github.com/algorhythms/Algo-Quicksheet
Author: idf@github

Algorithm Quicksheet

Classical equations, diagrams and patterns in algorithms

October 29, 2025

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 / Approaches, 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, approximated value
$\arg\max_x f(x)$	Argmax: the value x that maximizes f
$\binom{n}{k}$	n choose k , equal to $\frac{n!}{k!(n-k)!}$
$\text{range}(i, j)$	Range of number from i (inclusive) to j (exclusive)
$A[i:j]$	Subarray consist of $A_i, A_{i+1}, ..., A_{j-1}$.

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.
If $a = b$, then it reduces to

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

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:

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

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

Then:

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

1.3 Useful Math Equations

Logarithm.

$$\log_b mn = \log_b m + \log_b n$$
$$\log_b m^n = n \log_b m$$
$$\log_b a = \frac{\ln a}{\ln b}$$

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:

$$a^{\log_b n} = n^{\log_b a}$$
$$\Leftarrow \ln a^{\log_b n} = \ln n^{\log_b a}$$
$$\Leftarrow \frac{\ln n}{\ln b} \ln a = \frac{\ln a}{\ln b} \ln n$$

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

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

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

# defaultdict take constructor func as param
counter = defaultdict(int)
G = defaultdict(list)  # graph of v -> neighbors
G = defaultdict(dict)  # G[s][e] -> weight
G = defaultdict(lambda: defaultdict(int))  # G[s][e] -> weight
```

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

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

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

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]
2. Thus, `stk[-1]` stores the last unpaired bracket.

[2] The stack should always hold indices, rather than values.

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
```

3.4 Map

3.4.1 Math relations

1-1 Map. Mathematically, full projection. One map, dual entries.

```
class OneToOneMap:
    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)
```

3.5 Monotonic Stack

Ref array 9.3

3.6 Monotonic Queue

Ref array 9.4

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

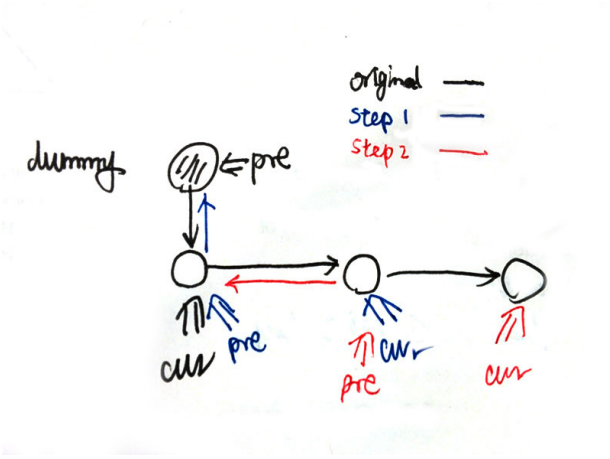


Fig. 4.1: Reverse the linked list

```
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 Merge K Linked List

Given an array of k linked-lists lists, each linked-list is sorted in ascending order. Merge all the linked-lists into one sorted linked-list and return it.

Core Clues:

1. Relax the problem: if only two lists, just compare and merge like merge sort
2. k lists $\Rightarrow$ Heap

```
use heap
```

```
--------------------
    | | | | | | |
    | | | | | | |
    | | | | | | |
    | | | | | | |
```

```
def mergeKLists(self, lists):
  h = []
  for node in lists:
    if node:
      heapq.heappush(h, (node.val, node))

  dummy = ListNode(0)
  current = dummy  # verbose name for return
  while h:
    val, mini = heapq.heappop(h)
    current.next = mini
    nxt = mini.next
    if nxt:
      heapq.heappush(h, (nxt.val, nxt))

    current = current.next

  return dummy.next
```

4.2.2 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:
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.pre = None
        self.next = None


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

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

        return -1

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

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

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

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

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

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

        return last

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

        self._appendleft(cur)
```

Use `OrderedDict`:

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity: int) -> None:
        self.capacity = capacity
        self.kv = OrderedDict()  # key -> value

    def get(self, key: int) -> int | None:
        if key not in self.kv:
            return None  # or raise KeyError
        # Move to MRU position
        self.kv.move_to_end(key, last=False)
        return self.data[key]

    def put(self, key: int, value: int) -> None:
        # If key exists, update value and move to MRU
        if key in self.kv:
            self.kv.move_to_end(key, last=False)
        self.kv[key] = value

        # Evict LRU if over capacity
        if len(self.kv) > self.capacity:
            self.kv.popitem(last=True)   # pop LRU (right side)
```

First Unique Number in the stream. Naive:

```python
class FirstUnique:
  def __init__(self, A):
    self.cnt = Counter()
    self.q = deque()
    for a in A:
      self.add(a)

  def showFirstUnique(self) -> int:
    while self.q and self.cnt[self.q[0]] > 1:
      # no need to dec counter
      self.q.popleft()
    return self.q[0] if self.q else -1

  def add(self, value: int) -> None:
    self.cnt[value] += 1
    self.q.append(value)
```

Using Double Linked List:

```python
from collections import OrderedDict

class FirstUnique:
  def __init__(self, A):
    self.cnt = defaultdict(int)
    self.uniques = OrderedDict()   # Ordered List
    for x in A:
      self.add(x)

  def showFirstUnique(self) -> int:
    return next(iter(self.uniques)) if self.uniques else -1

  def add(self, value):
    self.cnt[value] += 1

    if self.cnt[value] == 1:
      # first time: becomes unique; append to end
      self.uniques[value] = None
    elif self.cnt[value] == 2:
      self.uniques.pop(value, None)
```

Using Double Linked List without OrderedDict:

1. Maintain a map: val $\rightarrow$ node if seen once; None if unseen; DUP if seen ≥ 2;

```python
@dataclass
class Node:
  val: int
  prev: Node | None = None
  next: Node | None = None

class DoubleLinkedList:
  def __init__(self):
    self.head = Node(0)
    self.tail = Node(0)
    self.head.next = self.tail
    self.tail.prev = self.head

  def append(self, node):
    last = self.tail.prev

    node.prev = last
    node.next = self.tail

    last.next = node
    self.tail.prev = node
    return node

  def remove(self, node):
    prev = node.prev
    nxt =  node.next

    prev.next = nxt
    nxt.prev = prev

  def first(self):
    return self.head.next.val \
      if self.head.next is not self.tail else None

DUP = object()

class FirstUnique:
  def __init__(self, A):
    self.dll = DoubleLinkedList()
    # val -> node if seen once; None if unseen; DUP if seen >= 2;
    self.nodes = {}
    for x in A:
      self.add(x)

  def showFirstUnique(self) -> int:
    v = self.dll.first()
    return v if v is not None else -1

  def add(self, value: int):
    if value not in self.nodes:
      node = self.dll.append(Node(value))
      self.nodes[value] = node
    elif self.nodes[value] is not DUP:
      # seen once before
      self.dll.remove(self.nodes[value])
      self.nodes[value] = DUP
    else:
      # DUP
      pass
```

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

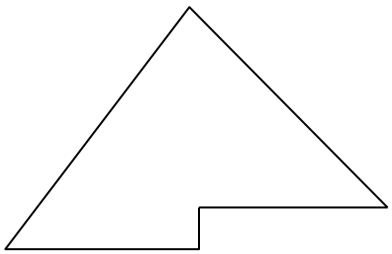


Fig. 5.1: Heap

5.2 Applications

5.2.1 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:

```
from dataclasses import dataclass

@dataclass
class HeapValue:
    val: int
    deleted: bool = False  # lazy delete

    def __lt__(self, other):  # reverse for max-heap
        return not self.val < other.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)
```

5.2.2 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
);
```

5.2.3 Stale-Checking Heap for Update

If introduceing **update** to the values in the heap, we can do stale checking as lazy validation.

Consider the following: $freq_i$ represents the frequency of A_i at step i. We want to keep track of the most frequent number at each step i. A contains duplicagtes. $freq_i$ can be negative.

```
def most_frequent_num(self, A, freq):
    counter = defaultdict(int)
    h = []
    ret = []
    for n, cnt in zip(A, freq):
        counter[n] += cnt
        heapq.heappush(h, (-counter[n], n))
        while True:
            c, maxa = h[0]
            if c != -counter[maxa]:  # stale check
                heapq.heappop(h)
            else:
                ret.append(maxa)
                break

    return ret
```

5.2.4 Maintain min max in Sliding Window

Count number of continuous subarrays in A s.t. within the subarray $|A_i - A_j| \leq 2, \forall i, j$. **Core clues**

1. Need to maintain max and min in the sliding window $\Rightarrow$ heap
2. In sliding window i, j, count of all valid subarrays ending AT j $\Rightarrow j - i + 1$

```
def continuousSubarrays(self, A):
    ret = 0
    i = 0  # starting index of the window
    j = 0  # ending index of the window
    h_min = []  # (a, idx)
    h_max = []  # (-a, idx)
    while j < len(A):
        heapq.heappush(h_min, (A[j], j))
        heapq.heappush(h_max, (-A[j], j))
        while -h_max[0][0] - h_min[0][0] > 2:
            # shrink
            i += 1
            while h_min[0][1] < i:
                heapq.heappop(h_min)
            while h_max[0][1] < i:
                heapq.heappop(h_max)

        ret += j-i+1
        j += 1

    return ret
```

Note: 3-layred nested `while` loop.

5.2.5 Find k smallest/largest

Partial Quicksort 7.4.1

5.3 Derivatives

5.3.1 Heap of Linked Lists

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

5.4 Inside the Library

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()

5.4.1 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 $a[3k+1]$.

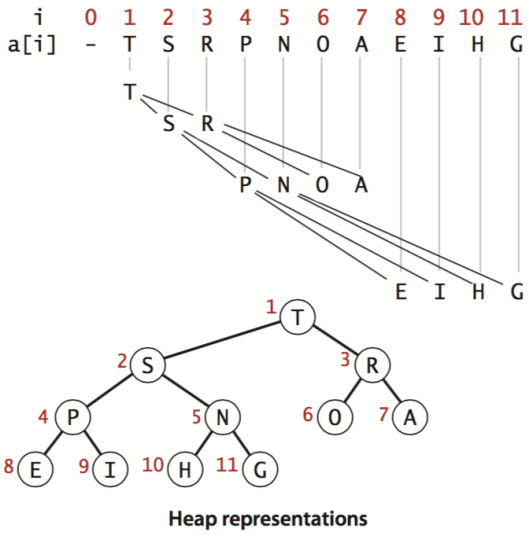


Fig. 5.2: Heap representation

5.4.2 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.

5.4.3 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
```

5.4.4 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:

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

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

$$= O(n)$$

Chapter 6
Tree

6.1 Binary Tree

6.1.1 Recursion

The basic opeartion with tree is recursion.

Information Propagation. To propagate information:

- To propagate children's information to current node ⇒ use return values of recursion calls.
- To propagate current node's information to children ⇒ use parameters for recursion calls
- To propagate current node's or children's information to global memory ⇒ use `self` or `global` vars.

Condition checking. To check the predicate, we normally check the predicate upon entering the recursion calls for trees as terminal condition. For graphs, we normally check the predicate before entering the recursion calls.

6.1.2 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. Propagate depth information to children.

```
def get_depth(self, cur, depth):
    if not cur:
        return depth

    return max(
        self.get_depth(cur.left, depth+1),
        self.get_depth(cur.right, depth+1),
    )
```

Max depth. Propagate depth information back to current node / parent.

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

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

    # do something

    return depth
```

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
```

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),
        )
```

Alternatively,

```
def minDepth(cur) -> int:
  if not cur:
    return 0
  if not cur.left and not cur.right:
    return 1

  left = minDepth(cur.left) if cur.left else sys.maxsize
  right = minDepth(cur.right) if cur.right else sys.maxsize
  return 1 + min(left, right)
```

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):
    # post function call check
    if not cur:
      return
    if self.found:
      return

    path.append(cur)
    if cur == target:
        self.found = True

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

The `found` is an obj's field of 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
```

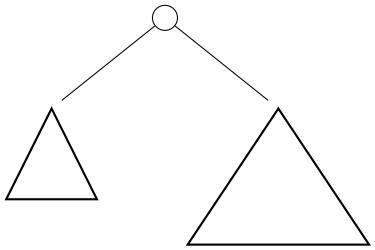
Method 1: In general binary tree, construct the path from root to $node_1$ and $node_2$ respectively, and **diff** the two paths. Time complexity: $O(\lg n)$, space complexity: $O(\lg n)$.

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 general bindary tree and to achieve $O(1)$ space, bottom up to find the first node that has p, q in its left and right substrees simultaneously.

A node is lca of itself.



```
def find_either(self, node, p, q):
    if node is None:
        return None

    # a node is lca of itself
    if node == p or node == q:
        return node

    left = self.find_either(node.left, p, q)
    right = self.find_either(node.right, p, q)

    # lca
    if left and right:
        self.ans = node
        return node

    return left if left else right
```

Method 4: `count` can be extended to find the LCA of a set of nodes

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

    l = self.count(node.left, p, q)
    r = self.count(node.right, p, q)
    m = 1 if node in (p, q) else 0
    ret = l + m + r
    if ret == 2 and not self.ans:
        # only keep the lowest
        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. **Core Clues:**

1. Greedy go left? $\Rightarrow$ a counter case $\Rightarrow$ need to check both sides.

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

Rightmost node can be similarly found.

Diameter of a tree (graph). The diameter of a tree $\equiv$ longest path in the tree.

Core clues:

1. Longest path $\Rightarrow$ Intuitily, we need to find one end of the edge, and then find the other end.
2. **Edge** $\Rightarrow$ Edge to edge, furtherest to frutherest.
3. **Find one end of the edge** $\Rightarrow$ start from any vertex, bfs to reach the farthest leaf.
4. **Find the other end** $\Rightarrow$ start from this leaf node, bfs to reach the other farthest leaf.

