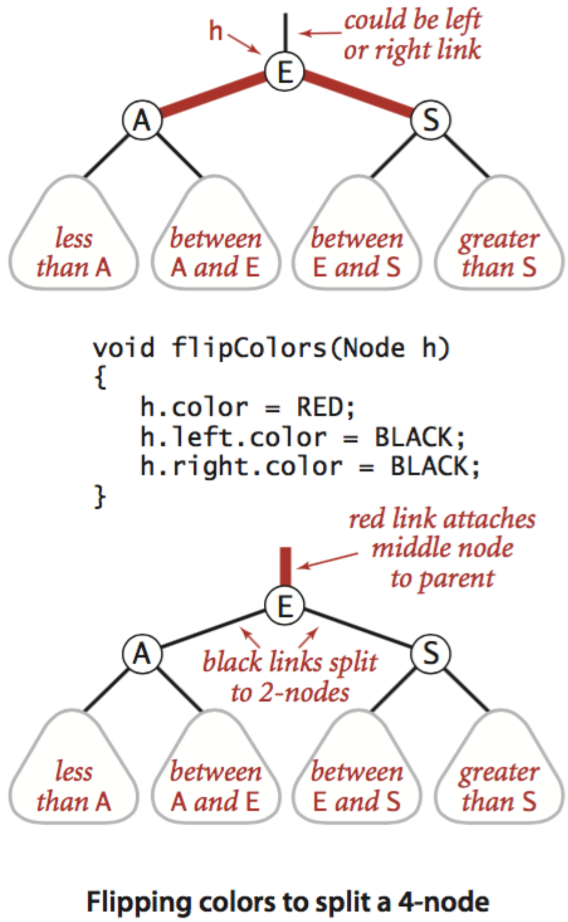


Fig. 7.6: Flip colors

```
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

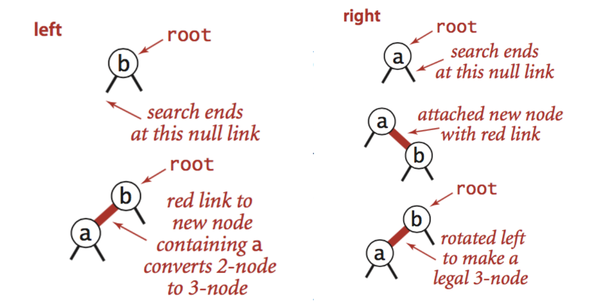


Fig. 7.7: (a) smaller than 2-node (b) larger than 2-nod

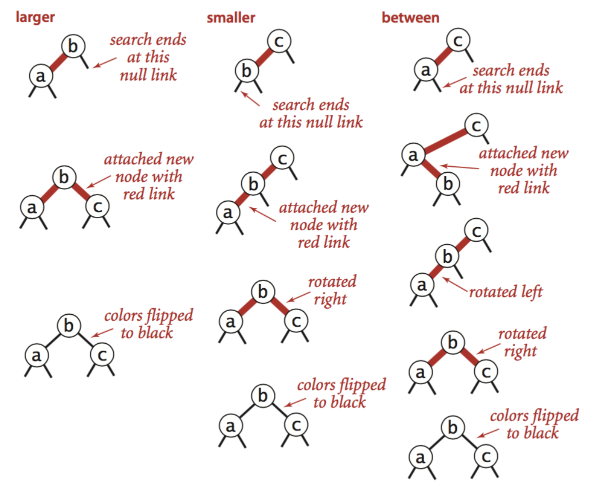


Fig. 7.8: (a) larger than 3-node (b) smaller than 3-node (c) between 3-node.

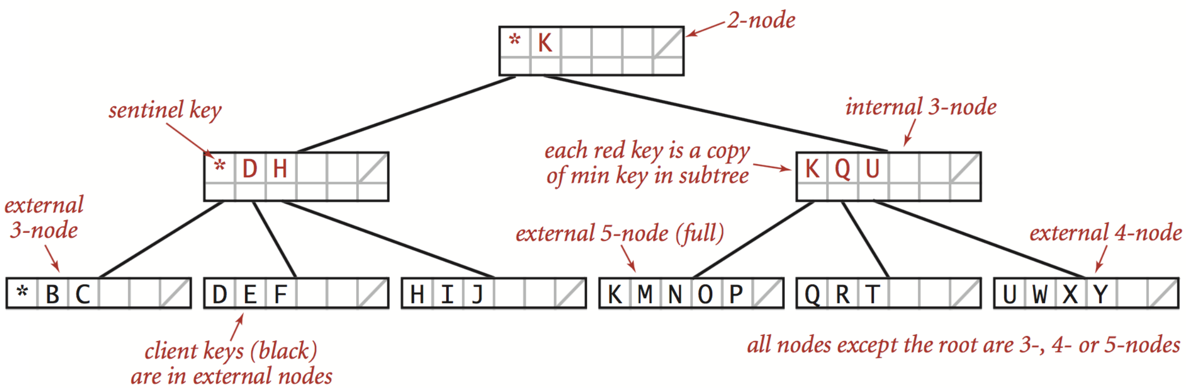
Deletion. Deletion is more complicated.

7.3 B-Tree

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

7.3.1 Basics

Half-full principle:



Anatomy of a B-tree set (M = 6)

Fig. 7.9: B-Tree

Attrs	Non-leaf	Leaf
Ptrs	$\lceil\frac{n+1}{2}\rceil$	$\lfloor\frac{n+1}{2}\rfloor$

Table 7.1: Nodes at least half-full

7.3.2 Operations

7.3.2.1 Insertion

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.

7.3.2.2 Deletion

Core clues

1. **Invariant**: children $\lceil\frac{n+1}{2}\rceil, \lfloor\frac{n+1}{2}\rfloor$
2. **Fuse**: fuse remaining to left sibling, if left not full. *Delete* upper level.
3. **Redistribute**: Extract the last key of left sibling, if left full. *Adjust* upper level.
4. **Non-leaf fuse**: fuse remaining to left sibling, if left not full. *Move down* the upper level.

7.4 AVL Tree

TODO

RB-Tree is preferred since shorter implementation code.

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.

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

Construction algorithm. Similar to all nearest smaller (or larger) values problem - Section 3.3.4.

Core clues:

1. Use stack to maintain a *strictly decreasing* stack, similar to find the all nearest large elements.
2. 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

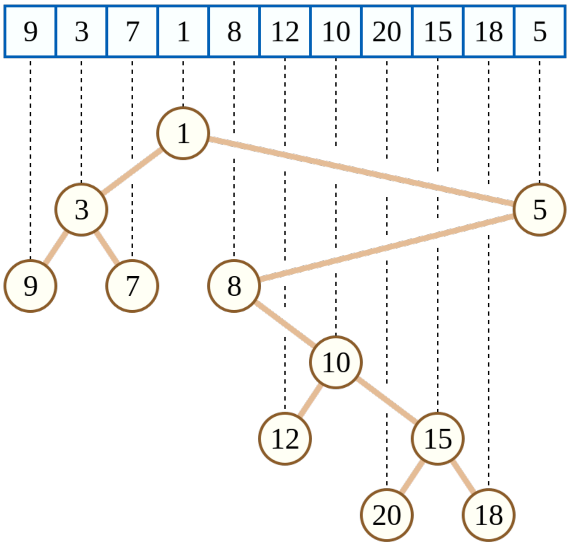


Fig. 7.10: Cartesian Tree

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.

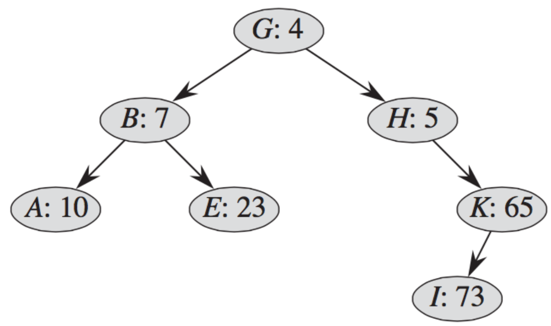


Fig. 7.11: Treap. Each node x is labeled with x.key: x.priority.

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. $|A_p|.. \leq ..|..unseen..|.. \geq ..|$ maintain the 3 subarrays.

5. Heap sort: compared to quick sort it is guaranteed $O(n \lg n)$, compared to merge sort it is $O(1)$ extra space.

8.2 Algorithms

8.2.1 Quick Sort

Concise. Concise but space inefficient:

```
def quicksort(self, A):
    if len(A) <= 1:
        return A
    pivot = A[len(A) // 2]
    left = [e for e in A if e < pivot]
    middle = [e for e in A if e == pivot]
    right = [e for e in A if e > pivot]
    return quicksort(left) + middle + quicksort(right)
```

Pivoting. Pivoting/Partitioning to be space efficient:

```
def quicksort(self, A, lo, hi):
    if lo >= hi:
        return
    p = pivot(A, lo, hi)
    quicksort(A, lo, p-1)
    quicksort(A, p+1, hi)
```

8.2.1.1 Normal pivoting

The key part of quick sort is pivoting. Using the last element as pivot:

```
def pivot(self, A, lo, hi):
    """
    pivoting algorithm:
    | left array | right array | p |
    | left array | p | right array |
    """
    p = hi
    l = lo
    for i in range(lo, hi - 1):
        if A[i] < A[p]:
            A[i], A[l] = A[l], A[i]
            l += 1

    A[l], A[hi] = A[hi], A[l]
    return l
```

Using the first element as pivot:

```
def pivot(self, A, lo, hi):
    """
    pivoting algorithm:
    | p | left array | right array |
    | left array | p | right array |
    """
    p = lo
    l = lo  # left array ending index
    for i in range(lo + 1, hi):
        if A[i] < A[p]:
            l += 1
            A[i], A[l] = A[l], A[i]

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

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

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

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

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 simultaneou
    Avoid the case that the algo goes O(N^2) with duplica
    """
    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

This problem is also known as *Dutch national flag problem*.