```
_, _, last = self.bfs(0, G)
level, pis, last = self.bfs(last, G)

def bfs(self, s, G) -> Tuple[int, List[Vertex], Vertex]:
    # bfs
    visited = defaultdict(bool)
    # predecessor
    pis = defaultdict(lambda: -1)
    last = s  # last visited
    level = 0
    q = [s]
    while q:
        new_q = []
        for e in q:
            last = e
            visited[e] = True
            for nbr in G[e]:
                if not visited[nbr]:
                    pis[nbr] = e
                    new_q.append(nbr)

        q = new_q
        level += 1

    return level, pis, last
```

Alternatively, use tree:

```
def diameterOfBinaryTree(self, root):
    self.ret = -1
    self.dfs(root)
    return self.ret

def diam(self, subroot):
    # get the longest path to any leaf node
    if not subroot:
        return 0

    left = self.diam(subroot.left)
    right = self.diam(subroot.right)
    self.ret = max(self.ret, left + right)
    return max(left, right) + 1  # +1 to make progress
```

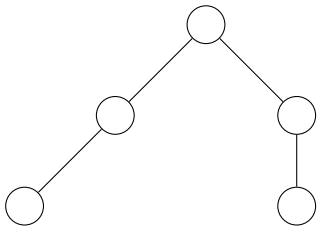
Distance to its leaves. Counstruct a map of distance $\rightarrow$ count. The distance is from current node to leaves, and the count is number of leaves.

Core Clues:

1. Objective: maintain a map from distance $\rightarrow$ to count.
2. Each: each node has a map of distance $\Rightarrow$ we need return value on the recursion stacks rather than a value passed in the recursion.
3. Leaf: the leaf node has a base case - distance to itself is 0

```
def distances(self, node) -> dict:
    counts = defaultdict(int)  # distance -> count
    if not node:
        return counts

    if not node.left and not node.right:
        counts[0] = 1
        return counts

    left = self.distances(node.left)
    right = self.distances(node.right)
    for k, v in left.items():
        for K, V in right.items():
            self.consume(k, K, v, V)

    # merge
    for k, v in left.items():
        counts[k+1] += v
    for k, v in right.items():
        counts[k+1] += v

    return counts
```

Max Distance to a target node. Node to node distance within a tree. Find the maximum distane from a node to a target node. The start node and cabe any node, $\exists node$. Not necessarily from the root.



Core Clues:

- Convert the tree to graph, and BFS from the target node is trivial, but it requires 2 passes
- To solve it in 1 pass, we need to dfs:
  1. Maintain a depth from current node to the target node **or** leaf.
  2. If root is the target $\Rightarrow$ get max depth from left and right subtrees
  3. If target is on one (e.g. left) side of the root $\Rightarrow$ depth to the target + right depth to leaf

```
self.maxa

# depth from a node to target or leaf
def dfs(self, cur, target) -> bool, int:
    if not cur:
        return False, -1

    l_found, l_depth = self.dfs(cur.left, target)
    r_found, r_depth = self.dfs(cur.right, target)

    if cur.val == target:
        depth = max(l_depth, r_depth) + 1
        self.maxa = max(self.maxa, depth)
        return True, 0

    if l_found:
        dist = (l_depth + 1) + (r_depth + 1)   # from child to cu
        self.maxa = max(self.maxa, dist)
        return l_found, l_depth + 1

    if r_found:
        dist = l_depth + r_depth + 2  # from child to cur
        self.maxa = max(self.maxa, dist)
        return r_found, r_depth + 1

    return False, depth
```

Longest Consecutive Sequence. A consecutive path is a path where the values of the consecutive nodes in the path differ by one. This path can be either increasing or decreasing.

Core Clues:

1. Break down the problem: treat asc and dec separately
2. Let *inc* with the length of increasing subarray ending at *node*

```
class Solution:
  def longestConsecutive(self, root) -> int:
    self.ret = 0
    self.dfs(root)
    return self.ret

  def dfs(node) -> Tuple[int, int]:
    if not node:
      return 0, 0  # inc, dec

    ginc = gdec = 1

    linc, ldec = dfs(node.left)
    rinc, rdec = dfs(node.right)

    inc = dec = 1
    if node.left and node.left.val == node.val + 1:
        inc = max(inc, linc + 1)
    if node.right and node.right.val == node.val - 1:
        dec = max(dec, rdec + 1)
    self.ret = max(self.ret, inc + dec - 1)
    ginc = max(ginc, inc)
    gdec = max(gdec, dec)

    # mirrored
    inc = dec = 1
    if node.left and node.left.val == node.val - 1:
        dec = max(dec, ldec + 1)
    if node.right and node.right.val == node.val + 1:
        inc = max(inc, rinc + 1)
    self.ret = max(self.ret, inc + dec - 1)
    ginc = max(ginc, inc)
    gdec = max(gdec, dec)

    return ginc, gdec
```

6.1.3 Vertical Traversal

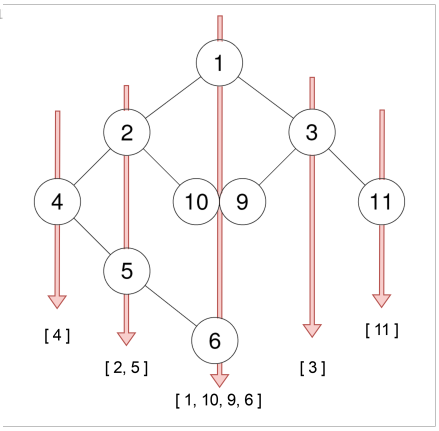


Fig. 6.1: Vertical Traversal

Core Clues :

1. Vertical traversal ⇒ columns, starting from root as 0, there can be negative and positive offset
2. Order within each column ⇒ level by level, thus BFS required

```
def verticalOrder(self, root):
    l = self.leftmost(root, 0)
    r = self.rightmost(root, 0)

    ret = [[] for _ in range(r-l-1)]
    self.bfs(root, -l-1, ret)
    return ret

def bfs(self, cur, col, ret):
    q = []
    if cur:
        q.append((cur, col))

    while q:
        new_q = []
        for e in q:
            v, c = e
            ret[c].append(v.val)
            if v.left:
                new_q.append((v.left, c-1))
            if v.right:
                new_q.append((v.right, c+1))
```

```
        q = new_q

def leftmost(self, cur, col):
    if not cur:
        return col
    return min(
        self.leftmost(cur.left, col-1),
        self.leftmost(cur.right, col+1)
    )

def rightmost(self, cur, col):
    if not cur:
        return col
    return max(
        self.rightmost(cur.left, col-1),
        self.rightmost(cur.right, col+1),
    )
```

If using `defaultdict(list)` with negative/positive offset, the order prefers left.

```
def verticalOrder(self, root) -> List[List[int]]:
    self.lists = defaultdict(list)
    self.dfs(root, 0)
    # re-adjust
    floor = min(self.lists.keys())
    ret = [[] for _ in range(len(self.lists))]
    for k, v in self.lists.items():
        ret[k - floor] = v
    return ret

def dfs(self, cur, offset):
    if not cur:
        return

    self.lists[offset].append(cur.val)
    self.dfs(cur.left, offset-1)
    self.dfs(cur.right, offset+1)
```

6.1.4 Morris Traversal

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

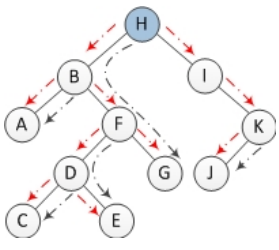


Fig. 6.2: Morris traversal time complexity

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.

6.1.4.1 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 not threaded (i.e. no `pre` right child), assign it to `cur`; go left
   b. If threaded, *consume* `cur`, go right. ($\equiv$ 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
```

6.1.4.2 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. ($\equiv$ no left, but no *consume*, since consume before).

Code:

```
def morris_preorder(self, root):
    cur = root
```

[3] ref

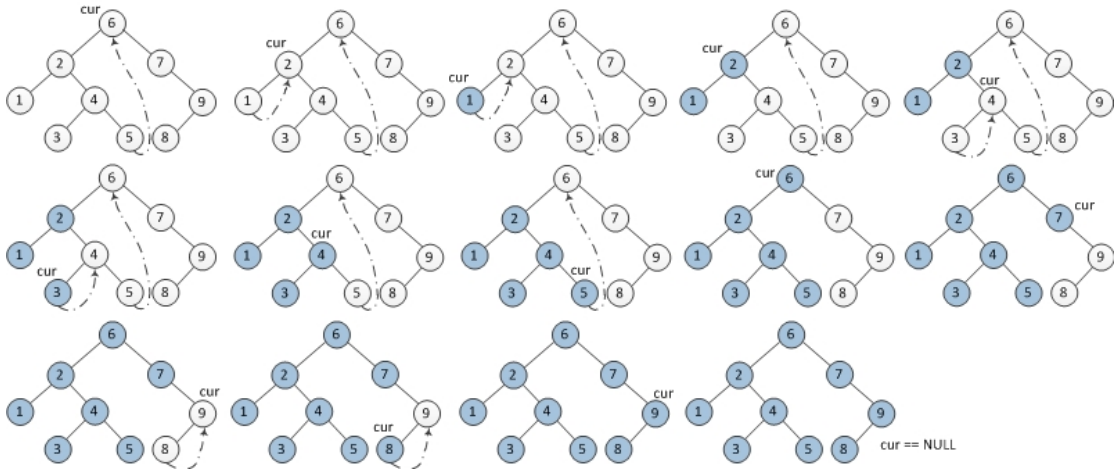


Fig. 6.3: Morris in-order traversal

```
    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
```

6.1.4.3 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):
    self._reverse(fr, to)

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

    self._reverse(to, fr)
```

6.1.5 Zig Zag Traversal

To zig zag traverse a tree, maintain a `is_left` to determine direction.

Find the longest zig zag path. starting from any node in the tree.

Core Clues:

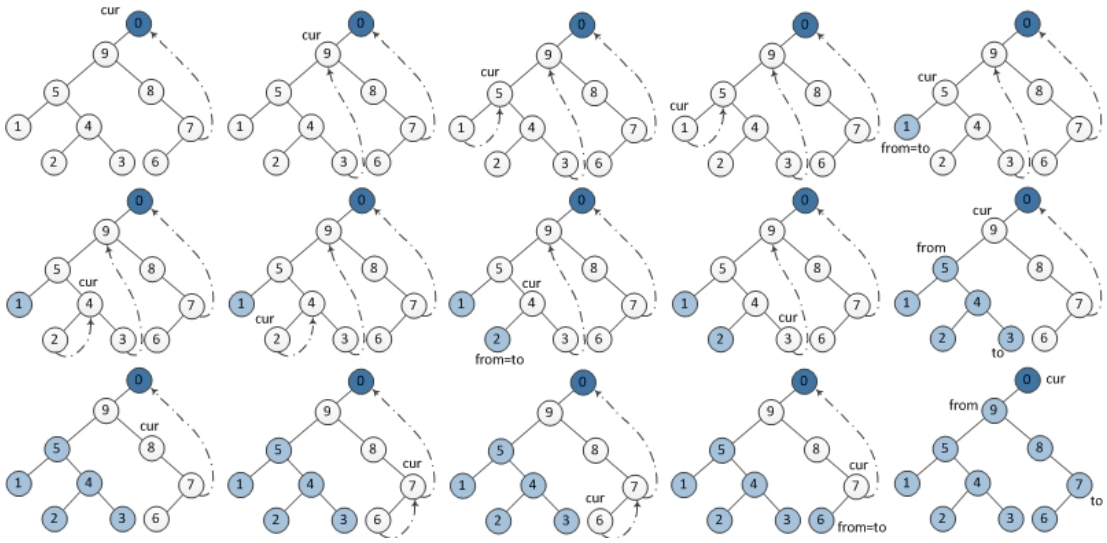


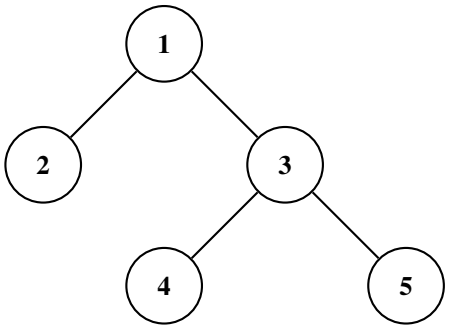
Fig. 6.4: Morris post-order traversal

1. To determine direction ⇒ use `is_left`
2. To check every node ⇒ restart the path with the current node.

```
def longestZigZag(self, root):
    self.maxa = 0
    self.dfs(root, True, 0)
    self.dfs(root, False, 0)  # redundant
    return self.maxa - 1

def dfs(self, node, is_left, l):
    if not node:
        return

    l += 1
    self.maxa = max(self.maxa, l)

    if is_left:
        self.dfs(node.right, False, l)
        # restart with the current node
        self.dfs(node.left, True, 1)
    else:
        self.dfs(node.left, True, l)
        # restart with the current node
        self.dfs(node.right, False, 1)
```

Although in if-else statement, the code is the same, but the structure shows the logic.

Binary Tree Serialization and Deser.



Serialized:

`[1,2,3,null,null,4,5,null,null,null,null]`

1. Go through serialization by BFS.
2. Ser by BFS can be either post-order or pre-order. It is straightforward.
3. Deser by BFS, to know the parent ⇒ it has to be pre-order BFS (encode before enqueue the q).

```
class Codec:
  def serialize(self, root):
    if not root:
      return "null"

    ret = [str(root.val)] # add result when enqueue
    q = [root]
    while q:
      new_q = []
      for cur in q:
        if cur.left:
          new_q.append(cur.left)
        ret.append(self.encode(cur.left))

        if cur.right:
          new_q.append(cur.right)
        ret.append(self.encode(cur.right))
```

```
      q = new_q

    return ",".join(ret)

  def deserialize(self, data):
    lst = data.split(",")
    root = self.decode(lst[0])  # decode when push in queue

    q = [root]
    i = 1  # simple BFS from q won't work
    while q:
      new_q = []
      for cur in q:
        if i < len(lst):
          cur.left = self.decode(lst[i])
          i += 1
          if cur.left:
            new_q.append(cur.left)

        if i < len(lst):
          cur.right = self.decode(lst[i])
          i += 1
          if cur.right:
            new_q.append(cur.right)

      q = new_q

    return root

  def decode(self, s):
    if s == "null":
      return None
    else:
      return TreeNode(int(s))

  def encode(self, node):
    if not node:
      return "null"
    else:
      return str(node.val)
```

6.2 Tree to Graph

Minimum Edge Reversals So Every Node Is Reachable. There is a simple directed graph with n nodes labeled from 0 to $n-1$. The graph would form a **tree** if all its edges were bi-directional.

For every node i, independently calculate the minimum number of edge reversals required so it is possible to reach any other node starting from node i through a sequence of directed edges.

Core Clues:

1. Transform the tree to an undirected graph $\Rightarrow$ need to check predecessor *pi* to avoid cycles
2. Edge weight: whether to reverse the direction
3. Relax the problem: check one root is simple graph traversal
4. Check all the roots: naively check every node independently. $\Rightarrow$ reuse the calculated information for adjacent nodes. After calculae results for ret_u
   a. if $u \rightarrow v$, $ret_v = ret_u + 1$ since we need to reverse the edge
   b. if $u \leftarrow v$, $ret_v = ret_v - 1$ since we don't need to reverse the edge but we need to undo the reversal included in ret_v.