3-way pivoting: pivot the array into 3 subarrays:

$$|.. \leq ..|.. = ..|..unseen..|.. \geq ..|$$

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

    pivot = A[lo]
    i = lo  # scanning pointer
    while i < right:  # not n nor hi
        if A[i] < pivot:
            left += 1
            A[left], A[i] = A[i], A[left]
            i += 1
        elif A[i] == pivot:
            i += 1
        else:
            right -= 1
            A[right], A[i] = A[i], A[right]
            # i stays

    return left, right
```

8.2.2 Merge Sort

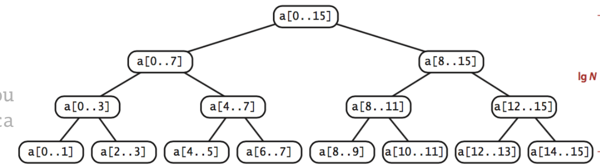


Fig. 8.1: Merge Sort

Normal merge Normal merge sort with extra space

```
def merge_sort(self, A):
  if len(A) <= 1:
    return

  mid = len(A)//2
  L, R = A[:mid], A[mid:]
  self.merge_sort(L)
  self.merge_sort(R)

  l, r, i = 0, 0, 0
  while l < len(L) and r < len(R):
    if L[l] < R[r]:
      A[i] = L[l]
      l += 1
    else:
      A[i] = R[r]
      r += 1
    i += 1

  if l < len(L):
    A[i:] = L[l:]
  if r < len(R):
    A[i:] = R[r:]
```

Merge backward. Merge two arrays in the place of one of the arrays.

```
def merge(self, A, m, B, n):
  """
  Arrays in asc order.
  Assume A has enough space.
  CONSTANT SPACE: starting backward.
  """
  i = m-1
  j = n-1
  closed = m+n

  while i >= 0 and j >= 0:
    closed -= 1
    if A[i] > B[j]:
      A[closed] = A[i]
      i -= 1
    else:
      A[closed] = B[j]
      j -= 1

  # either-or
  # dangling
  if j >= 0: A[:closed] = B[:j+1]
```

```
  # if i >= 0: A[:closed] = A[:i+1]
```

In-place & Iterative merge. In-place merge sort of array without recursive. The basic idea is to avoid the recursive call while using iterative solution.

The algorithm first merge chunk of length of 2, 4, 8 ... until 2^k where 2^k is large than the length of the array.

```
def merge_sort(self, A):
  n = len(A)
  l = 1
  while l <= n:
    for i in range(0, n, l*2):
      lo, hi = i, min(n, i+2*l)
      mid = i + l
      p, q = lo, mid
      while p < mid and q < hi:
        if A[p] < A[q]:
          p += 1
        else:
          tmp = A[q]
          A[p+1:q+1] = A[p:q]
          A[p] = tmp
          p, mid, q = p+1, mid+1, q+1

    l *= 2

  return A
```

The time complexity may be degenerated to $O(n^2)$.

8.2.3 Do something while merging

During the merging, the left half and the right half are both sorted; therefore, we can carry out operations like:

1. inversion count
2. range sum count

Count of Range Sum. Given an integer array nums, return the number of range sums that lie in $[lower, upper]$ inclusively. Make an array A of sums, where `A[i]` is `sum(nums[:i])`, and then feed to merge sort. Since both the left half and the right half are sorted, we can diff A in $O(n)$ time to find range sum.

```
def msort(A, lo, hi):
  if lo + 1 >= hi:
    return 0

  mid = (lo + hi)//2
  cnt = msort(A, lo, mid) + msort(A, mid, hi)

  temp = []
  i = j = r = mid
  for l in range(lo, mid):
    # range count
    while i < hi and A[i] - A[l] <  LOWER: i += 1
    while j < hi and A[j] - A[l] <= UPPER: j += 1
    cnt += j - i

    # normal merge
    while r < hi and A[r] < A[l]:
      temp.append(A[r])
      r += 1

    temp.append(A[l])

  while r < hi:  # dangling right
    temp.append(A[r])
    r += 1

  A[lo:hi] = temp
  return cnt
```

Here, the implementation of merge sort use: 1 for-loop for the left half and 2 while-loop for the right half.

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.

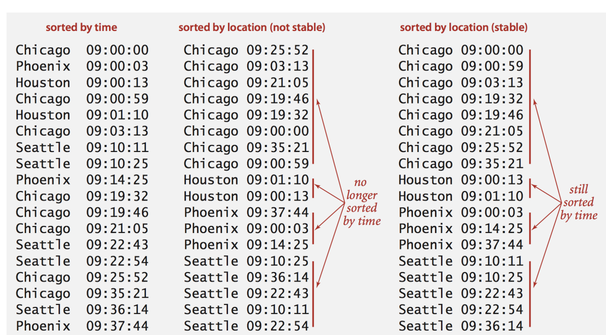


Fig. 8.2: Stale sort vs. unstable sort

8.3.2 Sorting 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 Sorting Summary

See Figure 8.3.

8.4 Partial Quicksort

8.4.1 Find k smallest

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

Version 1, construct heap with n numbers, and take k: $O(n + k \log n)$, where $O(n)$ is for constructing heap.

Version 2. construct heap with k numbers, and iterate n: $O(k + n \log k)$.

The 2nd version is much master than 1st based on emperical analysis; additionally, it has smaller memory impact.

In python there are:

```
heapq.nlargest(n, iterable[, key])
heapq.nsmallest(n, iterable[, key])
```

Partial Quicksort Then the $A[:k]$ is sorted k smallest. The algorithm recursively sort the $A[lo:hi]$

The average time complexity is

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

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

```
def partial_qsort(self, A, lo, hi, k):
    if lo >= hi:
      return

    p = self.pivot(A, lo, hi)
    self.partial_qsort(A, lo, p, k)
    if k <= p+1:
      return
    self.partial_qsort(A, p+1, hi, k)
```

The partial quick sort will find the k smallest number in sorted order. If the top k elements are not required to be sorted, then use find k-th algorithm

8.4.2 Find k-th: Quick Select

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

The average time complexity is

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

Refresh the `def pivot`:

```
def pivot(self, A, lo, hi):
    pivot = lo
    left = pivot  # left == pivot, means no left array
    for i in range(lo + 1, hi):
        if A[i] < A[pivot]:
            left += 1
            A[left], A[i] = A[i], A[left]

    A[left], A[pivot] = A[pivot], A[left]
    return left  # the pivot index
```

Find k-th with 2-way partitioning. Notice that only index changing while k is the same.

```
def find_kth(self, A, lo, hi, k):
    if lo >= hi:
      return

    p = self.pivot(A, lo, hi)
```

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

Fig. 8.3: Sort summary

```
    if k == p:
      return p
    elif k < p:
      return self.find_kth(A, lo, p, k)
    else:
      return self.find_kth(A, p+1, hi, k)
```

Find k-th with 3-way partitioning. Pay attention to the indexing. lt, gt means the last index of less-than portion and larger-than partion.

```
def find_kth(self, A, lo, hi, k):
    if lo >= hi:
      return

    lt, gt = self.pivot(A, lo, hi)
    if lt < k < gt:
      return k
    elif k <= lt:
      return self.find_kth(A, lo, lt+1, k)
    else:
      return self.find_kth(A, gt, hi, k)
```

Pivoting see section - 8.2.1.1.

Find k-th in union of two sorted array. Given sorted two arrays A, B, find the k-th element (0 based index).

Core clues:

1. To reduce the complexity of $O(\log(m+n))$, need to half the arrays.
2. Decide which half of the array to disregard.
3. Decide whether to disregard the median (i.e. boundary point).

```
def find_kth(self, A, B, k):
  if not A:  return B[k]
  if not B:  return A[k]
  if k == 0: return min(A[0], B[0])

  m, n = len(A), len(B)
  if A[m/2] >= B[n/2]:
    if k > m/2 + n/2:
      return self.find_kth(A, B[n/2+1:], k-n/2-1)  # exclude median
    else:
      return self.find_kth(A[:m/2], B, k)  # exclude median
  else:
    return self.find_kth(B, A, k)  # swap
```

8.4.3 Applications

Wiggle Sort. Given an unsorted array A, reorder it such that $A_0 < A_1 > A_2 < A_3$. Do it in $O(n)$ time and $O(1)$ space.

Core clues:

1. Quick selection for finding median (Average $O(n)$)
2. Three-way partitioning to split the data
3. Re-mapping the index to do in-place partitioning

Pre-processing Sorting can be an important pre-processing step as to:

1. Satisfying the output order (e.g. if multiple results are possible, output the one that's smallest in terms of the natural order).

```
class Solution(object):
  def wiggleSort(self, A):
    n = len(A)
    median_idx = self.find_kth(A, 0, n, n/2)
    v = A[median_idx]

    idx = lambda i: (2*i+1)%(n|1)
    lt = -1
    hi = n
    i = 0
    while i < hi:
      if A[idx(i)] > v:
        lt += 1
```

```
        A[idx(lt)], A[idx(i)] = A[idx(i)], A[idx(lt)]
        i += 1
      elif A[idx(i)] == v:
        i += 1
      else:
        hi -= 1
        A[idx(hi)], A[idx(i)] = A[idx(i)], A[idx(hi)]

  def pivot(self, A, lo, hi, pidx=None):
    lt = lo-1
    gt = hi
    if not pidx: pidx = lo

    v = A[pidx]
    i = lo
    while i < gt:
      if A[i] < v:
        lt += 1
        A[lt], A[i] = A[i], A[lt]
        i += 1
      elif A[i] == v:
        i += 1
      else:
        gt -= 1
        A[gt], A[i] = A[i], A[gt]

    return lt, gt

  def find_kth(self, A, lo, hi, k):
    if lo >= hi: return

    lt, gt = self.pivot(A, lo, hi)

    if lt < k < gt:
      return k
    if k <= lt:
      return self.find_kth(A, lo, lt+1, k)
    else:
      return self.find_kth(A, gt, hi, k)
```

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 *samller* than the element but appear *after* it.

8.5.1 MergeSort & Inversion Pair

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

Therefore the Merge-and-count key is `ret += len(A1) - i1`

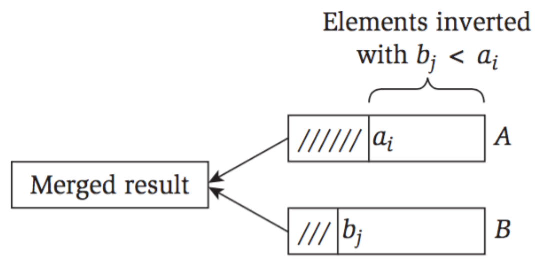


Fig. 8.4: Merge and Count

```
def merge(A1, A2, A):
  i1 = i2 =0
  ret = 0
  for i in range(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 $idx \to sum$ and $sum \to idx$; while BIT can only process $idx \to sum$.

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 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.

Caveat: need to sort the array in the preprocessing step.

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

```
class Node(object):
  def __init__(self, lo, hi, cnt):
    self.lo = lo
    self.hi = hi
    self.cnt = cnt  # size of empty slots
```

```
    self.left = None
    self.right = None

  def __repr__(self):
    return repr("[%d,%d)" % (self.lo, self.hi))


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

  def build(self, lo, hi):
    """a node can have right ONLY IF has left"""
    if lo >= hi: return
    if lo == hi-1: return Node(lo, hi, 1)

    root = Node(lo, hi, hi-lo)
    root.left = self.build(lo, (hi+lo)/2)
    root.right = self.build((lo+hi)/2, hi)
    return root

  def find_delete(self, root, sz):
    """
    :return: index
    """
    root.cnt -= 1
    if not root.left:
      return root.lo
    elif root.left.cnt >= sz:
      return self.find_delete(root.left, sz)
    else:
      return self.find_delete(root.right,
                  sz - 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__":
  # (val, inv)
  A = [(5, 0), (2, 1), (3, 1), (4, 1,), (1, 4)]
  assert Solution().reconstruct(A) == [5, 2, 3, 4, 1]
```

Duplicate. What if the array contains duplicate elements? Use a `Counter` to count the duplicate items already in the result.

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` $\equiv$ `bisect`

Note the subtle differences.

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_lect`.

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
```

Using `bisect_left` with multiple pre-checks to simply the find process.

```
def find(A, v):
  # A is sorted
  if not A:
    return None
  if v >= A[-1]:
    return A[-1]
  if v < A[0]:
    return None

  idx = bisect_left(A, v)
  if A[idx] == v:
    return v
  idx -= 1  # already checked before
  return A[idx]
```

9.1.2 idx equal or just higher

$A_{idx} \geq target$.

```
def bi_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.

By insertion point `i`, it means `all(val <= x for val in A[lo:i])` for the left side and `all(val > x for val in A[i:hi])` for the right side. `A[i]` is the first element larger than x.

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. Case by case analysis. Three cases to consider:

1. Monotonous
2. Trough
3. Peak

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

```
def find_min(self, A):
    lo = 0
    hi = len(A)
    mini = sys.maxsize
    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. `PI`: 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 range(n+1)]
    k = 1
    MIN[k] = A[0]  # store value
    for v in A[1:]:
        idx = bisect.bisect_left(MIN, v, 1, k+1)
        MIN[idx] = v
        k += 1 if idx == k+1 else 0

    return k
```

If need to return the LIS itself.

```
    n = len(A)
    MIN = [-1 for _ in range(n+1)]
    RET = [-1 for _ in range(n)]
    l = 1
    MIN[l] = 0  # store index
    for i in range(1, n):
        if A[i] > A[MIN[l]]:
            l += 1
            MIN[l] = i

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

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

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

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

Chapter 10
Array

10.1 Two-pointer Algorithm

Container With Most Water. Given coordinate (i, a_i), find two lines, which together with x-axis forms a container, such that the container contains the most water.

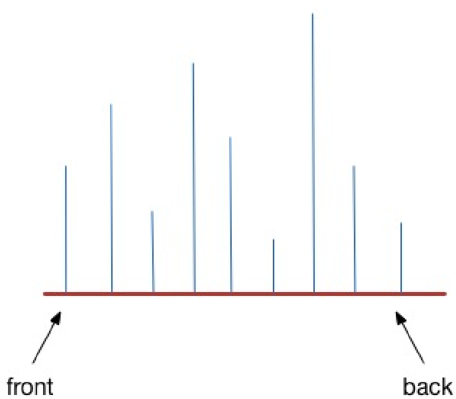


Fig. 10.1: Container with Most Water

When calculate water area, the water height h is constrained by $\min(h_{left}, h_{right})$. Core clues:

1. **Two pointers**: *start*, *back* at two ends. Calculate the current area
2. **Move one**: Move the shorter (lower height) pointer.

Why does this greedy algorithm work? When searching further, we always search the best one. When searching the most-water container with length l, we always reach the maximum area. When shrinking the water length by 1, we always reach the higher height.

If we don't drop the lower side, we will always be constrained by the lower side.

10.2 Circular Array

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

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

10.2.1 Circular max sum

Linear (acyclic) problem can be solved linear with dp algorithm for maximum subarray sum - Section 22.2.

The circular sum should use dp.

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

Core clues:

1. **State definitions**:
   Max subarray sum from index 0. Construct left max sum L_i for max sum over the $[0..i]$ with subarray starting at 0 (*forward* starting from the left side).
   Max subarray sum from index -1. 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:**
$$L_i = \max\left(L_{i-1}, sum(A[:i])\right)$$
$$R_i = \max\left(R_{i+1}, sum(A[i:])\right)$$
3. **Global result**:
$$maxa = \max(R_i + L_{i-1}, \forall i)$$

10.2.2 Non-adjacent cell

Maximum sum of non-adjacent cells in an array A. (House robbery problem)

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

1. Consider the A_i
2. Not consider the A_i

and solve them using linear maximum sum of non-adjacent cells separately - Section 22.2.

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

10.2.3 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 got the **sorted half** or the **unsorted half**.

10.3 Voting Algorithm

10.3.1 Majority Number

10.3.1.1 $\frac{1}{2}$ 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: Find majority number x in A. Mathematically, find x in array A with length n s.t. $cnt_x > n - cnt_x$.

Find a **pair** (a_i, a_j) in A, if $a_i \neq a_j$, delete both from A. The counter still holds that: $C_x^{A'} > |A'| - C_x^{A'}$. Proof, since $a_i \neq a_j$, at most 1 of them equals x, then $C_x^{A'}$ decrements at most by 1, $|A'|$ decrements by 2.

To find such pair $(a_i, a_j), a_i \neq a_j$, linear time one-pass algorithm. That's why *Moore's voting algorithm* is correct.

At any time in the execution, let A' be the prefix of A that has been processed, if $counter > 0$, then keep track the candidate x's counter, the x is the majority number of A'. If $counter = 0$, then for A' we can pair the elements s.t. are all pairs has distinct element. Thus, it does not hold that $cnt_x > n - cnt_x$; thus $x \in A'$. ■

Re-check: 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
```

10.3.1.2 $\frac{1}{3}$ 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.

10.3.1.3 $\frac{1}{k}$ 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, A, k):
    """
    Since majority elements appears more
    than ceil(n/k) times, there are at
    most k-1 majority number
    """
    cnt = defaultdict(int)
    for e in A:
        if e in cnt:
            cnt[e] += 1
        else:
            if len(cnt) < k-1:
                cnt[e] += 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, A))
            > len(A)/k:
```

```
        return key

    raise Exception
```

10.4 Two Pointers

10.4.1 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. How to do it in O(N).
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 range(n):
        # search for the next
        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
```

10.5 Index Remapping

10.5.1 Introduction

Virtual Index. Similar to physical and virtual machine, the underlying indexing i for an array A is the physical index, while we can create virtual indexing i' for the same array A to map $A_{i'}$ to a physical entry A_i.

10.5.2 Example

Interleaving indexes Given an array A of length n, we want to mapping the virtual indexes to physical indexes such that A_0 maps to A_1, A_1 maps to A_3,..., $A_{\lfloor n/2 \rfloor}$ maps to A_0, as followed:

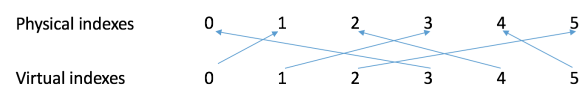


Fig. 10.2: Virtual Indices Remapping

```
0 -> 1
1 -> 3
2 -> 5
...
n/2-1 -> n-1 or n-2

n/2 -> 0
n/2+1 -> 2
...
n -> n-2 or n-1
```

If n is even,

$$(2*i+1)\%(n+1)$$

If n is odd,

$$(2*i+1)\%(n)$$

Thus, by combining two cases, we create the mapping relationship:

```
def idx(i):
    return (2*i+1) % (n|1)
```

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 char counters of `s`.
2. jump parent char

Code:

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

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

Jump within the looping to avoid repetition.

11.2 KMP

Find string W in string S within complexity of $O(|W|+|S|)$. KMP - reflect upon yourself before judging others.

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 range(len(needle)+1)]
T[0] = -1
T[1] = 0

cnd = 0  # candidate
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
```

`T[cnd]` is the length, thus just the next index to be processed in the next loop.

11.2.2 Searching algorithm

```
             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

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 `len(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
```

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. $\rightarrow$ double-ended queue.

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

Sliding Window Median. Find the list of median in the sliding window. $\rightarrow$ Dual heap with lazy deletion - section 13.4.2.

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 Divisor

gcd. Greatest common divisor.