```
class Solution:
  def minEdgeReversals(self, n, edges) -> List[int]:
    G = defaultdict(list)
    for u, v in edges:
      G[u].append((v, 0))
      G[v].append((u, 1))

    ret = [0 for _ in range(n)]
    # 1) traverse one node
    def dfs_sum(u, pi) -> int:
      total = 0
      for v, w in G[u]:
        if v != pi:
          total += w + dfs_sum(v, u)
      return total

    ret[0] = dfs_sum(0, -1)

    # 2) reroot
    def dfs_reroot(u, pi):
      for v, w in G[u]:
        if v != pi:
          if w == 0:
            ret[v] = ret[u] + 1
          else:
            ret[v] = ret[u] - 1

          dfs_reroot(v, u)

    dfs_reroot(0, -1)
    return ret
```

6.3 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.

6.3.1 Property

$\forall$ 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:
    def __init__(self, sz, lo, hi):
        self.sz = sz
        self.lo = lo
        self.hi = hi
```

6.3.2 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.

***K*-th Smallest element in a BST** . **Core Clues**:

1. Left tree smaller than root. Right tree larger than the root
2. Count the size of the subtree

```
class Solution:
  def kthSmallest(self, root, k):
    l = self.cnt(root.left)
    if l+1 == k:
      return root.val

    elif l+1 < k:
      return self.kthSmallest(root.right, k-(l+1))
    else:
      return self.kthSmallest(root.left, k)

  def cnt(self, root):
    if not root:
      return 0

    return self.cnt(root.left) + 1 + self.cnt(root.right)
```

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 that 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:
  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:
  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:
  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 A.2. However, there is a more elegant way using Segment Tree - Section 6.5.3.

6.3.3 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)
```

6.4 Binary Index Tree (BIT)

Compared to normal sorted array. Although we can find a target in $O(\lg N)$ but the insertion of it takes $O(N)$. BST is used to bisect find element and insert/update element in $O(\lg N)$. BIT is simplified version ot it.

6.4.1 Introduction

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

Compared to Segment Tree 6.5, 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

$$prefixSum \nrightarrow i$$

6.4.2 Implementation

Binary Representation Trick: If you write an index i in binary, the least significant set bit (LSB) of i tells you how many elements $rangeSum_i$ should sum over. Mathematically, if $i = 12, 0b1100$, $LSB(12) = 4, 0b0100$. This means $rangeSum_{12}$ stores the sum of the 4 elements ending at index 12 in the original array A, i.e. $A_9 + A_{10} + A_{11} + A_{12}$

Define a $rangeSum_i$:

$$rangeSum_i = A_i + A_{i-1} + A_{i-n}, \text{ where } n = LSB(i)$$

More formally

$$rangeSum_i = \sum_{\substack{j=i \\ \Delta:-1}}^{i-lsb(i)} A_j$$

Non-Overlapping Partial Sums: Because each element $fenw_i$ covers a subrange determined by $LSB(i)$, these subranges can be combined to get a prefix sum, and they

do not double count any elements. Essentially, when you want the prefix sum up to index i, you move from i backward in jumps of $LSB(i)$ and sum the relevant *rangeSum* entries:

$$prefixSum_i = \sum_k rangeSum_k, \text{ where } k \leftarrow k - LSB(k),$$

until k becomes 0.

More formally for querying $prefixSum_i$:

$$prefixSum_i = \sum_{\substack{k=i \\ \Delta:-lsb(k)}}^{1} rangeSum_k$$

For the range, we use $(j,i]$ here instead of $[j,i)$ since more elegant for `query(i)` and `update(i, v)`

Core Clues:

1. Binary Representation Trick.
2. Low bit, or Least Significant Bit. `LSB(i)=i&-i`
3. BIT uses array index starting from **1**, because 0 doesn't have *lowbit* $\Rightarrow$ 0 is the dummy root.
4. 16 is also dummy in the graph

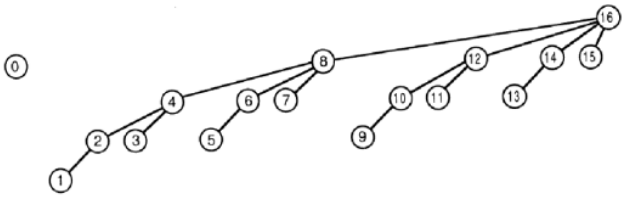


Fig. 6.5: Binary Indexed Tree *update* Operation. e.g. update $i = 10$

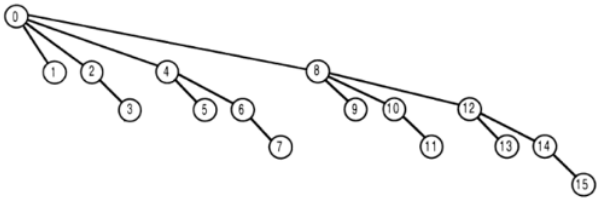


Fig. 6.6: Binary Indexed Tree *query* Operation. e.g. query $i = 10$

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 $O(\log_2 n)$.

```
class BIT:
  def __init__(self, n):
    """
    BIT uses index starting from 1
    0 is the dummy root
    """
    self.S = [0 for _ in range(n+1)]

  def __lsb(self, i):
    """
     i    =  0000 1100   (decimal 12)
    ~i    =  1111 0011
    -i    =  1111 0100   (2's complement, ~i+1)
    ----------------------------------
    & ret =  0000 0100   (decimal 4)
    """
    return i & -i

  def update(self, i, val):
    while i < len(self.S):
      self.S[i] += val
      i += self.__lsb(i)

  def query(self, i):
    ret = 0
    while i > 0:
      ret += self.S[i]
      i -= self.__lsb(i)

    return ret
```

6.4.3 For Rank

Count of Smaller Numbers After Self. Input: $[5,2,6,1]$. Output: $[2,1,1,0]$. **Core Clues:** Maintain a BIT as a map from the rank to accumulative count.

```
def count_smaller(A):
    ranks = sorted(set(A))
    R = {}  # 1-indexed rank
    for i, v in enumerate(ranks):
        R[v] = i+1

    tree = BIT(len(ranks))
    ret = deque()
    for i in range(len(A)-1, -1, -1):
        r = R[A[i]]
        count = tree.query(r-1)
        # strictly smaller, thus not r-1
        ret.appendleft(count)
        tree.update(r, 1)  # add count of 1

    return ret
```

Similarly, scanning from left to right, we can count of larger numbers before self - input: $[4,5,2,7,3,2,1]$, output: $[0,0,2,0,3,4,6]$.

6.4.4 Binary Index Tree & Inversion Count

Given A, calculate each element's inversion number.

Construct a BIT 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 a tree of **values as indices** with count the number inversions of at each value in range.
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
```

6.5 Segment Tree

6.5.1 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.

6.5.2 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
fold = lambda x, y: x+y

class Node:
  def __init__(self, lo, hi, val):
    self.lo, self.hi = lo, hi
    self.val = val
    self.left, self.right = None, None


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

  def build_tree(self, lo, hi) -> Node:
    """
    Bottom-up build
    segment: [lo, hi)
    Either check lo==hi-1 or have root.right
    only if have root.left
    """
    if lo >= hi:
      return

    if lo == hi-1:
      return Node(lo, hi, self.A[lo])

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

    # build subroot
    val = DEFAULT
    if left:
      val = fold(val, left.val)
    if right:
      val = fold(val, right.val)

    subroot = Node(lo, hi, val)
    subroot.left  = left
    subroot.right = right

    return subroot
```

```
  def query(self, subroot, lo, hi) -> int:
    """
    when querying, lo hi unchanged
    """
    if not subroot:
      return DEFAULT

    if hi <= subroot.lo or lo >= subroot.hi:
      return DEFAULT

    if lo <= subroot.lo and subroot.hi <= hi:
      return subroot.val

    l = self.query(subroot.left,  lo, hi)
    r = self.query(subroot.right, lo, hi)
    return fold(l, r)

  def modify(self, subroot, idx, val):
    if not subroot or idx < subroot.lo or idx >= subroot.hi:
      return

    if idx == subroot.lo and idx == subroot.hi-1:
      subroot.val = val
      self.A[idx] = val
      return

    self.modify(subroot.left,  idx, val)
    self.modify(subroot.right, idx, val)

    val = DEFAULT
    if subroot.left:
      val = fold(val, subroot.left.val)
    if subroot.right:
      val = fold(val, subroot.right.val)

    subroot.val = val
```

The above code abstracts out segment tree function using `lambda`.

6.5.3 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 $idx \to sum$, instead $idx \not\to sum$.

Core clues:

1. Segment Tree maintains a tree of **values as nodes** with a range $[lo, hi)$.
2. Define `cnt_this` as inversions at current node and `cnt_left` as inversions at left substree. Both values are mutually exclusive
3. We define the indices as the values A themselves. To get the inversion count of $a \Rightarrow$ `get(root, end)` $-$ `get(root, a)`.

```
class Solution:
  def build_tree(self, A):
    st = SegmentTree()  # frequency vector
    mini, maxa = min(A), max(A)
    # maxa+1 is the end dummy
    st.root = st.build(mini, maxa+1)
    return st

  def countOfLargerElementsBeforeElement(self, A):
    st = self.build_tree(A)
    ret = []
    end = max(A) + 1
    for a in A:
      inversion = \
        st.query(st.root, end) - st.query(st.root, a)
      ret.append(inversion)
      st.update(st.root, a, 1)

    return ret

@dataclass
class Node:
    cnt_this: int
    cnt_left: int
    lo: int
    hi: int
    left: Node | None
    right: Node | None

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

  def build(self, lo, hi) -> Node:
    if lo >= hi:
      return

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

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

  def query(self, subroot, i) -> int:
    if subroot.lo == i and subroot.hi-1 == subroot.lo:
      return subroot.cnt_left
    elif i < (subroot.lo+subroot.hi) // 2:
      return self.query(subroot.left, i)
    else:
      return subroot.cnt_left + subroot.cnt_this \
        + self.query(subroot.right, i)
```

6.5.4 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 num-

ber. 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 where 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 Solution:
  def reconstruct(self, A):
    """
    Given an array of tuples (val, inv)
    """
    # empty slots
    n = len(A)
    st = SegmentTree()
    st.root = st.build(0, n)

    ret = [0 for _ in range(n)]
    # from smallest to largest
    for a in sorted(A, key=lambda a: a.val):
      # +1 to include itself as the inv
      idx = st.find_delete(st.root, a.inv + 1)
      ret[idx] = a.val

    return ret

@dataclass
class Node:
  lo: int
  hi: int
  # size of empty slots
  cnt: int  # in range [lo, hi)
  left: Node | None = None
  right: Node | None = None

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

  def build(self, lo, hi):
    if lo >= hi:
      return
    if lo == hi-1:
      # leaf empty slot
      return Node(lo, hi, 1)

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

  def find_delete(self, root, sz) -> int:
    root.cnt -= 1  # delete
    if not root.left and not root.right:
      return root.lo

    lcnt = root.left.cnt if root.left else 0
    if sz <= lcnt:
      return self.find_delete(root.left, sz)
    else:
      return self.find_delete(root.right, sz - lcnt)

if __name__ == "__main__":
  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? If inversion is strictly greater $>$, then the above algorithm still works.

If inversion is $\geq$, then use a `Counter` to count the duplicate items already in the result.

```
placed = Counter()
for a in sorted(A, key=lambda x: x.val):
    # adjust target rank to skip equals already placed
    k = a.inv - placed[a.val] + 1
    idx = st.find_delete(st.root, k)
    ret[idx] = a.val
    placed[a.val] += 1
```

6.6 TRIE

6.6.1 Basic

Trie is aka radix tree, prefix tree.

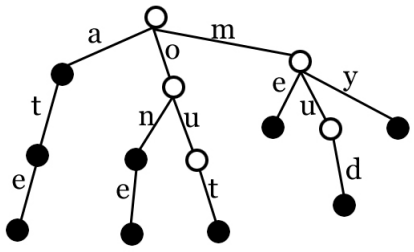


Fig. 6.7: Trie

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


class Trie:
    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
```

6.6.2 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.

```
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
```

6.6.3 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)))
```

6.6.4 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
```

6.6.5 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.

6.6.6 Applications

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

In memory file system:

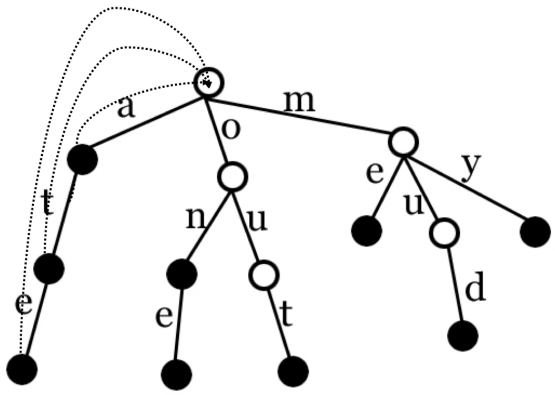


Fig. 6.8: Trie with threads from ending point to root

```python
@dataclass
class Node:
  is_file: bool = False
  name: str = ""
  children: Dict[str, "Node"] = field(default_factory=dict)
  # not directly children: dict = {} since {} is shared
  content: str = ""


class FS:
  def __init__(self):
    self.root = Node()

  def ls(self, path) -> List[str]:
    node = self._walk(path)
    if node.is_file:
      return [node.name]
    return node.children.keys()

  def mkdir(self, path):
    self._walk(path, create=True)

  def addContentToFile(self, path, content):
    node = self._walk(path, create=True, make_file=True)
    node.content += content

  def readContentFromFile(self, path):
    node = self._walk(path)
    return node.content

  def _walk(
    self, path, create=False, make_file=False
  ) -> Node:
    """
    Traverse to 'path' and return a node.
    - create: create missing directories if needed.
    - make_file: mark the final node as a file.
    """
    if path == "/":
      return self.root

    parts = [p for p in path.split("/") if p]
    cur = self.root
    for name in parts:
      if name not in cur.children:
        if not create:
          raise
        cur.children[name] = Node(name=name, is_file=False)

      cur = cur.children[name]

    if make_file:
      cur.is_file = True

    return cur
```

6.7 Quad Tree

Given a $n*n$ matrix grid of 0's and 1's only. We want to represent grid with a Quad-Tree.

```python
@datalcass
class Node:
  val: bool
  is_leaf: bool
  top_left: Node | None
  top_right: Node | None
  bottom_left: Node | None
  bottom_right: Node | None
```

If the current grid has the same value (i.e all 1's or all 0's) set `is_leaf` True and set `val` to the value of the grid and set the four children to Null and stop.

Core Clues:

1. Recursivly constrct $\Rightarrow$ need to partion the matrix into 4 parts

$$(r,c)$$
$$(r,c+\frac{l}{2})$$
$$(r+\frac{l}{2},c)$$
$$(r+\frac{l}{2},c+\frac{l}{2})$$

2. Adapt to the terminal condition

```python
def construct(self, grid):
  n = len(grid)

  def is_uniform(r, c, l):
    first = grid[r][c]
    for r in range(r, r + l):
      for c in range(c, c + l):
        if grid[r][c] != first:
          return False, None
    return True, first

  def build(r, c, l):
    is_uni, val = is_uniform(r, c, l)
    if is_uni:
```

```
      return Node(bool(val), True, None, None, None, None)

    sub_l = l // 2
    tl = build(r, c, sub_l)
    tr = build(r, c + sub_l, sub_l)
    bl = build(r + sub_l, c, sub_l)
    br = build(r + sub_l, c + sub_l, sub_l)
    # Any value for val
    return Node(True, False, tl, tr, bl, br)

  return build(0, 0, n)
```

Chapter 7
Sort

7.1 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.

7.2 Algorithms

7.2.1 Binary / 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)
```

7.2.1.1 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
```

Alternatively, 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.

7.2.1.2 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
```

7.2.1.3 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
```

7.2.2 Merge Sort

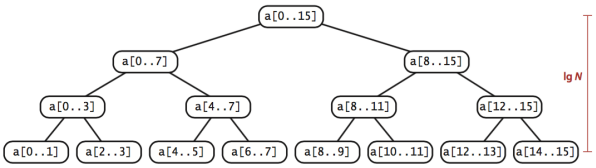


Fig. 7.1: Merge Sort

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 recursion. 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 $O(n^2)$.

7.2.3 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.

7.2.4 Bridge Two Sorted Arrays

We want to bridge (not merge) two sorted arrays by remove a minimal number of elements from two arrays.

$$A[:i] + B[j:]$$

For example:

```
A = [1, 2, 3, 10]
B = [1, 8, 9]

ret = [1, 2, 3, 8, 9]
```

Core Clues:

1. Search: We need to search the bridging indices on two arrays
2. Brute force: iterate both arrays, $O(N^2)$
3. Binary search: iterate A while biscet B, using `bisect.bisect_right`, $O(N \lg N)$
4. Two pointers: since both A and B are sorted, moving the index i in A forward, the search of j in B can only be forward. j cannot go backward.

```
def bridge(A, B):
    # A[:i]
    # B[j:]
    ret = 0
    j = 0
    for i in range(len(A)+1):
        while i-1 >= 0 and j < len(B) and A[i-1] > B[j]:
            j += 1

        ret = max(ret, i + (len(B)-j))

    return ret
```

7.3 PROPERTIES

7.3.1 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.

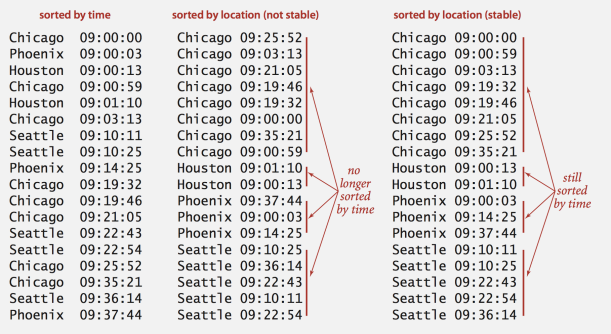


Fig. 7.2: Stale sort vs. unstable sort

7.3.2 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

7.3.3 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 $O(N)$.

7.3.4 Sorting Summary

See Figure 7.3.

7.4 PARTIAL QUICKSORT

7.4.1 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: $O(n \log k)$.

The 2nd version is much faster and more memory efficient.

- To find k-th *smallest*, maintain a *max*-heap of top k smallest number, with the top of the heap being the current k-th smallest.
- To find k-th *largest*, maintain a *min*-heap of tok k largest number, with the top of the heap being the current k-th largest.

In python built-in there are:

```
heapq.nlargest(n, iterable, key=None)
heapq.nsmallest(n, iterable, key=None)
```

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

The average time complexity is

	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. 7.3: Sort summary

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

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

7.4.2 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):
    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[p] = A[p], A[l]
    return l  # return 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)
    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 - 7.2.1.1.

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
```

7.4.3 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:
  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)
```

7.5 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.

7.5.1 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`

```
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
```

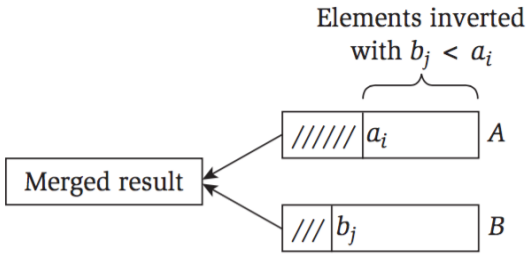


Fig. 7.4: Merge and Count

```
      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
```

Chapter 8
Search

8.1 Binary Search

Variants: Find the insertion point

1. `bisect_left` leftmost element to insert the target
2. `bisect_right` rightmost element to insert the target
3. Get the idx equal OR just lower (floor)
4. Get the idx equal OR just higher (ceil)

The above four have subtle differences.

```
target = 5
lst = [1, 3, 5, 7, 9]
idx = [0, 1, 2, 3, 4]
             ^  ^
             |  |
   bisect_left  bisect_right
```

8.1.1 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.

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

    return lo
```

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

    return lo
```

8.1.3 Generalized bisect

Find the smallest bound that satisfies some condition of *predicate*:

```
def bisect_left(A, predicate, lo, hi):
    while lo < hi:
        mid = (lo+hi) // 2
        if predicate(A[mid]):  # go right
            lo = mid + 1
        else:  # left
            hi = mid

    return lo
```

8.1.4 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 bi_search(self, A, t, lo, hi):
    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]
```

8.1.5 idx equal OR just higher

$A_{idx} \geq target$.

```
def bi_search(self, A, t, lo, hi):
    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
```

8.1.6 bisect

8.1.6.1 Built-in Library

Assuming A is already sorted.

1. `bisect.bisect_left(A, x)`: If x is already present in A, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use to `list.insert()`.
2. `bisect.bisect_left(A, x)`: Similar to `bisect_left()`, but returns an insertion point which comes after (to the right of) any existing entries of x in a. The return value is suitable for use to `list.insert()`.
3. `lst.insert(i, x)`: Insert x at position i, the existing A_i is push towards the end
4. `bisect_left` only looks at `__lt__`
5. `bisect_left` can have `key`: `bisect_left(A, x, lambda a: B[a])`

Find bounds. Given a sorted list A and a target q, find the largest element less than or equal to q and the smallest element greater than or equal to q. If no such element exists, return -1 for that bound.

```
def find_bounds(A, q):
    idx = bisect.bisect_left(A, q)
    if idx < len(A) and A[idx] == q:
        lo = A[idx]
    else:
        lo = A[idx-1] if idx > 0 else -1

    idx = bisect.bisect_right(A, q)
    if idx > 0 and A[idx-1] == q:
        hi = A[idx-1]
    else:
        hi = A[idx] if idx < len(A) else -1

    return lo, hi
```

8.2 Applications

8.2.1 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
```

Random Point in Area. You are given an array of non-overlapping axis-aligned rectangles rects where $rects_i = [a_i, b_i, x_i, y_i]$ indicates the bottom-left corner and the top-right corner point. Design an algorithm to pick a random integer point inside the space covered by one of the given rectangles, including edges.

1. Probablistic select a point in the area space
2. Need to identify which rectangle for the target area $\Rightarrow$ prefix sum of the area
3. Find the first prefix area sum larger than target area $\Rightarrow$ `bisect`
4. Boundary problem: `pref = [3, 7, 12]`, when $k = 0$, we assign to the 1st rectangle, when $k = 3$, we assign to the 2nd rectangle $\Rightarrow$ `bisect_right`
5. Assign a point of the target rectangle for the target area

```
class Solution:
  def __init__(self, rects):
    self.rects = rects

    self.pref = []  # prefix sums of area
    subtotal = 0
    for x1, y1, x2, y2 in rects:
      subtotal += (x2 - x1 + 1) * (y2 - y1 + 1)
      self.pref.append(subtotal)

    self.total = subtotal

  def pick(self):
    k = random.randrange(self.total)
    # not bisect_left
    i = bisect.bisect_right(self.pref, k)

    base = self.pref[i-1] if i - 1 >= 0 else 0
    offset = k - base

    x1, y1, x2, y2 = self.rects[i]
    w = (x2 - x1 + 1)
    x = x1 + (offset % w)
    y = y1 + (offset // w)
    return [x, y]
```

8.3 COMBINATIONS

8.3.1 Extreme-value problems

Longest increasing subsequence (LIS). Array A.

Clues:

1. `L`: The *min* index *last/tail* 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. `Binary search`: For each currently scanning index `i`, if it smaller (i.e. $\neg$ increasing), to maintain the `L`, binary search to find the position to update the min value. The `bi_search` need to find the element $\geq$ to `A[i]`.

```
def LIS(self, A):
    n = len(A)
    L = [-1 for _ in range(n+1)]
    k = 1
    L[k] = A[0]  # store value
    for v in A[1:]:
        j = bisect.bisect_left(L, v, 1, k+1)
        L[j] = v
        k += 1 if j == k+1 else 0

    return k
```

The bisect index range can be avoided by building `L` dynamically. Let L_i be the smallest index that a LIS of length $i+1$ ending at that index.

```
def LIS(self, A):
    L = []
    for i in range(len(A)):
        j = bisect.bisect_left(
            L, A[i], key=lambda e: A[e]
        )
        if j < len(L):
            L[j] = i
        else:
            L.append(i)

    return len(L)
```

If need to return the LIS itself, we need to maintain a predecessor array π.

```
def LIS(self, A):
    n = len(A)
    L = []
    pi = [-1] * n
    for i in range(n):
        j = bisect.bisect_left(
            L, A[i], key=lambda e: A[e]
        )
        if j < len(L):
            L[j] = i
        else:
            L.append(i)

        pi[i] = L[j-1] if j-1 >= 0 else -1

    # build the LIS
    ret = []
    cur = L[-1]
    while cur != -1:
        ret.append(A[cur])
        cur = pi[cur]

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

Note that monotonic queue is not applicable here. Monotonic queue is used for sliding-window problems. The LIS problem is fundamentally different because it deals with a subsequence that is not necessarily contiguous, rather than a sliding window that is contiguous.

8.4 HIGH DIMENSIONAL SEARCH

8.4.1 2D Search

2D search matrix I. Search a target in $m \times n$ mat.

The mat as the following properties:

1. Integers in each *row* are sorted from left to right.
2. 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. Search a target in $m \times n$ mat.

The mat as the following properties:

1. Integers in each *row* are sorted from left to right.
2. 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:

$$O(m \log n)$$

More formally

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

8.4.2 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.

Chapter 9
Array

9.1 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.

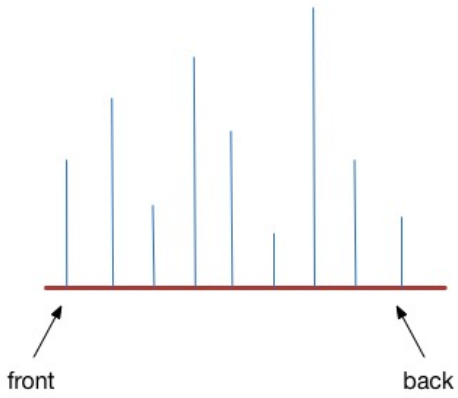


Fig. 9.1: Container with Most Water

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

1. **Extreme cases**: start from the extreme cases $\Rightarrow$ be greedy.
2. **Two pointers**: Two pointers *lo*, *hi* at two ends, calculate the current area.
3. **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.

Minimum Window Substring. Given two strings s and t respectively, return the minimum window substring of s such that every character in t (including duplicates) is included in the window.

```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
```

Core Clues:

1. Window $\Rightarrow$ sliding window, two pointers
2. Check whether cover t with duplicates $\Rightarrow$ dictionary
3. Whether full match $\Rightarrow$ counter of missing or matched

```
def minWindow(s, t):
  N = len(s)
  requires = defaultdict(int)
  for ch in t:
    requires[ch] += 1

  missing = len(t)
  lo = 0
  start = 0
  end = len(s)
  for i in range(N):
    ch = s[i]
    if ch in requires:
      if requires[ch] > 0:
        missing -= 1
      requires[ch] -= 1

    # when full matched
    if missing == 0:
      hi = i + 1
      while lo < hi:
        if s[lo] in requires:
          if requires[s[lo]] < 0:
            requires[s[lo]] += 1
          else:
            break

        lo += 1

      # update best window
      if hi - lo < end - start:
        start, end = lo, hi

      # since found, move left to search for next valid window
      if s[lo] in requires:
        requires[s[lo]] += 1
        missing += 1
      lo += 1

  return s[start:end]
```

9.1.1 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 rearrange(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]

        pos_expt = not pos_expt

        # handle same-idx swap
        if i == neg:
            neg += 1
        if i == pos:
            pos += 1
```

9.2 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.

9.2.1 Circular max sum

Linear (acyclic) problem can be solved linear with DP algorithm for maximum subarray sum - Section 19.2.

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 $A[:i]$ with subarray starting at 0 (*forward* starting from the left side), up to A_{i-1}.
   Max subarray sum from index -1. Construct right max sum R_i for max sum over the indexes $A[i:]$, with subarray ending at -1 (*backward* starting from the right side), up to A_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**:

$$maxa = \max(R_i + L_i, \forall i)$$

9.2.2 Non-adjacent cell

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

Relax to linear problem, non-circular array. We need to consider the decision on A_i.

1. Take the A_i
2. Not take the A_i

The transition funciton is:

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

Linear 19.2.

To solve it in circular array, either

- Include A_0 and exlcude A_{-1} or
- Include A_{-1} and exclude A_0.

```
def rob(self, A):
    n = len(A)
    if n <= 1:
        return sum(A)

    # ignore A[~0]
    F = [0] * (n+2)  # two dummy states F_0, F_1
    for i in range(2, 2 + n - 1):
        F[i] = max(F[i-1], F[i-2] + A[i-2])

    ret = F[~1]  # not F[~0]

    # ignore A[0]
    F = [0] * (n+2)
    for i in range(2 + 1, 2 + n):
        F[i] =  max(F[i-1], F[i-2] + A[i-2])
```

```
    ret = max(ret, F[~0])
    return ret
```

9.2.3 Binary search

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

```
A = [1, 2, 3, 4, 5, 6, 7]
B = [4, 5, 6, 7, 1, 2, 3]
```

1. If $A_0 < A_{mid}$, then all values in the first half of the array are sorted.
2. If $A_{mid} < A_{\sim 0}$, 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**.

9.3 Local Search - Monotonic Stack

9.3.1 Next/Prev Smaller Value

Previous Smaller Element. Left / previous 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.