```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b

    return a
```

Proof. Euclidean Algorithm. The Euclidean algorithm is based on the principle that the GCD of two numbers a and b is the same as the GCD of b and $a\%b$ until b becomes zero. Prove the following recursive form:

$$gcd(a,b) = gcd(b,r)$$

1. **Divisibility**: Let g be the GCD of a and b. By definition, g divides both a and b. From the equation $a = b \cdot q + r$, it follows that g must also divide the remainder r, because any divisor of both a and b divides linear combination of them: $r = a - b \cdot q$.
2. **Reduction**: If we replace a with b and b with r, the GCD remains unchanged, and the problem size gets smaller (since $r < b$).
3. **Termination**: The algorithm repeatedly reduces the size of the numbers by replacing the larger number with the remainder. Eventually, one of the numbers will become zero, and the GCD is the other number, since $gcd(a,0) = a$.

13.3 Prime Numbers

13.3.1 Sieve of Eratosthenes

13.3.1.1 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.

13.3.1.2 Refinements

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

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

For example, consider $p = 5$. The first multiple of 5 that we need to mark is $5^2 = 25$, because:

1. $5 \times 2 = 10$ has already been marked when processing $p = 2$.
2. $5 \times 3 = 15$ has already been marked when processing $p = 3$

3. $5 \times 4 = 20$ has already been marked when processing $p = 2$.

Therefore, we only need to start marking multiples from p^2, since all smaller multiples of p have already been handled.

```
def count_primes_sieve(N):
    if N < 2:
        return 0
    # intialize all numbers prime candidates
    primes = [True for _ in range(N+1)]
    # 0 and 1 are not prime numbers
    primes[0] = primes[1] = False

    p = 2
    while (p * p <= N):
        if primes[p]:
            for i in range(p * p, N+1, p):
                primes[i] = False
        p += 1

    return sum(prime)
```

Time complexity. Iterating `p` is $O(N)$ and for all the prime number, each inner loop take $\frac{N}{p}$. Then it becomes

$$\sum_{p \le N} \frac{N}{p} = N \sum_{p \le N} \frac{1}{p}$$

It looks like $O(N \log N)$ but p are prime numbers, it becomes $O(N \log \log N)$. The **Prime Number Theorem** tells us that the number of primes less than or equal to N is approximate

$$\frac{N}{\log N}$$

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

To summarized, the refinements include:

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

```
def count_primes(N):
    if N < 3:
        return 0
    primes = [
        False if i%2 == 0 else True
        for i in range(n)
    ]
    primes[0], primes[1] = False, False
    for i in range(3, int(math.sqrt(N))+1, 2):
        if primes[i]:
            for j in range(i*i, n, 2*i):
                primes[j] = False

    return prime.count(True)
```

13.3.2 Factorization

Backtracking: Section-19.6.1.1.

13.4 Median

13.4.1 Basic DualHeap

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

Here we use the negation of the value as a trick to convert min-heap to max-heap.

```
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"""
```

13.4.2 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
```

13.5 Modular

13.5.1 Power of 4

To check whether a number of the power of 4, we can check whether it mod 3 equals 1.

$$\begin{aligned} 4^a &\equiv 1^a \mod 3 \\ &\equiv 1 \mod 3 \end{aligned}$$

Alternatively, we can use bit manipulation based on the power of 4 in the binary form of `repeat n 1 << 2`, and checks whether there is even number of 0's in binary form.

13.6 Ord

Number in lexical order. Given an integer n, return 1 - n in lexicographical order. For example, given 13, return: [1,10,11,12,13,2,3,4,5,6,7,8,9].

Using DFS.

```
def dfs(N, cur, ret):
    ret.append(cur)
    for d in range(10):
        nxt = cur * 10 + d
        if nxt <= N:
            dfs(N, nxt, ret)
        else:
            break

N = 105
ret = []
for i in range(1, 10):
    if i <= N:
        dfs(N, i, ret)
    else:
        break
```

Using iterative appraoch.

```
def gen():
    i = 1
    for _ in range(n):
        yield i
        if i * 10 <= n:
            i *= 10  # * 10
        elif i % 10 != 9 and i + 1 <= n:
            i += 1  # for current digit
        else:
            while i % 10 == 9 or i + 1 > n:
                i /= 10
            i += 1
```

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 range(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
```

Multiplication. The key to big number multiplication is to break down the problem:

1. Multiply one digit by one.
2. Add one big number by one.
3. Add a list of big number with increasing significance

Details see code.

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*, except for brackets, which are put onto stack directly.
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:
      # maintain invariant
      while stk and not precdn(stk[-1]) < precdn(elt):
        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, except for brackets.

In-fix

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

can be written as the intermediate representation (IR)

```
3 4 2 1 + * 5 + -
```

reverse as the pre-fix

```
- + 5 * + 1 2 4 3
```

```python
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:
      # maintain invariant
      while stk and not precdn(stk[-1]) <= precdn(elt):
        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 the intermediate representation (IR)

```
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? Does the **timing** of choice matter?
2. Are the object draws **repeatable**?
3. Are the objects partially **duplicated**?

If order does not matter or repeated objects, you can preset the order.

15.1.2 Basic formula

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

The 1st equation is complimentary. Considering a binary choice $x_i \in 0,1$ indicating whether to choose an item `e`. Choosing k objects (assigning 1) directly means not choosing $n-k$ objects (assigning 0).

$$
\begin{aligned}
X &= \{x_0, x_1, \ldots x_{n-1}\} \\
x_i &\in 0,1 \\
\sum_i x_i &= k \\
\sum_i 1 - x_i &= n-k \\
|X| &= n
\end{aligned}
$$

The 2nd equation is deduping. Considering a set of objects $A = \{a,b,c,d,e\}$. $n = 5$. The number of total permutations is 5!. If we want to pick $k = 3$ objects: $list = [\alpha_0, \alpha_1, \alpha_2]$, but the order does not matter or element are repeatble, it becomes $set = \{\alpha_0, \alpha_1, \alpha_2\}$. It means 3! permutation *list* is degraded to a single *set*. At the same time, the complimentary $list_{comp} = [\alpha_3, \alpha_4]$ is degraded to a single set_{comp}. For example, we are choosing $\{a,b,c\}$ and when the program is processing the permutation *abcde* or *adbec*, we need to count duplicates for deduping.

$$
\begin{aligned}
\mathbf{abc}de &= \mathbf{bac}de \\
\mathbf{abc}de &= \mathbf{abc}ed \\
\mathbf{a}d\mathbf{b}e\mathbf{c} &= \mathbf{b}d\mathbf{a}e\mathbf{c} \\
\mathbf{a}d\mathbf{b}e\mathbf{c} &= \mathbf{a}e\mathbf{b}d\mathbf{c} \\
ret &= \frac{5!}{3! \cdot 2!}
\end{aligned}
$$

The 3rd equation is DP. Let $F_{n,k}$ be the number of combinations of choosing k elements from n elements. We can either pick the n-th item or not.

$$F_{n,k} = F_{n-1,k} + F_{n-1,k-1}$$

15.1.3 N objects, K ceils. Stars & Bars

When $N = 10, K = 3$:

$$x_1 + x_2 + x_3 = 10$$

is equivalent to

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

, notice that $*$ are non-order, and it is possible to have

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

then the formula is:

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

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

Unique paths. Given a $m \times n$ matrix, starting from $(0,0)$, ending at $(m-1, n-1)$, can only goes down or right. What is the number of unique paths?

Let $F_{i,j}$ be the number of unique paths at `[i][j]`.

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

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

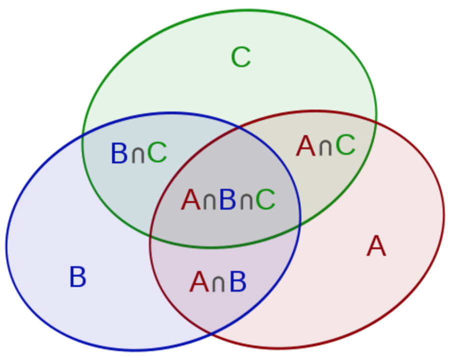


Fig. 15.1: Inclusion–exclusion principle

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

Generally,

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

15.2 Combinations with Limited Repetitions

Determine the number of combinations of 10 letters (order does not matter) that can be formed from 12 letters of 3A, 4B, 5C, i.e. multisets $S = \{3.A, 4.B, 5.C\}$.

15.2.1 Basic Solution

$|S| = 12$. If there are unlimited the number of any of the letter, it is $\binom{10+2}{10}$ by stars and bars of $x1 + x2 + x3 = 10$; then we get the universal set,

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

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

The result is:

$$
\begin{aligned}
|3A \cap 4B \cap 5C| =& |U| \\
&- \sum (|P_i| \cdot \forall i) \\
&+ \sum (|P_i \cap P_j| \cdot \forall i, j) \\
&- \sum (|P_i \cap P_j \cap P_k| \cdot \forall i, j, k)
\end{aligned}
$$

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

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

Similarly, we can get P_B, P_C.

To calculate $|P_i \cap P_j|$, take $|P_A \cap P_B|$ as an example for 4A and 5B; thus,

$$
|P_A \cap P_B| = \binom{1+2}{2}
$$

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 Interpretation

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. $k \in [1, n!]$. $O(n!)$ in time complexity is easy.