Core clues:

1. Nearest $\equiv$ spatial locality.
2. **Search Backward**: for each position in a sequence of numbers, search among the *previous* positions for the last position that contains a smaller value.
3. **Invariant**: Maintain a *motonotically increasing* stack.
4. If the question asks for all nearest *larger* values, maintain a *motonotically decreasing* stack.

```
def prev_smaller_item(self, A):
    N = len(A)
    L = [-1 for _ in A]
    stk = []
    for i in range(N):
        while stk and A[stk[-1]] >= A[i]:
            stk.pop()

        if stk:
            L[i] = stk[-1]

        stk.append(i)  # store the idx

    return L
```

Next smaller Element - Mirrored Problem. For each item in the array A, find the next cloest smaller item.

Core Clues:

1. Search forward: inituitively when we scan the current index i, we want to find the next cloeset index j that satifies the condition. However, we cannot store the candidates j's in some data structure to reduce time complexity from $O(N^2)$ of brute force.
2. **Match backward**: hold previous items in some data structure, and efficient match the previous element when $A_{prev} > A_{cur}$.
3. Monotonic stack: use a monotonic stack to remember the previous items that are pending to find a smaller element in the future. It is a monotonic increasing stack from left to right in this case.
4. Resolve the previous pending items: when current index i has a smaller value than the previous items, we resolve the previous items as we have found the next closest smaller item as i for them, by popping the stack.
5. Alternatively, we can scan from right to left to reduce to previous smaller element.

The monotonic stack holds pending items that have not found the next smaller element, untill the monotonicity is broken by the current scanning element A_i. We pop the previous pending items and resolve their next smaller element as A_i.

```
def next_smaller_item(A):
    N = len(A)
    # next item on the right
    R = [-1 for _ in range(N)]
    # store the previous, mono asc stk
    stk = []

    for i in range(N):
        while stk and A[stk[-1]] >= A[i]:
            prev = stk.pop()
            R[prev] = i  # found
        stk.append(i)

    return R
```

Largest Mountain. Find the largest mountain area in the histogram. Given N non-negative integers H representing the histogram's bar height where the width of each bar is 1, find the area of largest mountain.

We are to remove some bricks from the bar to form a mountain-shaped arrangement. In this arrangement, the tower heights are non-decreasing, reaching a maximum peak value with one or multiple consecutive towers and then non-increasing. For example, $H = [6,5,3,9,2,7]$ bececomes $[3,3,3,9,2,2]$ with the largest area of 22.

Core Clues:

1. DP: let L_i be the largest asc slide area for `H[:i]`, peaked at H_{i-1}, scanning from left to right. Let R_i

be the largest desc slide area for `H[i:]`, peaked at H_i, scanning from right to left.

2. Previous smaller element: When processing H_i, define lo as the previous smaller element for H_i on the left

$$L_{i+1} = + \begin{cases} L_{prev+1} & \text{for range } [0, prev] \\ (i - prev) * H_i & \text{for range } (prev, i] \end{cases}$$

Substitute i with $i-1$,

$$L_i = + \begin{cases} L_{prev+1} & \text{for range } [0, prev] \\ (i - 1 - prev) * H_{i-1} & \text{for range } (prev, i-1] \end{cases}$$

3. Mono stack: use monotononically increasing stack to find the previous smaller item on the left.
4. Reverse then reverse back: for R_i, we can reuse the same logic, but since it is from right to left, we need to revere H to get R, and then reverse R back.

```
def largest_mountain_area(self, H):
    N = len(H)
    L = self.dp(H)
    R = self.dp(H[::-1])[::-1] # reverse then reverse back

    maxa = 0
    for i in range(N+1):
        # H[:i] & H[i:]
        cur = L[i] + R[i]
        maxa = max(maxa, cur)

    return maxa

def dp(self, H):
    N = len(H)
    L = [0 for _ in range(N+1)]
    stk = []  # mono asc
    for i in range(1, N+1):
        idx = i-1
        while stk and H[stk[-1]] > H[idx]:
            stk.pop()

        prev = stk[-1] if stk else -1
        # [0, prev] and (prev, idx]
        L[i] = L[prev+1] + (idx-prev) * H[idx]
        stk.append(idx)

    return L
```

9.3.2 Local Minimum/Maximum

- To find the boundaries for **local max** of A_{mid} as (lo, i):

$$A_{lo} > A_{mid} < A_i$$

  $\Leftarrow$ maintain a monotonically **decreasing** stack.
- To find the boundaries for **local min** of A_{mid} as (lo, i):

$$A_{lo} < A_{mid} > A_i$$

  $\Leftarrow$ maintain a monotonically **increasing** stack.
- The boundary is (lo, i): $(lo, mid], [mid, i)$.

Minimize Cost of Merging Subarray. We have an array A of positive integers. We must repeatedly merge two adjacent subarray into one, starting from 1-element subarray. The cost of merging two subarrays is the product of their largest elements. Determine the total cost to merge everything into one subarray.

Core Clues:

1. Cost: To remove any number a, it costs $a * b$, where $b \geq a$. So a has to be removed by a bigger neighbor b. To minimize this cost, we need to minimize b.
2. Greedy: Max element is used at each subarray $\Rightarrow$ put big leaf nodes closer to the top/root $\Rightarrow$ greedily start with the smallest leaf.
3. Everytime the smaller element is merged, it is gone. Brute:

```
i = A.index(min(A))
cost += A[i] * min(A[i-1], A[i+1])
A.pop(i)
```

   This is $O(N^2)$.
4. Merging is a local event. Find local min: we need to find i s.t.

$$A_{lo} > A_{mid} < A_i$$

   $\Leftarrow$ keep a monotonic decreasing stack for $A_{0:i}$, till the monotonicity is broken by A_i, then we need to pop the stack and maintain the monotonicity.
5. A_{mid} is merged. The mono stack is holding the unmerged items on the left side.

```
def min_cost_merging_leaf_values(A):
    cost = 0
    stk = []
    for a in A:
        while stk and stk[-1] <= a:
            mid = stk.pop()
            # merging mid
            if stk:
                left = stk[-1]
                cost += mid * min(left, a)
            else:
                cost += mid * a

        stk.append(a)

    # can append sys.maxsize as sentiel to flush
    while len(stk) > 1:
        mid = stk.pop()
        left = stk[-1]
        cost += mid * left

    return cost
```

Sum of All Subarray Ranges. A subarray range is defined as max − min. Find the sum of subarray ranges for all subarrays.

Core Clues:

1. To get the range: get the min and max. To get all subarrays: iterate i and j, keep tracks of min and max. Thus $O(N^2)$.
2. We transform the range sums:

$$\begin{aligned}&\sum range\\ &= \sum(\max - \min)\\ &= \sum \max - \sum \min\end{aligned}$$

3. To get $\sum$max: #times as local min $\times$ val.
4. To get the **bounds** for local min:

$$A_{lo} < A_{mid} > A_i$$

   $\Leftarrow$ use monotonically increasing stack.
5. The number of subarrays centered at A_{mid}: $|(lo, mid]| \times |[mid, i)|$.
6. Similarly, find the **bounds** for local max:

$$A_{lo} > A_{mid} < A_i$$

```
def subarray_ranges(self, A):
    N = len(A)
    ret = 0

    sum_max = 0
    stk = []  # mono desc stk for local max
    A.append(sys.maxsize)
    for i in range(N+1):
        while stk and A[stk[-1]] < A[i]:
            mid = stk.pop()
            lo = stk[-1] if stk else -1
            # times: [mid, i), (lo, mid]
            cnt = (i - mid) * (mid - lo)
            sum_max += cnt * A[mid]
        stk.append(i)
    A.pop()

    sum_min = 0
    stk = []  # mono asc stk for local min
    A.append(-sys.maxsize-1)
    for i in range(N+1):
        while stk and A[stk[-1]] > A[i]:
            mid = stk.pop()
            lo = stk[-1] if stk else -1
            # times: [mid, i), (lo, mid]
            cnt = (i-mid) * (mid - lo)
            sum_min += cnt * A[mid]
        stk.append(i)
    A.pop()

    return sum_max - sum_min
```

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

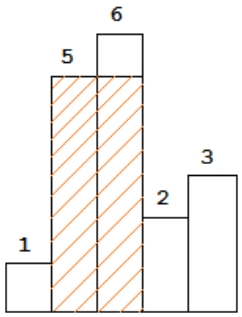


Fig. 9.2: Largest rectangle in histogram
$i \rightarrow$ height 2, $cur \rightarrow$ height 6, 5, 1

Core clues:

1. Largest rectangle: when scanning index i, calculate the rectangle area with a height h s.t. $h > H_i$. We want to find the leargest rectangle in `H[lo:i]`.
2. **Local min**: the rectangle area is calcualted as local min height $\times$ its range. To find the boundaries for local min of A_{mid} as (lo, i). H and A are interchangeable:

$$A_{lo} < A_{mid} > A_i$$

3. Monotonic stack: maintain the monotonic increasing (non-decreasing) stack, with increasing indices with scanning i.
4. Post-processing in the end

Code:

```
def largest_rectangle_area(self, H):
    N = len(H)
    maxa = -sys.maxsize-1
    stk = []  # store the idx, mono asc stack

    H.append(-sys.maxsize-1)  # sentiel to flush
    for i in range(N+1):
        while stk and H[stk[-1]] > H[i]:
            mid = stk.pop()
            # calculate area when popping
            # bound: (lo, i), [lo+1, i)
            lo = stk[-1] if stk else -1
            l = i - (lo+1)
            area = H[mid]*l
            maxa = max(maxa, area)

        stk.append(i)
    H.pop()  # clear

    return maxa
```

Maintain a monotonic stack to store 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.

- Height: on the left side of h, all the higher heights than h are popped thus absent; while on the right side of h, all have higher heights until H_i.
- Width: 1) what are **not in** the stack have higher heights than h; and 2) the earlier elements **in** the stack have smaller heights than h, $\Rightarrow$ the lower bound is $lo = stk_{-1} + 1$.

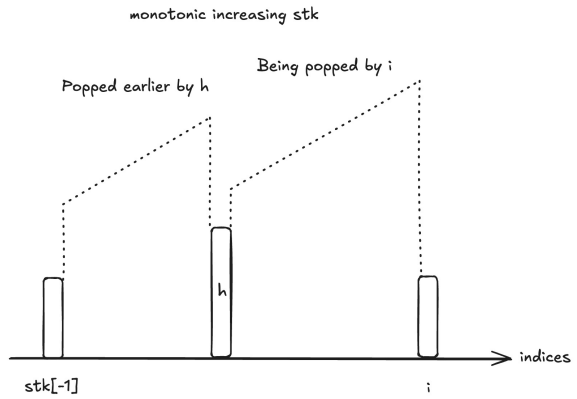


Fig. 9.3: Monotonic stack before and after h

9.3.3 Monotonic Greedy Search

Lexically Smallest Distinct-Element Subsequence. Given a string s, return the lexicographically smallest subsequence of s that contains all the distinct characters of s exactly once. For example, `s = "cbacdcbc"`, return `"acdb"`. `s = "cba"`, return `"cba"`

Core Clues:

1. Greedy goal: smallest $\Rightarrow$ always place smaller chars earlier if possible to get the lexicographically smallest result.
2. Contains all the distinct chars of s exactly once $\Rightarrow$ `visited`
3. Monotonic stack: maintain a stack where chars are as lexicographically small as possible in stack by popping bigger chars if they can reappear later.
4. Last occurrence check: to ensure the result contain all distinct chars $\Rightarrow$ before popping a char from the stack, ensure it appears again later.

```
def smallest_subsequence(s):
    stk = []
    last_occur = defaultdict(int)
    for i, c in enumerate(s):
        last_occur[c] = i
    visited = defaultdict(bool)
    for i, c in enumerate(s):
        if not visited[c]:
            while stk and stk[~0] > c \
                and last_occur[stk[~0]] > i:
                e = stk.pop()
                visited[e] = False

            stk.append(c)
            visited[c] = True

    return "".join(stk)
```

Longest Positive Subarray Sum. Given a list A of numbers, find the longest positive subarray sum.

Core Clues:

1. Subarray sum $\Rightarrow$ prefix sum.
2. Longest positive subarray sum $\not\Rightarrow$ for A_i, find the previous smaller prefix sum
3. Longest positive subarray sum $\Rightarrow$
   - Smaller previous index $\Rightarrow$ more potential to be longest
   - Smaller prefix sum $\Rightarrow$ more potential to be positive

   For A_i, the larger previous index is a candidate only because it is having a smaller prefix sum, otherwise there is a smaller index before it for a better potential for find the longest.
4. Monotonic stack: maintain a value monotonically decreasing stack with increasing indices. The larger indices are put into stack only because it is having a smaller prefix sum.
5. Greedy longest length: greedily scan $j = i$ from end, and pop the stack until the smallest j.

Correctness of Greedy $\Leftarrow$ strictly better:

1. Scanning form the left, for the same j, if $i < i'$, we still need to check i' if its prefix sum is smaller because i' is a valid candidate for the longest positive subarray sum.
2. Scanning from the right, for the same j, if $i > i'$, there is no need to check (j, i') since we have already checked (j, i) which is longer. We have to check a smaller j' for i' to find the longest candidate.

```
def longest_positive_subarray_sum(A):
    N = len(A)
    # prefix sum for A[:i]
    P = [0 for _ in range(N+1)]
    for i in range(1, N+1):
        P[i] = P[i-1] + A[i-1]

    stk = []  # mono desc stack
    for i in range(N+1):
        if not stk or P[stk[~0]] > P[i]:
            stk.append(i)  # greedy
```

```
    ret = 0
    for i in range(N, 0, -1):
        while stk and P[i] - P[stk[~0]] > 0:
            j = stk.pop()
            ret = max(ret, i - j)

    return ret
```

9.4 Sliding Window - Monotonic Queue

9.4.1 Sliding Window

Typically applied in sliding window.

Sliding Window Maximum. Given an array A, 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. $\Rightarrow$ mono queue.

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

We want to keep track of the max in sliding window `A[i:j+1]` in $O(1)$. Consider a queue q to only store max element at q_0. When expanding the window by j, if **trailing** A_j is **the largest**, we put it at q_0, if it is **second largest**, we put it q_1. Similarly for the **third largest** at q_2. When shrinking the window by i, if q_0 fall out of the window, we need to promote q_1 as the largest.

If processing a current element A_j to the queue, and the back of the queue has some element smaller than or equal to A_j, that smaller element will never be the maximum again once A_j is in the window. Because **trailing** A_j **dominates** it (strictly better), and they both sit in the queue.

Now the queue is always in strictly decreasing order from front to back. The front is the largest element. ■

Count Number of Subarrays. Count number of continuous subarrays in A s.t. within the subarray $|A_i - A_j| \leq 2, \forall i, j$.