```
def getPermutation(n, k):
    A = [i + 1 for i in range(n)]
    ret = []
    genPermutation(A, 0, [], ret)
    ret.sort()  # requried
    return ret[k - 1]

def genPermutation(A, idx, cur, ret):
    if idx == len(A):
        ret.append("".join(map(str, cur)))

    for j in range(idx, len(A)):
        A[idx], A[j] = A[j], A[idx]
        cur.append(A[idx])
        genPermutation(A, idx + 1, cur, ret)
        A[j], A[idx] = A[idx], A[j]
        cur.pop()
```

Can we do it in $O(nk)$ or less?
Reversed Cantor Expansion
Core clues:

1. `A = [1, 2, ..., n]`
   Suppose for n element, the k-th permutation is:
   `ret = [a0, a1, a2, ..., an-1]`. A_i is different from a_i.
2. **Basic case.** Since `[a1, a3, ..., an-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  # since k was 1-indexed

    A = [i + 1 for i in range(n)]
    # k %= math.factorial(n)  # if k > n!
    ret = []
    for i in range(n-1, -1, -1):
        idx = k // math.factorial(i)
        cur = A.pop(idx)
        ret.append(cur)
        k = k % math.factorial(i)

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

15.3.2 Numbers counting

Count numbers with unique digit. Given a non-negative integer n, count all numbers with unique digits, x, where $0 \le x < 10^n$.

Digit by digit:

1. The 1st digit has 10 possibilities. The 2nd digit has 9 possibilities. Therefore it seems to be A_{10}^n.
2. Exception: The first digit cannot be 0. Therefore it is $9 \times 9 \times 8 \times ... \times (10-i)$

15.4 CATALAN NUMBER

15.4.1 Math

Definition.

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

Proof. Proof of Calatan 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.

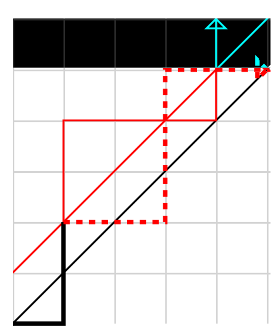


Fig. 15.2: Monotonic Paths

- Total monotonic paths including both exceeding and not exceeding - from $2n$ choose n right and n up:

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

- Find out how many combinations that exceeds the main diagonal. Flip all the exceeding lines at the line just above the diagonal line, we will see a rectangle of size $(n-1)*(n+1)$ has formed. Thus the total number of monotoic paths in the new rectange is - from $2n$ choose $n-1$ right and $n+1$ up:

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

- Thus, the number of path without *exceeding* (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, #($\geq$ #).

#Full binary trees. Number of different full bindary tree. A full binary tree is a tree in which every node has either 0 or 2 children. Consider it as a set of same binary operators with their operands. Reduce this problem to #Parentheses.

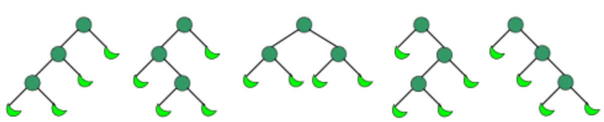


Fig. 15.3: #Full bindary trees. Circles are operators; crescents are operands.

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 $\left\{ {n \atop k} \right\}$.

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

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 entire deck A.

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

```
def shuffle(A):
  n = len(A)
  for i in range(n):
    j = random.randrange(n)
    A[i], A[j] = A[j], A[i]
```

Consider 3 cards, the easiest proof that this algorithm does not produce a uniformly random permutation is that it generates $3^3 = 27$ possible plans (consider steps in plans, duplicated result included), but there are only 3! = 6 permutations. Since $27\%6 \neq 0$, there must be some permutation is that is picked too much, and some that is picked too little.

In general, it generates n^n possible plans, and

$$n^n \% n! \neq 0$$

16.1.2 Knuth Shuffle

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

Intuition. Similar to drawing lots, the order of drawing lots does not affect the probability of a given outcome.

Core clues:

1. choose index uniformly $\in [i, N)$.
2. just like shuffling the poker card.

```
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]);
    }
}
```

```
def shuffle(A):
  n = len(A)
  for i in range(n):
    j = random.randrange(i, n)
    A[i], A[j] = A[j], A[i]
```

Proof: $n!$ permutations.

16.1.3 Random Maximum

Find the index of maximum number in an array with a probability of $\frac{1}{\#maxima}$, instead of the first seen index.

```
def random_max(A):
    # k is the counter
    maxa, maxa_idx, k = A[0], 0, 1
    for i in range(1, len(A)):
        if maxa < A[i]:
            maxa, maxa_idx, k = A[i], i, 1
        elif maxa == A[i]:
            k += 1
            if random.random() < float(1)/k:
                maxa_idx = i

    return maxa_idx
```

Intuition. Consider the maximums array of length $L = k$, the `current` *max* has the probability $\frac{1}{k}$ of being selected. When moving to $L = k+1$, the `current` *max* has the probability $\frac{1}{k+1}$ of being selected, the `previous` *max* has the probability of $\frac{k}{k+1}$ to stay. Then the `previous` *max* has the new updated probability of being selected as:

$$P(max_k) = \frac{1}{k} \cdot \frac{k}{k+1} = \frac{1}{k+1}$$

Proof. Prove via mathematical induction.

That is, assuming it works for any array of size N, prove it works for any array of size $N+1$.

So, given an array of size $N+1$, think of it as a sub-array of size N followed a new element at the end. By assumption, your algorithm uniformly selects one of the max elements of the sub-array... And then it behaves as follows:

If the new element is larger than the max of the sub-array, return that element. This is obviously correct.

If the new element is less than the max of the sub-array, return the result of the algorithm on the sub-array. Also obviously correct.

The only slightly tricky part is when the new element equals the max element of the sub-array. In this case, let the number of max elements in the sub-array be k. Then, by hypothesis, your algorithm selected one of them with probability $\frac{1}{k}$. By keeping that same element with probability $\frac{k}{k+1}$, you make the overall probability of selecting that same element equal

$$\frac{1}{k} \cdot \frac{k}{k+1} = \frac{1}{k+1}$$

, as desired. You also select the last element with the same probability.

To complete the inductive proof, just verify the algorithm works on an array of size 1.

Random pick index. Similar question - given an array of integers with possible duplicates, randomly output the index of a given target number.

```
def pick(self, target):
    sz = 0
    ret = None
    for idx, val in enumerate(A):
        if val == target:
            sz += 1
            rv = random.randrange(0, sz)
            if rv == 0:  # or any number < sz
                ret = idx

    return ret
```

16.2 Sampling

16.2.1 Reservoir Sampling

Sample k from A, where the length of A is either very large or unknown or dynamic.

```
def reservoir_sample(A, k):
  R = A[:k]  # k for size

  for i in range(k, len(A)):
    # rv for random variable
    rv = random.randrange(0, i+1)
    if rv < k:
      R[rv] = A[i]
```

Intuition. When processing index i and A_i, we have the elements of length i already been scanned. Every element $\in [A_0, A_1, ..., A_i]$ has a probability in the reservoir of:

$$P_{select} = \frac{k}{i+1}$$

16.3 Distribution

16.3.1 Geometric Distr

$$P(X = k) = (1-p)^{k-1} p$$

Expected number of trials of get a specific outcome:

$$E[T] = \frac{1}{p}$$

, which is the mean of the Geometric Distr.

16.3.2 Binomial Distr

Notations:

$$B(n, p)$$

pmf:

$$\binom{n}{k} p^k (1-p)^{n-k}$$

16.4 Expected Value

16.4.1 Dice value

Expected value of rolling dice until getting a 3

16.4.2 Coupon collector's problem

Given n coupons, how many coupons do you expect to draw with replacement before having drawn each coupon at least once?

$$E[T] = \Theta(n \lg n)$$

, where T is number of trial (i.e. time).

Let T be the time to collect all. t_i be the time to collect the i-th new different coupon. p_i be the probability of collecting the i-th coupon after $i-1$ coupons have been collected. Observe that:

$$
\begin{aligned}
p_1 &= \frac{n}{n} \\
p_2 &= \frac{n-1}{n} \\
p_i &= \frac{n-i+1}{n}
\end{aligned}
$$

Thus,

$$
\begin{aligned}
E[T] &= \sum_{i=1}^{n} E[t_i] \\
&= \sum \frac{1}{p_i} \\
&= n(\frac{1}{1} + \frac{1}{2} + ... + \frac{1}{n})
\end{aligned}
$$

Dice. How many times must you roll a die until each side has appeared?

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.

```
MAX = 0x7FFFFFFF
MIN = 0x80000000
MSK = 0xFFFFFFFF
```

17.1.2 Operations

Basics

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.

2's complement

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

Negative numbers: `0b10000000` to `0b11111111` (-128 to -1 in decimal). This process essentially "wraps around" the positive binary sequence to start from the most negative number.

```
{
  # 4 overflow
  3: 0b011,
  2: 0b010,
  1: 0b001,
  0: 0b000,
  #########
 -1: 0b111,
 -2: 0b110,
 -3: 0b101,
 -4: 0b100,
}
```

Since 0 takes a position in the binary representation:

- ~ 0 is -1
- ~ 1 is -2
- ~ 3 is -4.

Check 2's power 2's power is in the form of `0b01000`.

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

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

$$x = 0bijk1000$$
$$\sim x = 0bIJK0111$$
$$-x = 0bIJK1000$$

where IJK represent bits that are the negation of ijk.

$$LSB = x \ \& \ -x$$

This above is left extended with 0's. To left extended with 1's.

$$LSB_{1extended} = \sim(lsb-1)$$

Rightmost bit set. MSB. Find the most significant bit.

```
msb = 0
while num >> msb:
  msb += 1
return msb  # index
```

If zero-indexed, the MSB is `msb - 1`.

Alternatively,

```
msb = 1
while msb <= x:
  msb <<= 1
return msb >> 1
```

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 range(len(s)/2))
```

Divide by 2. To divide a number by 2, it should be `x >> 1` rather than `x >> 2`.

17.1.3 Python

Python int is larger than 32 bit. If 32bit signed int, in python, we may need to mask the int:

1. Mask to 32bit: `x & MSK`,
2. Left extended with 1's: `~(x ^ MSK)`

, where `MSK = 0xFFFFFFFF`, 8 F's for 32 bits.

17.2 Radix

Convert to hexadecimal, but with 2's complement. Easy to convert positive number of hex, but need to pay more attention to negative number when thinking in the decimal representation.

Everything easy to convert the number even under 2's complement if thinking in the binary representation.

Core proccess:

1. current digit we need: `num & 0xF`
2. next significant number: `num >>= 4`

17.3 Circuit

It is under 32-bit assumption, for Python, we need additional masking in the previous section.