Core clues:

1. Need to maintain max and min in the sliding window $\Rightarrow$ Monotonic Queue
2. `q_min` is monotonically increasing
3. `q_max` is monotonically decreasing
4. `q_min` and `q_max` are storing the indices.

```
def continuous_subarrays(self, A):
    q_max = deque()  # mono desc
    q_min = deque()  # mono asc
    i = 0
    j = 0
    ret = 0
    while j < len(A):
        # process A[j]
        while q_max and A[q_max[~0]] <= A[j]:
            q_max.pop()
        q_max.append(j)
        while q_min and A[q_min[~0]] >= A[j]:
            q_min.pop()
        q_min.append(j)

        while q_max and q_min \
            and A[q_max[0]] - A[q_min[0]] > 2:
            # shrink to pass the breaking idx
            prev = q_max.popleft() \
                if q_max[0] < q_min[0] \
                else q_min.popleft()
            i = prev + 1

        ret += j - i + 1
        j += 1

    return ret
```

Longest Subarray With Absolute Diff Less Than or Equal to Limit . Given an array of integers A, and an integer *limit*, return the size of the longest non-empty subarray such that the absolute difference between any two elements of this subarray is less than or equal to limit.

Core Clues:

1. Keep tracks the max and min the sliding window $\Rightarrow$ monotonic queue

```
def longestSubarray(self, A, limit) -> int:
  # q_max mono decreasing, front is window max
  # q_min mono increasing, front is window min
  q_max, q_min = deque(), deque()
  ret = 0
  i = 0

  for j in range(len(A)):
    while q_max and A[q_max[~0]] < A[j]:
      q_max.pop()
    q_max.append(j)

    while q_min and A[q_min[~0]] > A[j]:
      q_min.pop()
    q_min.append(j)

    while q_max and q_min \
        and A[q_max[0]] - A[q_min[0]] > limit:
      # move lo up and evict stale indices
      if q_max[0] == i:
        q_max.popleft()
      if q_min[0] == i:
        q_min.popleft()
      i += 1

    ret = max(ret, j - i + 1)

  return ret
```

9.5 MATRIX

Diagonal Traverse .

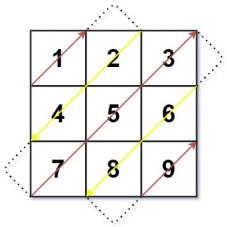


Fig. 9.4: Diagonal Traverse

Core Clues : Matrix by index

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{bmatrix}$$

1. Diagonal $\Rightarrow r+c=\kappa$ where κ is a constant.
2. Order *Ra* κ determines the order of elements within a diagonal.

```
def findDiagonalOrder(self, matrix):
    R, C = len(matrix), len(matrix[0])
    F = [[] for _ in range((R-1)+(C-1)+1)]
    for r in range(R):
        for c in range(C):
            F[r+c].append(matrix[r][c])

    ret = []
    for i in range(R+C-1):
        if i % 2 == 1:
            ret.extend(F[i])
        else:
            ret.extend(F[i][::-1])

    return ret
```

9.6 VOTING ALGORITHM

9.6.1 Majority Number

9.6.1.1 $\frac{1}{2}$ 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
```

9.6.1.2 $\frac{1}{3}$ 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.

9.6.1.3 $\frac{1}{k}$ 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
```

9.7 INDEX REMAPPING

9.7.1 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.

9.7.2 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:

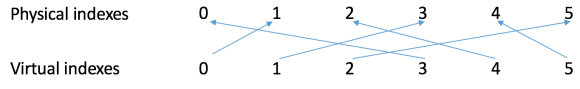


Fig. 9.5: Virtual Indices Remapping

Find the pattern:

$$2 * i = p'$$

```
N = 6
v       p'      p
0       1       1
1       3       3
2       5       5
3       7       0
4       9       2
5       11      4

N = 7
v       p'      p
0       1       1
1       3       3
2       5       5
3       7       0
4       9       2
5       11      4
6       13      6

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 10
String

10.1 Palindrome

10.1.1 Palindrome anagram

- **Test palindrome anagram.** Char counter, number of odd count should ≤ 0.
- **Count palindrome anagram.** See Section-13.1.4.
- **Construct palindrome anagram.** Construct all palindrome anagrams given a string `s`.

Core Clues:

1. Different choices of char $\Rightarrow$ backtracking, choose the next from the char counters of `s`.
2. To avoid loop $\Rightarrow$ jump parent char

```
def backtrack(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.backtrack(s, counters, k, k*n+cur+k*n, ret)
        counters[k] += n*2
```

Jump within the iterations of choice to avoid dead loop.

How to handle odd counter? Check outside the backtrack, and potentially raise error.

10.1.2 Number

Next Permutation. Given a string n representing an int, return the closest int (not including itself), which is a palindrome. For example $n =$ '123', return '121'.

1. Palindrome $\Rightarrow$ Mirror the left half of n.
2. 19997 has two candidates 19991 and 20002 $\Rightarrow$ for the half, $+/-1$ for carry/borrow
3. 1000 has two candidates 999 and 1001 $\Rightarrow$ More/less digits of n $\Rightarrow$ checking numbers $10..0, 9..9$

```
def nearestPalindromic(self, s: str) -> str:
  l = len(s)
  odd_l = l & 1
  if odd_l:
    half = s[:l//2] + s[l//2]
  else:
    half = s[:l//2]

  def mir(half: str) -> str:
    if not odd_l:
      return half + half[::-1]
    else:
      return half + half[::-1][1:]

  # candidates
  cands = {
    mir(str(int(half) - 1)),
    mir(half),
    mir(str(int(half) + 1)),
  }
  cands |= {
    str(10 ** l + 1),
    str(10 ** (l - 1) - 1)
  }
  cands.discard(s)
  return min(
    cands,
    key=lambda e: (abs(int(s) - int(e)), int(e))
  )
```

10.2 Anagram

Group of strings by anagram. Using a frequency vector

```
class Solution:
  def groupAnagrams(self, strs):
    # key: 26-tuple counts; value: list of words
    d = defaultdict(list)
    for s in strs:
      cnt = [0] * 26
      for ch in s:
        cnt[ord(ch) - ord('a')] += 1
      d[tuple(cnt)].append(s)
    return list(d.values())
```

10.3 KMP

Find the pattern p in string s within complexity of $O(|P|+|S|)$.

10.3.1 Prefix matching suffix table

Intuition: when a mismatch happens at p_i vs s_i, don't restart from p_0; instead jump to the next best candidate i right after the matched prefix & suffix, keeping the already verified prefix.

Let L_i be the length of the longest proper prefix that matches a suffix ending at p_i.

1. A proper prefix is a prefix that is not equal to the whole string itself. Proper ensures that $L_i < i+1$.
2. Need to maintain the longest proper prefix of a substring that is also a suffix of it.

i	0	1	2	3	4	5	6
p_i	A	B	C	D	A	B	D
L_i	0	0	0	0	1	2	0

```
def kmp_lps(p):
    L = [0] * len(p)
    i = 1
    pre = 0
    while i < len(p):
        if p[i] == p[pre]:
            L[i] = pre + 1
            pre += 1
            i += 1
        elif pre:  # fallback
            pre = L[pre - 1]
        else:  # no fallback
            L[i] = 0
            i += 1

    return L
```

Time complexity:

- From code itself it appears to be $O(|P|^2)$.
- Define $\Delta = i - pre$.
- i never goes backward - i either forwards of stays.
- Any time i doesn't move forward, Δ must rise
- j is bounded by $O(|P|)$ and Δ is bounded by $O(|P|)$

10.3.2 Searching algorithm

```
def kmp_search(p, s):
```

```
              1         2
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:     ABCDABD
i:     0123456

              1         2
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
W:         ABCDABD
i:         0123456
```

Fig. 10.1: KMP example

```
    L = kmp_lps(p)
    ret = []
    i = 0
    j = 0
    while j < len(s):
        if p[i] == s[j]:
            i += 1
            j += 1
            if i == len(p):
                ret.append(j - i)
                i = L[i - 1]
        elif i:
            i = L[i - 1]
        else:
            j += 1

    return ret
```

Time compelxity:

- Define $\Delta = j - i$.
- j never goes backward
- Any time j doesn't move forward, Δ must rise
- j is bounded by $O(|S|)$ and Δ is bounded by $O(|P|+|S|)$

10.3.3 Applications

1. Find needle in haystack.
2. Shortest palindrome

10.3.4 Subsequence

Given a string s and an array of strings *words*, return the number of $words_i$ that is a subsequence of s. For example, input: `s = "abcde", words = ["a","bb","acd","ace"]`

Core clues:

1. Test whether one string is another's subsequence is straightforwad, how to test multiple strings is one's subsequence? ⇒ consume the multiple strings in one iteartion.
2. Each character can be a candidate ⇒ char map.
3. Each $words_i$ is only match once ⇒ iterator

```
def numMatchingSubseq(self, S, words) -> int:
  """
  Linear O(|S| + sum(|word|))
  no need to if-check with HashMap + Iterator
  """
  itr_lsts = defaultdict(list)
  for w in words:
    itr_lsts[w[0]].append(iter(w[1:]))

  for c in S:
    itrs = itr_lsts.pop(c, [])
    for itr in itrs:
      ch = next(itr, None)
      itr_lsts[ch].append(itr)

  return len(itr_lsts[None])
```

Note `itr_lsts` can be short formed as `itrss`

Chapter 11
Math

11.1 Functions

Equals. Requirements for equals

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

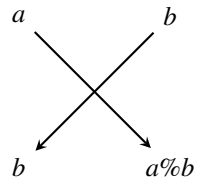
Compare. Requirements for compares (total order):

1. Antisymmetry
2. Transitivity
3. Totality

11.2 Divisor

gcd. Greatest common divisor.

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




```
# recursive abbr
def gcd(a, b):
    if b == 0:
        return a

    return gcd(b, a % b)

# iterative
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 $gcd(a,0) = a$.

11.3 Power

power(x, n). To calculate x^n.

Core Clues :

1. $O(N)$ is trivial, need to do better than it
2. Divide the problem by half

$$x^n = (x^2)^{n/2}$$
$$x^n = x^{n/2} * x^{n/2} * x^n \mod 2$$

```
def pow(self, x, n):
    is_invert = False if n > 0 else True

    n = abs(n)
    ret = 1.0
    while n > 0:
        if n & 1 == 1:
            ret *= x

        n >>= 1
        x *= x

    if is_invert:
        ret = 1.0 / ret

    return ret
```

11.4 PRIME NUMBERS

11.4.1 Sieve of Eratosthenes

11.4.1.1 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.

11.4.1.2 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

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

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)
```

11.4.2 Factorization

Backtracking: Section-17.7.1.1.

11.5 Median

11.5.1 Basic DualHeap

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

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:
  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"""
```

11.5.2 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
from dataclasses import dataclass

@dataclass
class Value:
    val: int
    deleted: bool


class Heap:
    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:
    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
```

11.6 Modular

11.6.1 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.

11.7 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].

Enumerate to find the pattern:

```
1       10      11      ...     19
2       21      22      ...     29
3       31      32      ...     39
4 ...
.
.
```

Using DFS.

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

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

Optionally, 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 12
Arithmetic

12.1 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.

12.2 Calculator

12.2.1 Parsing

Basic calculator. Given a string s which represents an expression, evaluate this expression and return its value.

```
s = "3+2*2"
s = " 3/2 "
s = " 23+5 / 2 "
```

Core Clues:

1. Parse the string char by char
2. Lookback number and lookback operator
3. Sentiels: `None`.
4. Append `"0"` to flush

```
def calculate(self, s: str) -> int:
    stk = []
    num = None
    op = None

    for c in s + '\0': # sentinel to flush the last number
        if c == ' ':
            continue

        if c.isdigit():
            num = (num if num else 0) * 10 + int(c)
            continue

        # c is operator, assign c to op, process op first
        if op == '+' or op is None:
            stk.append(num)
        elif op == '-':
            stk.append(-num)
        elif op == '*':
            stk[-1] = stk[-1] * num
        elif op == '/':
            # truncate toward zero (not use // for negatives)
            stk[-1] = int(stk[-1] / num)
        else:
            raise

        num = None
        op = c

    return sum(stk)
```

Add brackets ().

1. Use the original function as a recursive function
2. A `"("` is a recurisve call stack, and A `")"` is a return the current recurisve call.

```
def calculate(self, s: str) -> int:
    s += '\0'
    val, _ = self.parse(s, 0)
    return val

def parse(self, s, i: int) -> tuple[int, int]:
    stk = []
    num = None
    op = None

    while i < len(s):
        c = s[i]
        if c == ' ':
            i += 1
            continue

        if c.isdigit():
            num = (num if num else 0) * 10 + int(c)
            i += 1
            continue

        if c == '(':
            # Evaluate bracketed expression
```

```
            subtotal, j = self.parse(s, i + 1)
            num = subtotal
            i = j
            continue

        # at this point c is an operator or ')'
        # assign c to op
        num = num if num else 0
        if op == '+' or op is None:
            stk.append(num)
        elif op == '-':
            stk.append(-num)
        elif op == '*':
            stk[-1] = stk[-1] * num
        elif op == '/':
            # Truncate toward zero
            stk[-1] = int(stk[-1] / num)
        else:
            raise
        num = None

        if c == ')':
            # finish this level, skipping ')'
            return sum(stk), i + 1
        else:
            op = c  # include '\0'
            i += 1

    return sum(stk), i
```

Alternatively, iterative Approach: note that the order inside the for-loop cannot change

```
def calculate(self, s: str) -> int:
    stk = []  # holds signed ints, plus (, the op just before (
    num = None
    op = None

    for c in s + '\0':
        if c == ' ':
            continue

        if c.isdigit():
            num = (num if num else 0) * 10 + int(c)
            continue

        if c == '(':
            # Save the operator before (
            stk.append(op)  # +/-
            stk.append('(')
            op = None # reset
            continue

        # c is +/-/): flush
        if num != None:
            if op == '+' or op is None:
                stk.append(num)
            else:  # '-'
                stk.append(-num)
            num = None

        if c == ')':
            subtotal = 0
            while stk and stk[-1] != '(':
                subtotal += stk.pop()
            stk.pop()  # remove (

            prev_op = stk.pop()  # op before (
            if prev_op == '+' or prev_op is None:
                stk.append(subtotal)
            else:  # '-'
                stk.append(-subtotal)
            # op remains unchanged
        else:  # only for +/-
            op = c

    return sum(stk)
```

Adding `"*/"`

```
def calculate(self, s: str) -> int:
    stk = []
    num = None
    op = None

    for c in s + '\0':
        if c == ' ':
            continue

        if c.isdigit():
            num = (num if num else 0) * 10 + int(c)
            continue

        if c == '(':
            stk.append(op)
            stk.append('(')
            op = None  # reset
            continue

        if num != None:
            if op == '+' or op is None:
                stk.append(num)
            elif op == '-':
                stk.append(-num)
            elif op == '*':
                stk[-1] = stk[-1] * num
            else:  # '/'
                stk[-1] = int(stk[-1] / num)
            num = None

        if c == ')':
            subtotal = 0
            while stk and stk[-1] != '(':
                subtotal += stk.pop()
            stk.pop()  # remove (

            prev_op = stk.pop()
            if prev_op == '+' or prev_op is None:
                stk.append(subtotal)
            elif prev_op == '-':
                stk.append(-subtotal)
            elif prev_op == '*':
                stk[-1] = stk[-1] * subtotal
            else:  # '/'
                stk[-1] = int(stk[-1] / subtotal)
        else:
            op = c

    return sum(stk)
```

12.3 Appendix - 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.

12.3.1 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.

12.3.2 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
```

12.3.3 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
```

```
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
```

12.3.4 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 12.3.1, but pay attention to operands order, which should be reversed when hitting a operator.

Chapter 13
Combinatorics

13.1 Basics

13.1.1 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.

13.1.2 Basic formula

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

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).

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

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 (note: always choose k, not decide whether to choose the k-th).

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

The 1st term is not to choose n-th as the k-th, and the 2nd term is to choose n-th as the k-th

How to construct nCr in code:

```
nCr = [
    [0 for _ in range(K)]  # not default to 1
    for _ in range(N+1)
]
# nCr[0][x] must be 0 where x > 0

nCr[0][0] = 1
for n in range(1, N+1):
    nCr[n][0] = 1
    for r in range(1, K):
        nCr[n][r] = nCr[n-1][r-1] + nCr[n-1][r]
```

Use built-in library:

```
math.comb(n, r)
math.perm(n, r)
math.factorial(n)
```

13.1.3 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]`.

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

13.1.4 N objects, K types

What is the number of permutation of N objects with K different types? To handle duplicates, we need to dedupe:

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

13.1.5 Inclusion–Exclusion Principle

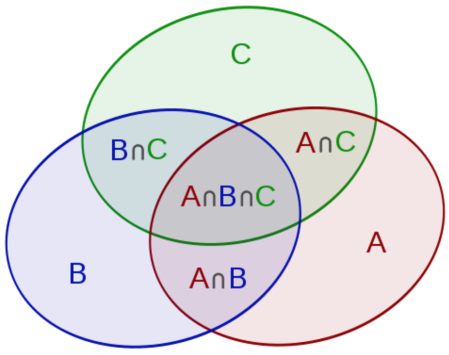


Fig. 13.1: Inclusion–exclusion principle

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

Generally,

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

13.2 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 $S = \{3.A, 4.B, 5.C\}$.

13.2.1 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 $|P_i \cap P_j \cap P_k|$.

13.2.2 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:

$$1+x+x^2+x^3 = (1-x^4)/(1-x)$$
$$1+x+x^2+x^3+x^4 = (1-x^5)/(1-x)$$
$$1+x+x^2+x^3+x^4+x^5 = (1-x^6)/(1-x)$$

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

13.3 PERMUTATION

13.3.1 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))
```

Next Permutation. A permutation of an array of integers is an arrangement of its members into a sequence or linear order. For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

Return the next permutation of a given array.
Core Clues:

1. Obervations:

```
[1, 2, 3, 4]  [1, 2, 4, 3]
[1, 3, 4, 2] : [1, 4, 2, 3]
[1, 4, 2, 3, 3, 2] : [1, 4, 3, 2, 2, 3]
[1, 4, 2, 3, 2, 3] : [1, 4, 2, 3, 3, 2]
```

2. find the suffix shape of \, and the pivot point.

```
[1, 2, 3, 4] : [1, 2, 4, 3]
       ^
[1, 3, 4, 2] : [1, 4, 2, 3]
    ^
[1, 3, 5, 4, 2] : [1, 4, 2, 3, 5]
    ^
```

3. scanning from right to left, find the first larger item than pivot

```
[1, 2, 3, 4] : [1, 2, 4, 3]
       ^  |
[1, 3, 4, 2] : [1, 4, 2, 3]
    ^  |
[1, 3, 5, 4, 2] : [1, 4, 2, 3, 5]
    ^     |
```

4. Swap the rightmost first larger item and make the suffix shape of /

```
[1, 2, 3, 4] -> [1, 2, 4, 3] : [1, 4, 2, 3]
       ^
[1, 3, 4, 2] -> [1, 4, 3, 2] : [1, 4, 2, 3]
    ^  |
[1, 3, 5, 4, 2] -> [1, 4, 5, 3, 2] : [1, 4, 2, 3, 5]
    ^     |
```

```
def nextPermutation(self, A):
  N = len(A)
  # 1) find pivot, find the suffix shape of \
  i = N-2
  while i >= 0 and not A[i] < A[i + 1]:
    i -= 1

  if i < 0:
    # already the highest permutation
    A.reverse()
    return

  # 2) find rightmost element greater than pivot
  j = N-1
  while not A[j] > A[i]:
    j -= 1

  # 3) swap
  A[i], A[j] = A[j], A[i]

  # 4) make the suffix shape of /, reverse
  l, r = i+1, N-1
  while l < r:
    A[l], A[r] = A[r], A[l]
    l += 1
    r -= 1
```

13.3.2 Numbers counting

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

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 $9 \times 9 \times 8 \times ... \times (10-i)$

13.4 Catalan Number

13.4.1 Math

Definition.

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

Proof. Proof of 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.

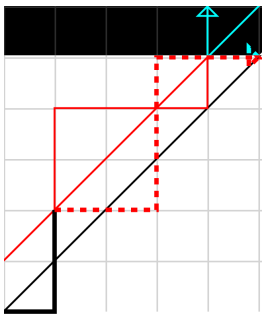


Fig. 13.2: Monotonic Paths

- 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:

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

13.4.2 Applications

The paths in Figure 13.2 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.

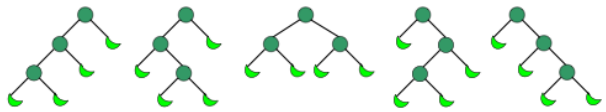


Fig. 13.3: #Full bindary trees. Circles are operators; crescents are operands.

13.5 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 $\{{n \atop k}\}$.

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

Chapter 14
Probability

14.1 SHUFFLE

Equal probability shuffle algorithm.

14.1.1 Incorrect naive solution

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

```
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

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

14.1.2 Uniform Shuffle - Knuth Shuffle

$$|..closed..|i.....|$$

Knuth 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.

```
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.

14.1.3 Random Maximum

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

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 probility of $\frac{k}{k+1}$ to stay. Then the `previous` *max* has the new updated probability of being selected as:

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

```
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
```

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
```

14.2 Sampling

14.2.1 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:

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

14.3 Distribution

14.3.1 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.

14.3.2 Binomial Distr

Notations:

$$B(n, p)$$

pmf:

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

14.4 Expected Value

14.4.1 Dice value

Expected value of rolling dice until getting a 3

14.4.2 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:

$$p_1 = \frac{n}{n}$$
$$p_2 = \frac{n-1}{n}$$
$$p_i = \frac{n-i+1}{n}$$

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 15
Bit Manipulation

15.1 Concepts

15.1.1 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  # 0111,1...
MIN = 0x80000000  # 1000,0...
MSK = 0xFFFFFFFF  # 1111,1...
# hex is 4 bit
```

15.1.2 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.

Shift. Shift operator $<<$ and $>>$ has lower precedence than arithmetic operators.

2's complement

$$-x = \sim x + 1$$

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 sequence $\Rightarrow$

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

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
```

$$\sim x = -x - 1$$

To determine whether a string is palindrome:

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

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

$$x \;\&\; (x-1) == 0$$

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

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

$$x = \texttt{0bhijk1000}$$
$$\sim x = \texttt{0bHIJK0111}$$
$$-x = \texttt{0bHIJK1000}$$

where `HIJK` represent bits that are the negation of `hijk`. Thus,

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

The LSB above is left extended with 0's. To left extended with 1's.

$$lsb = \texttt{0b0001,0000}$$
$$lsb' = \texttt{0b1111,0000}$$
$$lsb - 1 = \texttt{0b0000,1111}$$

Thus,

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

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

```
# msb_idx
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
```

15.1.3 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.

15.2 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`

15.3 Circuit

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

15.3.1 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
```

15.3.2 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
```

15.3.3 Multipler

15.4 Single Number

15.4.1 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:

0000
0001
0010
...
1111

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

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

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

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

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

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

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

15.4.2 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.

15.5 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 16
Greedy

16.1 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.

16.1.1 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

16.2 Extreme First

Schedule jobs with duration and deadlines. Given a list of jobs A and each A_i has a (duration, deadline). Determine whether we can finish all the jobs.

Greedily choose the earliest deadline. Why earliest-deadline-first is the only schedule we need to test?

1. Deadline inversions are harmless to fix. Call an **inversion** two consecutive jobs A_i, A_j where the first one's A_i deadline is later than the second one's A_j . **Swap** such a pair.
2. A_j with the earlier deadline now finishes **sooner**, so it meets its deadline.
3. A_i **inherits** A_j's old completion time ($A_i.t + A_j.t = A_j.t + A_i.t$) and A_i has a later deadline; thus A_i can finish.
4. $\Rightarrow$ Result: a swap never turns a feasible schedule into an infeasible one.

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)
```

16.2.1 Coverage

Minimum to cover. Given n representing the garden starts at the point 0 and ends at the point n. An array *ranges* of length $n+1$ where $ranges_i$ means the i-th tap can water the area $[i-ranges_i, i+ranges_i]$

Find the minimum number of taps that should be open to water the whole garden.

Core clues:

1. Each ranges cover $[start, end] \Rightarrow$ need a map either by start or end.
2. Minimum taps to cover is maximum covering range $\Rightarrow$ greedy

```
def minTaps(self, n, ranges):
  # reaches: max end by start
  reaches = [0 for _ in range(n + 1)]
  for i, r in enumerate(ranges):
    s = max(0, i - r)
    e = min(n+1, i + r + 1)  # [i-r, i+r+1)
    reaches[s] = max(reaches[s], e)

  ret = 0
  lo, hi = 0, 1 # [lo, hi)
  maxa = hi
  # Greedily extend coverage until we reach n
  while maxa <= n:
    # search for one tap that reaches farthest
    for i in range(lo, hi):
      maxa = max(maxa, reaches[i])

    # cannot extend coverage
    if maxa <= hi:
      return -1
    ret += 1

    lo = hi
    hi = maxa

  return ret
```

Note that the dictionary name can be short form by keys as `starts` or values as `reaches` and the latter is preferred.

Chapter 17
Backtracking

17.1 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.

17.2 Memoization

dfs can be memoized.

```
import functools

# default is maxsize=128
@functools.lru_cache(maxsize=None)
def dfs(self, *args):
    pass


@functools.lru_cache(maxsize=None)
def F(self, i, j, A):
    return min(
        (
            self.F(i, k, A) + self.F(k, j, A)
            + max(A[i:k]) * max(A[k:j])
            for k in range(i+1, j)
        ),  # generator must be parenthesized
        default=0,  # default for empty seq
    )
```

17.3 Permutations

Permutation. Generate all permutaitons of a list A.

Core Clues:

- Advancing the index `cur`

```
# itertools.permutations

def permutations(self, A, cur, ret):
    # in-place
    if cur == len(A):
        ret.append(list(A))  # clone
        return

    for i in range(cur, len(A)):
        # swap
        A[cur], A[i] = A[i], A[cur]
        self.permutations(A, cur+1, ret)
        # restore
        A[i], A[cur] = A[cur], A[i]
```

Permutation and subsequence. Return the number of all possible non-empty subsequences of letters we can permutate. For example `input = "AAB"`. The possible sequences are $A, B, AA, AB, BA, AAB, ABA, BAA$.

Core Clues:

- All elements are interconnected as neighbors
- There is no advancing of index `i`. This is more like DFS than backtracking.

```
def dfs(self, inputs, visited, cur, ret):
    # add to result set as we build the subsequences
    ret.add("".join(cur))

    for i, v in enumerate(inputs):
        # iterate all neighbors
        if not visited[i]:
            visited[i] = True
            cur.append(v)
            self.dfs(inputs, visited, cur, ret)
            cur.pop()
            visited[i] = False
```

17.4 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()
```

17.5 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.

17.5.1 Palindrome

17.5.1.1 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:

```
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]
```

17.5.2 Word Abbreviation

Core clues:

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

```
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 ''
```

17.5.3 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.

```
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.

```
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
```

17.6 Cartesian Product

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

17.6.1 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
```

17.7 Math

17.7.1 Decomposition

17.7.1.1 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.

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

17.8 Arithmetic Expression

17.8.1 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:

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

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)
```

17.8.2 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
```

17.9 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
  :s: original string
  :cur: current string builder
  :left: number of remaining left parentheses in s[0..i]
  :pi: last removed char
  :i: current index
  :rmcnt: number of remaining removals needed
  :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
```

17.10 TREE

17.10.1 BST

17.10.1.1 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
```

17.10.2 Graph

Word Search. Given an $m \times n$ grid of characters board and a string word, return true if word exists in the grid.
Core Clues:

1. dfs

```
dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
def exist(self, A, word) -> bool:
  M, N = len(A), len(A[0])
  L = len(word)
  visited = [
    [False for _ in range(N)]
    for _ in range(M)
  ]

  def dfs(r, c, i) -> bool:
    if i == L:
      return True

    if not (0 < r <= M and 0 < c <= N):
      return False

    if A[r][c] != word[i]:
      return False

    visited[r][c] = True
    for dr, dc in dirs:
      if dfs(r + dr, c + dc, i + 1):
        return True

    visited[r][c] = False
    return False

  for r in range(M):
    for c in range(N):
      if dfs(r, c, 0):
        return True

  return False
```

Chapter 18
Graph

18.1 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)
```

Convert a `parent` array to graph, `parent[i]` is the parent of node `i`.

```
G = defaultdict(list)
for i, pi in enumerate(parent):
    G[pi].append(i)
```

Complexity. Basic complexities:

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

Table 18.1: Time complexities

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

1. Acyclic
2. All connected

18.2 DFS

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 -1
    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 -1
```

Number of Islands. The most fundamental and classical problem.

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

Clue:

1. Identify one island $\Rightarrow$ dfs.
2. To count multiple islands, need to mark/color the islands $\Rightarrow$ visited.

```
class Solution:
  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)
    ]
    # visited = defaultdict(difaultdict(bool))
    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 di, dj in self.dirs:
      I = i+di
      J = j+dj
      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 18.8) to reduce each query's complexity from $O(mn)$ to $O(\log mn)$.

Without updating, DFS itself is enough.