17.3.1 Full-adder

Plus. Handle carry: only 1 + 1 needs carry, thus `a & b` determines carry.

```
def plus(a, b):
  carry = (a & b) << 1
  out = a ^ b
  if carry != 0:
    return plus(out, carry)
  else:
    return out
```

Half Adder. One bit `a, b`:

```
def half_add(a, b):
  carry = a & b
  out   = a ^ b
  return out, carry
```

Full Adder. One bit `a, b, cin`. `out = a ^ b ^ cin`. and `cout = a & b | cin & a ^ b`

```
def full_add(a, b, cin):
  out, c1 = half_add(a, b)
  out, c2 = half_add(out, cin)
  cout    = c1 | c2  # ^ possible
  return out, cout
```

17.3.2 Full-substractor

Substract. Handle borrow: only 0 - 1 needs borrow, thus `~a & b` determines borrow.

```
def sub(a, b):
  borrow = (~a & b) << 1
  out = a ^ b
  if borrow != 0:
    return sub(out, borrow)
  else:
    return out
```

Half Substractor. One bit `a, b`:

```
def half_sub(a, b):
  borrow = (~a & b)
  out = a ^ b
  return out, borrow
```

Notice negation can be done in xor.

```
~a == 1 ^ a
```

Full Substractor. One bit `a, b, bin`. `out = a ^ b ^ bin`.

```
def full_sub(a, b, bin):
  out, b1 = half_sub(a, b)
  out, b2 = half_sub(out, bin)
  bout = b1 | b2
  return out, bout
```

17.3.3 Multipler

17.4 Single Number

17.4.1 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:

$$
\begin{array}{l}
0000 \\
0001 \\
0010 \\
... \\
1111
\end{array}
$$

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).

17.4.2 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.
- Then do $a_i \wedge a_j$ in two groups respectively to find out a and b from each group.

17.5 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.

Maximum XOR Maximum XOR of Two Numbers in an Array. To achieve $O(N)$, check bit by bit rather than number by number.

Chapter 18
Greedy

18.1 Introduction

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

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

Greedy algorithm is a degraded DP since the the past substructure is not remembered.

18.1.1 Proof

The proof technique for the correctness of the greedy method.

Proof by contradiction, the solution of greedy algorithm is $\mathcal{G}$ and the optimal solution is $\mathcal{O}$, $\mathcal{O} \neq \mathcal{G}$ (or relaxed to $|\mathcal{O}| \neq |\mathcal{G}|$).

Two general technique it is impossible to have $\mathcal{O} \neq \mathcal{G}$:

1. **Exchange method**: Greedy-Choice Property. Any optimal solution can be transformed into the solution produced by the greedy algorithm without worsening its quality.
2. **Stays-ahead method**: Optimal Substructure. the solution constructed by the greedy algorithm is consistently as good as or better than any other solution at every step or position

18.2 Extreme First

Rearranging String k distance apart. Given a non-empty string s and an integer k, rearrange the string such that the same characters are at least distance k from each other.

Core clues.

1. The char with the most count put to the result first - greedy.
2. Fill every k slots as cycle - greedily fill high-count char as many as possible.

Implementations.

1. Use a heap as a way to get the char of the most count.
2. `while` loop till exhaust the heap

```
def rearrangeString(self, s, k):
  if not s or k == 0: return s

  d = defaultdict(int)
  for c in s:
    d[c] += 1

  h = []
  for char, cnt in d.items():
    heapq.heappush(h, Val(cnt, char))

  ret = []
  while h:
    cur = []
    for _ in range(k):
      if not h:
        return "".join(ret) if len(ret) == len(s) else ""

      e = heapq.heappop(h)
      ret.append(e.val)
      e.cnt -= 1
      if e.cnt > 0:
        cur.append(e)

    for e in cur:
      heapq.heappush(h, e)

  return "".join(ret)
```

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`.

Jump. Jump to skip ones the same as its parent to avoid duplication.

Complexity. $O(b^d)$, where b is the branching factor and d is the depth.

19.2 Memoization

dfs can be memoized.

```
import functools

@functools.lru_cache(maxsize=None)
def dfs(self, *args):
  pass
```

19.3 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))  # clone
        return

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

        # not select
        self.dfs(A, i+1, k, cur, remain, ret)

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

19.4 String

In general,

- Break down the sequence into `left` and `right` two parts.
- Choose `left` as terminal state, while DFS search the `right`.
- Combine `left` and the search results from `right`.
- Sometimes it is easier to search `left` and choose `right` as terminal state.

19.4.1 Palindrome

19.4.1.1 Palindrome partition.

Given `s = "aab"`, return:
`[["aa","b"], ["a","a","b"]]`

Core clues:

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

Search process:

```
input: "aabbc"

"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:

```python
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))  # clone

    for i in range(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]
```

19.4.2 Word Abbreviation

Core clues:

1. Pivot a letter
2. Left side as a number, right side dfs

```python
def dfs(self, word):
    if not word:
        yield ""

    for l in range(len(word)+1):
        left_num = str(l) if l else ""
        for right in self.dfs(word[l+1:]):
            yield left_num + word[l:l+1] + right
            # note word[l:l+1] and right default ''
```

19.4.3 Split Array - Minimize Maximum Subarray Sum

Split the array into m parts and minimize the max subarray sum.

Core clues:

1. Take one subarray from left, and search the right side for the minimum max subarray.
2. To make process in the DFS, always make the left part a subarray, and DFS the right part.
3. Pivot an index to break the array into the left and right parts.

Search right.

```python
def dfs(self, cur, m):
    """
    * p break the nums[cur:] into left and right part
    * sums is the prefix sum (sums[i] == sum(nums[:i]))
    """
    if m == 1:
        return self.sums[len(nums)] - self.sums[cur]

    mini = float("inf")
    for j in range(cur, lens(nums)):
        left = self.sums[j + 1] - self.sums[0]
        right = self.dfs(j + 1, m - 1)
        # minimize the max
        mini = min(mini, max(left, right))

    return mini
```

Alternatively, search left.

```python
def dfs(self, hi, m):
    """
    j break the nums[:hi] into left and right part
    sums is the prefix sum (sums[i] == sum(nums[:i]))
    """
    if m == 1:
        return self.sums[hi] - self.sums[0]

    mini = float("inf")
    for j in range(hi):
        right = self.sums[hi] - self.sums[j]
        left = self.dfs(j, m - 1)
        # minimize the max
        mini = min(mini, max(left, right))

    return mini
```

19.5 Cartesian Product

Each state can generate multiple combinations. Search through all the combinations.

19.5.1 Pyramid Transition Matrix.

```
"""
 (H, I ..)
   / \
(D,E) (F, G)
```

```
 / \ / \
A   B   C
"""
def dfs(
    self,
    T: Dict[Tuple[str, str], Set[str]],
    level: str,
) -> bool:
    """
    T - Transition matrix
    stores all the possible end states from state1
    and state2
    [s1, s2] -> set of end states
    """
    if len(level) == 1:
        return True

    for nxt_level in itertools.product(
        *[T[a, b] for a, b in zip(level, level[1:])]
    ):
        if self.dfs(T, nxt_level):
            return True

    return False
```

19.6 MATH

19.6.1 Decomposition

19.6.1.1 Factorize a number

Core clues:

1. Expand the search tree **horizontally**.
2. Take the last number on the stack, and factorize it recursively.

Search tree:

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

[4, 4]
```

Take the last number from the list and factorize it.

Code:

```
ret = []  # collector

def dfs(cur, remain, ret):
  if remain == 1:
    res.append(list(cur))
    return

  start = 2 if not cur else cur[-1]
  # if start = 2, it generates duplicate combinations
  for factor in range(start, remain + 1):
    if remain % factor == 0:
      dfs(cur + [factor], remain // factor)

dfs([], target, ret)
```

Using a single stack to conserve space, we need to maintain the stack **cur**.

```
self.dfs([16], [])

def dfs(self, cur, ret):
  if len(cur) > 1:
    ret.append(list(cur))  # clone

  n = cur.pop()
  start = cur[-1] if cur else 2
  for i in range(start, int(sqrt(n))+1):
    if self.predicate(n, i):
      cur.append(i)
      cur.append(n // i)
      self.dfs(cur, ret)
      cur.pop()  # pop the i here. pop n // i in dfs

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.

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

19.7 ARITHMETIC EXPRESSION

19.7.1 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:

$$\text{“123”}, 6 \rightarrow [\text{“}1+2+3\text{”}, \text{“}1*2*3\text{”}]$$
$$\text{“232”}, 8 \rightarrow [\text{“}2*3+2\text{”}, \text{“}2+3*2\text{”}]$$

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,  # scanning index
    cur_str,  # The current str builder
    cur_val,  # To reach the target
    mul,  # first operand for multiplication
    ret,
):
  if pos >= len(num):
    if cur_val == target:
      ret.append(cur_str)
  else:
    for i in range(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,
            f"nxt_val", nxt_val,
            nxt_val, ret)
      else:  # +, -, *
        self.dfs(num, target, i+1,
            f"{cur_str}+{nxt_val}", cur_val+nxt_val,
            nxt_val, ret)
        self.dfs(num, target, i+1,
            f"{cur_str}-{nxt_val}", cur_val-nxt_val,
            -nxt_val, ret)
        self.dfs(num, target, i+1,
    f"{cur_str}*{nxt_val}", cur_val-mul+mul*nxt_val,
            mul*nxt_val, ret)
```

19.7.2 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
```

19.8 Parenthesis

Remove Invalid Parentheses. Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Core clues:

1. **Backtracking**: All possible results → backtrack.
2. **Minrm**: Find the minimal number of removal.
3. **Jump**: To avoid duplicate, remove all brackets same as previous one π at once.

To find the minimal number of removal:

```
def minrm(self, s):
    """
    returns minimal number of removals
    """
    rmcnt = 0
    left = 0
    for c in s:
        if c == "(":
            left += 1
        elif c == ")":
            if left > 0:
                left -= 1
            else:
                rmcnt += 1

    rmcnt += left
    return rmcnt
```

To return all possible results, do backtracking:

```
def dfs(self, s, cur, left, pi, i, rmcnt, ret):
  """
  Remove parenthesis
  backtracking, post-check
  :param s: original string
  :param cur: current string builder
  :param left: number of remaining left parentheses in s[0..i]
  :param pi: last removed char
  :param i: current index
  :param rmcnt: number of remaining removals needed
  :param ret: results
  """
  if left < 0 or rmcnt < 0 or i > len(s):
    return
  if i == len(s):
    if rmcnt == 0 and left == 0:
      ret.append(cur)
    return
```

```
  if s[i] not in ("(", ")"):  # skip non-parenthesis
    self.dfs(s, cur+s[i], left, None, i+1, rmcnt, ret)
  else:
    if pi == s[i]:
      while i < len(s) and pi and pi == s[i]:
        i += 1
        rmcnt -= 1
      self.dfs(s, cur, left, pi, i, rmcnt, ret)
    else:
      self.dfs(s, cur, left, s[i], i+1, rmcnt-1, ret)
      L = left+1 if s[i] == "(" else left-1  # consume "("
      self.dfs(s, cur+s[i], L, None, i+1, rmcnt, ret)  # not rm
```

19.9 TREE

19.9.1 BST

19.9.1.1 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) -> List[TreeNode]:
  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)  # new instance
        root.left = left_root
        root.right = right_root

        roots.append(root)

  return roots
```

Chapter 20
Divide & Conquer

20.1 Principles

Divide.

Reduce # sub-problems. After dividing, we have a sub-problems. Now need to identify the redundancy in the a sub-problems. Find the common shared calculations among sub-problems and thus try to reduce a to $a-1$. Identify the **commonality**.

Sub-problem dimension. Reduce the dimensionality of the original problem; thus consider the simpler version of the problem.

Input dimension. Increase the representation dimensionality of the input. For example, in FFT (Fast Fourier Transform) augment the input with complex space.

$$w_{j,k} = e^{j2\pi i/k}$$

Chapter 21
Graph

21.1 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 = defaultdict(list)
```

Complexity. Basic complexities:

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

Table 21.1: Time complexities

Graph & Tree. For a undirected graph to be a tree, it needs to satisfied two conditions:

1. Acyclic
2. All connected

21.2 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
    m = len(grid)
    n = len(grid[0])
    visited = [
      [False for _ in range(n)]
      for _ in range(m)
    ]
    for i in range(m):
      for j in range(n):
        if not visited[i][j] and grid[i][j] == "1":
          self.dfs(grid, i, j, visited)
          cnt += 1

    return cnt

  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 21.8) to reduce each query's complexity from $O(mn)$ to $O(\log mn)$.

Without updating, DFS itself is enough.

21.3 BFS

21.3.1 BFS with Abstract Level

BFS goes through the vertices level by level in a queue.

Distance can be arbitrarily defined. BFS can start with a set of vertices in abstract level of distance, not necessarily neighboring vertices.

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

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

Then it 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, cell in enumerate(row)
    if cell == 0
  ]
  for i, j in q:
    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:  # test distance
        mat[I][J] = mat[i][j]+1
        q.append((I, J))
```

21.4 Detect Acyclic

1. `path` represent the current path, and it is reset/pop 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, and A, B have already been visited $C \to A \to B$.
   b. Excluding predecessor `pi` is wrong considering 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`. But it is okay to double check `visited`.

21.4.1 Directed Graph

Detect cycles (any) in directed graph.

```
def dfs(self, V, v, visited, path) -> bool:
  if v in path:
    return False

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

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

21.4.2 Undirected Graph

Detect cycles (any) in undirected graph.

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

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

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

21.5 Directed Graph

Use `G = defaultdict(dict)` to represent directed graph, so that later on the edge weight can be accessed as `G[s][e]`. The `s` start and `e` end are the vertices, and `G[s][e]` returns edge weight.

DFS. DFS in directed graph:

```
def dfs(self, G, s, e, path):
    if s not in G or e not in G:
        return INVALID
    if e in G[s]:
        return G[s][e]
    for nbr in G[s]:
        if nbr not in path:
            path.add(nbr)
            val = self.dfs(G, nbr, e, path)
            if val != -1.0:
                return val * G[s][nbr]
            path.remove(nbr)

    return INVALID
```

21.6 PATHS

21.6.1 Euler Path - Every Edge Once

An Eulerian path is a path in a graph which visits every edge exactly once ($\forall e \in E$). Vertices can be repeated.

Hierholzer's algorithm to find an Euler path in a graph. The graph must be directed graph.

Core clue.

1. The algorithm exhaustively visit all the edges during the dfs.
2. We must **remove** the edge after visting, otherwise circle.
3. After dfs a vertex v's all neighbors and path, `appendleft` v.

```
def findItinerary(self, tickets):
    G = defaultdict(list)
    for s, e in tickets:
        heapq.heappush(G[s], e)  # heap lexical order

    ret = deque()
    self.dfs(G, 'JFK', ret)
    return list(ret)

def dfs(self, G, cur, ret):
    while G[cur]:
        # need to remove the edge after visting
        nbr = heapq.heappop(G[cur])
        self.dfs(G, nbr, ret)

    ret.appendleft(cur)
```

Alternatively, instead of using `heapq`, using `deque`.

```
def findItinerary(self, tickets):
    G = defaultdict(deque)
    for s, e in tickets:
        G[s].append(e)

    for s, l in G.items():
        G[s] = deque(sorted(l))

    ret = deque()
    self.dfs(G, "JFK", ret)
    return list(ret)

def dfs(self, G, cur, ret):
    while G[cur]:
        # need to remove the edge after visting
        nbr = G[cur].popleft()
        self.dfs(G, nbr, ret)

    ret.appendleft(cur)
```

21.6.2 Hamiltonian Path - Every Vertex Once

A Hamiltonian path is a path in a graph which visits every vertex exactly once ($\forall v \in V$). This problem is proved to be NP.

21.7 TOPOLOGICAL SORTING

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

21.7.1 Algorithm

Core clues:

1. **DFS neighbors first**. For a given vertex v, 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 v and push it to the result queue `appendleft`.

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 *path*.

```
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)

# return whether the current path is acyclic
def dfs_topo(self, V, v, visited, path, ret) -> bool:
  if v in path:
    return False

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

```
  path.remove(v)
  visited.add(v)
  ret.appendleft(v)
  return True
```

Alternatively, we can encode `path` into `visited`.

```
# encode the visited using 0, 1, 2
def dfs_topo(
  self,
  G: Dict[int, List[int]],
  u: int,
  visited: Dict[int, int],
  ret: deque,
):
  """
  Topological sort
  G = defaultdict(list)
  visited = defaultdict(int)
  # 0 not visited, 1 visiting, 2 visited

  pre-condition: u is not visited (0)
  """
  visited[u] = 1
  for nbr in G[u]:
    if visited[nbr] == 1:
      return False
    if visited[nbr] == 0:
      if not self.topo_dfs(G, nbr, visited, ret):
        return False

  visited[u] = 2
  ret.appendleft(u)  # visit larger first
  return True
```

The time complexity of topological sorting is $O(|E|+|V|)$ since it needs to goes to every edges and every vertices.

21.7.2 Applications

1. Course scheduling problem with pre-requisite.

21.8 UNION-FIND

21.8.1 Simplified Union Find

Simplified code with unbalanced size. Union-find and disjoint set are used interchangeably. Union-find emphasizes on algorithm while disjoint set emphasizes on data structure.

Core clues:

1. π **array**:an array to store each item's predecessor pi. The predecessor are lazily updated to its ancestor.
2. When `x == pi[x]`, then `x` is the ancestor (i.e. root).
3. Otherwise, `pi[x] = find(pi[x])`

```
class DisjointSet:
    def __init__(self):
        self.pi = {}

    def union(self, x, y):
        pi_x = self.find(x)
        pi_y = self.find(y)
        self.pi[pi_y] = pi_x

    def find(self, x):
        if x not in self.pi:
          self.pi[x] = x
          return x

        pi = self.pi[x]
        if x != pi:
          self.pi[x] = self.root(pi)
        return self.pi[x]
```

Note, for the final counting of component. Need to do a final find on all nodes to update the ancestor.

```
component_cnt = len(set(
    ds.find(x)
    for x in ds.pi.keys()
))
```

Improvements:

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

21.8.2 Algorithm

Weighted union-find with path compression.
Core clues:

1. π **array**: predecessor pi.
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.

```
class UnionFind(object):  # or DisjointSet
  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 find(self, x):
    """path compression"""
    pi = self.pi[x]
    if x != pi:
```

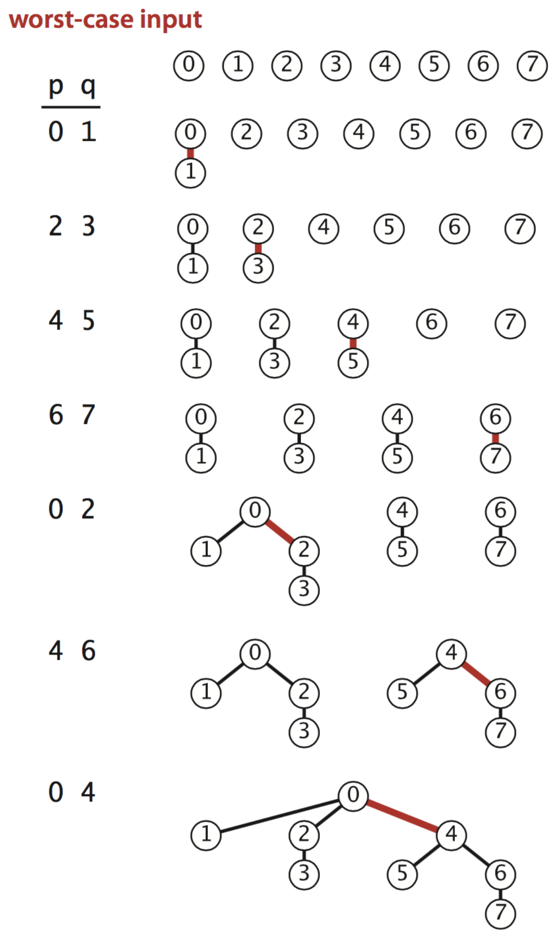


Fig. 21.1: Weighted quick-union traces

```
    self.pi[x] = self.find(pi)
  return self.pi[x]

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

21.8.3 Complexity

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

21.9 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. We don't pre-project the axis beforehand, since it will take $O(mn)$ to collect the projected 1d array. Instead, we only project it during binary search when checking the mid item. Checking takes $O(m)$, searching takes $O(\log n)$.
3. Do the same for y-axis.

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

21.10 MST

Minimum spanning tree.

21.10.1 Kruskal's algorithm

Core clues:

1. Vertices $v \in V$ are divided into different sets
2. Extract min edges to unionize the sets
3. Terminates when $\forall v \in V$ are in the same set.

Code:

```
def kruskal(G):
  ret = []
  uf = UnionFind()
  for v in G.V:
```

```
    uf.add(v)

  G.E.sort()  # sort by weights
  for u, v in G.E:
    if not uf.isunion(u, v):
      A.append((u, v))
      uf.unionize(u, v)
```

Complexity: $O(|E|\log|E|)$.

Chapter 22
Dynamic Programming

22.1 INTRODUCTION

The philosophy of dp:

1. The definition of **states**: redefine the original problem into relaxed substructure.
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.

State definitions. The state definition is the **redefinition** of the original problem as substructure.

Three general sets of state definitions of the substructure.

1. ends *at* index i (i **required**)
2. ends *before* index i (i **excluded**)
3. ends *at* or *before* index i (i **optional**)

22.1.1 Common programming 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

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

Firstly,

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

should be reduced to

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

Secondly,

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

should be reduced to

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

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.

Backtrace array. Normally dp returns the number of count/combinations. To reconstruct the result, store the parent index a backtrace array .

$$\pi[i] = j$$

22.2 SEQUENCE

22.2.1 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.

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

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

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

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

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 range(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 tow options: choose A_{i-1} or not.

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

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

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

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

H-index Given an array of citations A of a researcher, write a function to compute the researcher's h-index.

Need some re-representation of information:

1. Let C_i be the #paper with $= i$ citations.
2. Let F_i be the #paper with $\geq i$ citations.

$$F_i = F_{i+1} + C_i$$

DP takes $O(n)$. If it is sorted, use binary search to achieve $O(\lg n)$.

Interleaving String Given s, a, b find whether s is formed by the interleaving of a and b.

Let $F_{i,j}$ be $s[:i+j]$ is interleaved from $a[:i], b[:j]$.

We have to options to choose $s[i+j-1]$, either from $a[i-1]$ or from $b[j-1]$:

$$F_{i,j} = \Big(F_{i-1,j} \wedge s[i+j-1] = a[i-1]\Big) \\ \vee \Big(F_{i,j-1} \wedge s[i+j-1] = b[j-1]\Big)$$

Largest divisible subset. Given a list of distinct positive integers A, find the largest subset S such that every pair (S_i, S_j) of elements in this subset satisfies: $S_i \% S_j = 0$ or $S_j \% S_i = 0$.

Let F_i be the length of the divisible subset ending at A_i.

$$F_i = \max_{j: j<i, A_i \% A_j = 0} (1 + F_j)$$

Let π_i be the index of the previous element of A_i in the divisible subset. π_i is used to reconstruct the array.

$$\pi_i = \arg \max_{j: j<i, A_i \% A_j = 0} (1 + F_j)$$

22.2.2 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.

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

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. 22.1: Trapping Rain Water

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

The the total volume vol:

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

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.

22.2.3 Automata

Decode ways. 'A' encodes 1, 'B' 2, ..., 'Z' 26, Given an encoded message containing digits S, determine the total number of ways to decode it.

For example, given encoded message 12, it could be decoded as "AB" (1 2) or "L" (12). Thus, the number of ways decoding 12 is 2.

Let F_i be number of decode ways for $s[:i]$.

- If s_{i-1} is 0, we only have one way to decode: 10 or 20.

$$F_i = F_{i-2}$$

- If s_{i-1} is not 0, we have two one ways to decode: 1) $1 \sim 9$; or 2) $10 \sim 26$

$$F_i = F_{i-1} + F_{i-2}$$

```
if s[i-1] != "0":
    F[i] = F[i-1]
    if 10 <= int(s[i-2]+s[i-1]) < 27:
        F[i] += F[i-2]
else:  # 0 is special
    if s[i-2] in ("1", "2"):
        F[i] = F[i-2]
    else:
        return 0
```

Regex

22.3 Graph

22.3.1 Binary Graph

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_{ij} (lower right corner).

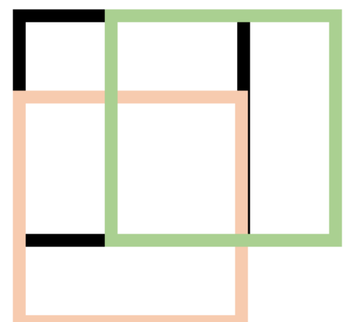


Fig. 22.2: Expand the maximal square

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

22.3.2 General Graph

Shortest path in graph containing negative weights. Find the shortest path from s to t.

Let $F_{n,v}$ be the shortest path from v to t with at most n vertices.

Then we can two options:

$$F_{n,v} = \min\Big(F_{n-1,v}, \min_{u \in Nbr}\left(F_{n-1,w} + c_{uv}\right)\Big)$$

, where c_{uv} is the weight cost on edge (u,v), Nbr is neighbors of v.

Notice that there should not be any negative cycle otherwise the path can be $-\infty$.

22.4 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.

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

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.

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

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 $s[i:j]$ is palindrome.

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

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

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 range(n+1)] for _ in range(n)]
for i in range(n):
    pa[i][i] = True
    pa[i][i+1] = True

for i in range(n-2, -1, -1):
    for j in range(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.

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

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

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

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

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

22.5 Divide & Conquer

22.5.1 Tree

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

```
 1          3      3       2       1
  \        /      /       / \       \
   3      2      1       1   3       2
  /      /        \                   \
 2      1          2                   3
```

Let F_i be the #BSTs constructed from i elements. The pattern of choosing one element as the root is:

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

Thus, in general,

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

22.5.2 Array

Burst balloons. Given n balloons, indexed from 0 to $n-1$. Each balloon is painted with a number on it represented by array A. You are asked to burst all the balloons. If the you burst balloon i you will get `A[left] * A[i] * A[right]` coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Core clues:

1. Divide & Conquer
2. Boundary definition
3. Reverse Thinking: think about n balloons if A_i is the last one to burst, what now? - Backward dp

Let $F_{i,j}$ be the max scores burst all over `A[i:j]`.

$$F_{i,j} = \max(F_{i,k} + A_{i-1}A_kA_j + F_{k+1,j} \cdot \forall k)$$

, where k is the last one to burst.

Notice that A_{i-1}, A_j go beyond $[i, j)$.

22.6 Knapsack

Knapsack problem is different from the sequence problem. It is a problem of **bag** rather than of sequence, since the order of element does not matter.

22.6.1 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.

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

Advanced backpack problem[3].

22.6.2 Sum - 0/1 Knapsack.

subset sum. Given a list of numbers A, find a subset (i.e. subsequence, not slice) that sums to target t.

Let $F_{i,v}$ be #subset of $A[: i+1]$ (ending at A_i), can be sum to target v.

You have two options: either select A_i or not.

$$F_{i,v} = F_{i-1,v-A_i} + F_{i-1,v}$$

Time complexity: $O(nk)$.

k sum. Similar to subset sum, but restrict the length of subset of k.

Given n distinct positive integers, integer k ($k \leq n$) and a number target. Find k numbers which sums to 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.

$$sum\binom{j}{i} = v$$

Let $F_{i,j,v}$ be 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.

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

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

22.7 Local and Global Extremes

22.7.1 Long and short stocks

22.7.1.1 At most k transactions

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

[3] Nine Lectures in Backpack Problem.

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.

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

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

Space optimization.

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

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,

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

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.

22.7.1.2 With cool down

Find the maximum profit. You may complete as many transactions as you like with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Let F_i be the max profit from day 0 to day i, selling stock at day i. (i.e. ended at)

Let M_i be the max profit from day 0 to day i. (i.e. ended at or brefore)

For F_i, at each day i, you have two options: 1) Sell the stock that has been held for multiple days. 2) Sell the stock held for 1 day. Notice the 1st option, it is equivalent to reverting the previous transaction, selling at day i instead of day $i-1$.

$$\begin{aligned} F_i = \max\big(& F_{i-1} + \Delta \\ & M_{i-2-CD} + \Delta\big) \end{aligned}$$

, where $CD = 1$, the cool down time, $\Delta = A_i - A_{i-1}$

For M_i, simply,

$$M_i = \max(M_{i-1}, F_i)$$

22.8 GAME THEORY - MULTI PLAYERS

Assumption: the opponent take the optimal strategy for herself.

22.8.1 Coin game

Single 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.

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

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 either of the ends of the line until there are no more coins left. The player with the larger amount of money wins.

let $F^{p}_{i,j}$ 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.

$$
\begin{aligned}
F^{p}_{i,j} = \max\big(& A_i + S[i+1:j] - F^{p'}_{i+1,j}, \\
& A_j + S[i:j-1] - F^{p'}_{i,j-1}\big)
\end{aligned}
$$

Chapter 23
Interval

23.1 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.

$$|\leftarrow p$$

23.2 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):
    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.
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)].end)
        # itvls[-len(right)-1]

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

23.3 Event-driven algorithms

23.3.1 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.

23.3.2 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. And we need min-heap to pop the early ones.

```
def max_overlapping(intervals):
    maxa = 0
    intervals.sort(key=lambda x: x.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 24
General

24.1 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.

Small samples. Try out with some small input sample.

Corner cases. Atypical input.

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.

subsarray 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