18.3 BFS

18.3.1 BFS with Weights

Each time, a query will add an edge from vertex u to v, and find the shortest path from the *origin* to *target* after each update.

Core clues:

1. Forward vs. backward: `dist` is the distance from origin, not to destination.
2. BFS updates the distance from the source of the newly added edge, not from the origin of the graph.
3. Shortest distance ⇒ 0-1 Dijsktra only nodes with *improved* distance are added to the *pending* queue.

```
def solve(self, G, dist, queries: List[Tuple])
    ret = []
    for u, v in queries:
        G[u].append(v)
        self.bfs(G, dist, u)  # from u
        ret.append(dist[~0])

    return ret

def bfs(self, G, dist, s):
    """
    Known origin and end destination
    """
    q = deque()
    q.append(s)
    while q:
        v = q.popleft()
        for nbr in G[v]:
            if dist[nbr] > dist[v] + 1:
                # It and its descendants require update
                dist[nbr] = dist[v] + 1
                q.append(nbr)
```

x

18.3.2 Transitive Closure

Coverage closure in coordinates. Given a list of bombs indexed by (x,y) with a range r. The range of a bomb is defined as the area where its effect can be felt. The impact is circle.

Core Clues:

1. Given **pairwise** relations and query for **transitive** relations ⇒ Convert to a graph problem, two nodes are neighbors/linked if within range. To build the graph G, do pairwise checks $O(V^2)$.
2. Check max closure ⇒ dfs/bfs each node

```
def maximumDetonation(bombs):
    N = len(bombs)
    G = [[] for _ in range(N)]
    for i in range(N):
        x, y, r = bombs[i]
        for j in range(N):
            if i == j:
                continue

            x2, y2, _ = bombs[j]
            dx = x - x2
            dy = y - y2
            if dx * dx + dy * dy <= r * r:
                G[i].append(j)

    gmax = 1
    for i in range(N):
        visited = defaultdict(bool)
        visited[i] = True
        q = deque([i])

        cnt = 0
        while q:
            new_q = []
            for cur in q:
                cnt += 1
                for nbr in G[cur]:
                    if not visited[nbr]:
                        visited[nbr] = True
                        new_q.append(nbr)
            q = new_q

        gmax = max(gmax, cnt)

    return gmax
```

Evaluate Division. Given an array of variable pairs equations and an array of real numbers values, where $equations_i = [a,b]$ and $values_i$ represent the equation $values_i = a/b$.

Also given some queries, where $queries_i = [c,d]$ represents the jth query where you must find the answer for c/d. **Core Clues:**

1. Given pairwise relations and query for transitive relations ⇒ graph
2. DFS for the queries

```
def calcEquation(self, equations, values, queries):
  # Build graph: G[u][v] = u / v
  G = defaultdict(dict)
  for (a, b), w in zip(equations, values):
    G[a][b] = w
    G[b][a] = 1.0 / w

  def dfs(s, e, acc, visited):
    if s == e:
```

```
      return acc

    visited[s] = True
    for nbr, w in G[s].items():
      if not visited[nbr]:
        ret = dfs(nbr, e, acc * w, visited)
        if ret != -1.0:
          return ret

    return -1.0

  rets = []
  for s, e in queries:
    ret = dfs(s, e, 1.0, defaultdict(bool))
    rets.append(ret)

  return rets
```

18.3.3 Shortest Path & Dijkstra

Dijstrak. Shortest path in the currently processing nodes ⇒ the queue holds the distance to the elements in a priority queue as shortest first.

Swim in Rising Water. Given an integer matrix *grid* where each value $grid[i][j]$ represents the elevation at that point (i, j). It starts raining, and water gradually rises over time. At time t, the water level is t, meaning any cell with elevation less than equal to t is submerged or reachable. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distances in zero time.

Start from top left, return the minimum time until you can reach the bottom right.

Core Clues:

1. For any given path, the minimal time is the maximal elevation of a cell on the path
2. For any given cell, the minimal time to be on is the minimal of max elevations in possible paths containing the given cell.
3. Min of maxes ⇒ Dijstrak
4. In Dijstrak implementation, we need priority queue ⇒ Heap

```
INF = sys.maxsize
dirs = [(1,0),(-1,0),(0,1),(0,-1)]
def swimInWater(self, grid):
  N = len(grid)
  # dist[i][j] = minimal of max elevations
  dists = [
    [INF for _ in range(N)]
    for _ in range(N)
  ]
  dists[0][0] = grid[0][0]
  h = [(dists[0][0], 0, 0)]  # (distance_so_far, i, j)
  while h:
    dist, i, j = heapq.heappop(h)
    # first time pop is minimal
    if (i, j) == (N-1, N-1):
      return dist

    # skip outdated minimal distance
    if dist > dists[i][j]:
      continue

    for di, dj in dirs:
      I, J = i + di, j + dj
      if 0 <= I < N and 0 <= J < N:
        # transfer function from i,j to I,J
        nxt_dist = max(dist, grid[I][J])
        if nxt_dist < dists[I][J]:
          dists[I][J] = nxt_dist
          heapq.heappush(h, (nxt_dist, I, J))

  return dists[~0][~0] # redundant
```

Alternatively, we can binary search the time. The predicate is whether we can BFS to the target.

```
def bfs(self, grid, t) -> bool:
  N = len(grid)
  if grid[0][0] > t:
    return False

  visited = defaultdict(lambda: defaultdict(bool))
  q = [(0,0)]
  visited[0][0] = True
  while q:
    new_q = []
    for i, j in q:
      if (i, j) == (N-1, N-1):
        return True

      for di, dj in dirs:
        I, J = i + di, j + dj
        if 0 <= I < N and 0 <= J < N \
        and not visited[I][J] and grid[I][J] <= t:
          new_q.append((I, J))
          visited[I][J] = True
    q = new_q

  return False

def swimInWater(self, grid):
  N = len(grid)
  lo = 0
  hi = N*N
  while lo < hi:
    mid = (lo + hi) // 2
    if self.bfs(grid, mid):
      hi = mid
    else:
      lo = mid + 1

  return lo
```

0-1 Dijstrak. Only two buckets of priorities: d or $d+1$ ⇒ the queue always pop from the left the lowest weighted candidate ⇒ add the new candidate to the deque with a larger weight ⇒ checking the compared to decide append left or right.

Time compexlity: visit every node and every edge $\Rightarrow$ $O(|V|+|E|)$.

Minimum cost to form a valid path. A valid path in the grid is a path that starts from $(0,0)$ and ends at the $(\sim 0, \sim 0)$ following the signs on the grid. The sign is in $A_{i,j}$: 1 right, 2 left, 3 down, 4 up.

Core Clues:

1. Cost of changing signs $\Rightarrow$ weights in graph
2. Minimum cost $\Rightarrow$ minimum distance $\Rightarrow$ BFS $\Rightarrow$ 0-1 Dijstrak

```
def minCost(self, A):
    dirs = {
        1: (0, 1),
        2: (0, -1),
        3: (1, 0),
        4: (-1, 0),
    }

    M, N = len(A), len(A[0])
    W = [
        [sys.maxsize for _ in range(N)]
        for _ in range(M)
    ]
    W[0][0] = 0

    q = deque()
    while q:
        i, j = q.popleft()
        for orient, (di, dj) in dirs.items():
            I = i + di
            J = j + dj
            if 0 <= I < M and 0 <= J < N:
                # delta weight
                dw = 0 if A[i][j] == orient else 1
                if W[i][j] + dw < W[I][J]:
                    W[I][J] = W[i][j] + dw
                    if dw == 0:
                        q.appendleft((I, J))
                    else:
                        q.append((I, J))
    return W[~0][~0]
```

18.3.4 Multi-source BFS

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

Distance can be targetedly defined to solve the problem. BFS can start with a set of vertices in abstract level of distance, not necessarily neighboring vertices.

Shortest distance to nearest origins. Given -1 denotes obstacles, 0 denotes targets, calculate all other vertices' Manhattan distance to its nearest target. **Core clues:**

1. Shortest distance $\Rightarrow$ bfs
2. Multiple source/origins $\Rightarrow$ multi-source bfs.

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

Then it is calculated as:

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

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

def wallsAndGates(self, mat):
  M = len(mat)
  N = len(mat[0])
  q = [
    (i, j)
    for i in range(M)
    for j in range(N)
    if mat[i][j] == 0
  ]
  for i, j in q:
    for di, dj in self.dirs:
      I, J = i+di, j+dj
      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  # not min(...)
        q.append((I, J))
```

Shortest bridge of two islands. Given an binary matrix A where 1 represents land and 0 represents water. Return the smallest number of 0's you must flip to connect the two islands.

Core Clues:

1. Two islands $\Rightarrow$ coloring $\Rightarrow$ since only two islands $\Rightarrow$ color one of them is enough
2. The points on the island edge are sources $\Rightarrow$ multi-source BFS
3. No harm to treat inland points as sources.

```
def shortestBridge(self, A) -> int:
  dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
  M, N = len(A), len(A[0])
  inbound = lambda i, j: 0 <= i < M and 0 <= j < N

  def dfs(i, j, visited, color):
    A[i][j] = color
    visited[i][j] = True

    for di, dj in dirs:
      I = i + di
      J = j + dj
      if inbound(I, J) and A[I][J] == 1 \
        and not visited[I][J]:
        dfs(I, J)

  # Find first island and color it
  def color():
    visited = defaultdict(defaultdict(bool))
    for i in range(M):
```

```
    for j in range(N):
      if A[i][j] == 1 and not visited[i][j]:
        dfs(i, j, visited, color=2)
        return
color()

# Multi-source BFS
q = deque()
for i in range(M):
  for j in range(N):
    if A[i][j] == 2:
      q.append((i, j))
step = 0
visited = defaultdict(defaultdict(bool))
while q:
  new_q = []
  for i, j in q:
    for di, dj in dirs:
      I = i + di
      J = j + dj
      if inbound(I, J) and not visited[I][J]:
        if A[I][J] == 1:
          return step  # reached 1's
        if A[I][J] == 0:
          new_q.append((I, J))
          visited[I][J] = True
  q = new_q
  step += 1

return -1
```

Escape the Spreading Fire. Given a matrix *grid*. Each cell has one of three values:

- 0 represents grass
- 1 represents fire
- 2 represents a wall that you and fire cannot pass through

Starting from top-left cell to travel to the safehouse at the bottom-right cell. Every minute, you may move to an adjacent grass cell. After your move, every fire cell will spread to all adjacent cells that are not walls.

Return the maximum wait time that you can stay in your initial position before moving.

Note that even if the fire spreads to the safehouse immediately after you have reached it, it will be counted as safe.

Core Clues:

1. Every minute is a tick.
2. Spreading filre $\Rightarrow$ multi-source BFS to calculate the fire arrival time for each cell
3. Reachable from start to end? $\Rightarrow$ BFS from start to calculate travel time.
4. Numeric binary search for the maximum wait time.
5. Note that: Normal cells: must have travelTime $<$ fireTime. Goal cell: travelTime $\leq$ fireTime.

```
DIRS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class Solution:
  def maximumMinutes(self, grid) -> int:
    M, N = len(grid), len(grid[0])
    # 1) Multi-source BFS for fire arrival times
    fire = [
      [sys.maxsize for _ in range(N)]
      for _ in range(M)
    ]
    q = []
    for i in range(M):
      for j in range(N):
        if grid[i][j] == 1:  # fire source
          fire[i][j] = 0
          q.append((i, j))

    while q:
      new_q = []
      for i, j in q:
        t = fire[i][j]
        for di, dj in DIRS:
          I, J = i+di, j+dj
          if 0 <= I < M and 0 <= J < N and grid[I][J] != 2:
            if fire[I][J] > t + 1:
              fire[I][J] = t + 1
              new_q.append((I, J))
      q = new_q

    # 2) reachable after wait for T
    def reachable(T) -> bool:
      if fire[0][0] <= T:
        return False

      q = [(0, 0, T)]  # (r, c, time)
      visited = defaultdict(lambda: defaultdict(bool))
      visited[0][0] = True
      while q:
        new_q = []
        for i, j, t in q:
          if (i, j) == (M-1, N-1):
            return t <= fire[i][j]

          for di, dj in DIRS:
            I, J = i + di, j + dj
            if 0 <= I < M and 0 <= J < N \
              and grid[I][J] != 2 and not visited[I][J]:
              nt = t + 1
              if (I, J) != (M-1, N-1):
                if nt < fire[I][J]:
                  new_q.append((I, J, nt))
                  visited[I][J] = True
              else:
                # edge case
                return nt <= fire[I][J]
        q = new_q

      return False

    # 3) Binary search the maximum feasible wait
    lo, hi = 0, 10**9 + 1
    ret = -1
    while lo < hi:
      mid = (lo + hi) // 2
      if reachable(mid):
        ret = mid
        lo = mid + 1
      else:
        hi = mid
```

```
    return ret
```

18.4 Detect Cycle

1. Circle in current dfs? ⇒ `path` represents the current path, and it is reset/pop after a dfs. `path` is the `rec_stk` that represents the **recursion stack** of the dfs.
2. Next node connect back to the current node ⇒
   a. For directed graph:
      i. 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 $C \to A \to B$. `visited` should be updated only in the **end** of the dfs.
      ii. Otherwise excluding predecessor `pi` is wrong considering the case of $A \leftrightarrow B$
   b. For undirected graph:
      i. Besides `cur`, we should track predecessor `pi`. We should dfs for all neighbors except for the **predecessor** `pi`. $A - B$.
      ii. Excluding neighbors in `visited` is redundant, due to `pi`. But it is okay to double check `visited`.

Directed Graph. Detect cycles (any) in directed graph.

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

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

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

Undirected Graph. Detect cycles (any) in undirected graph.

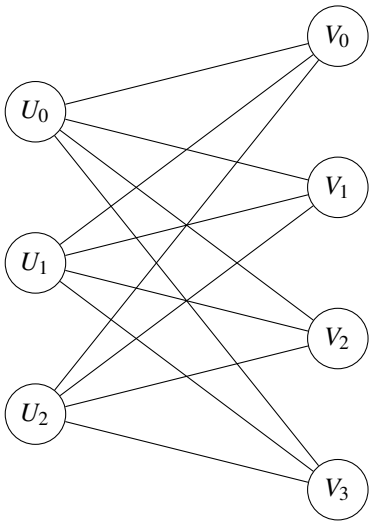
```
def dfs(self, G, cur, pi, visited, path):
  if cur in path:
    return False

  path.add(cur)
  for nbr in G[cur]:
    if nbr != pi:  # nbr not in visited
      if not self.dfs(G, nbr, cur, visited, path):
        return False
  path.remove(cur)
  visited.add(cur)
  return True
```

Alternatively, we can collapose *path* into *visited* using coloring:

```
visited[u] = 0  # unvisited
visited[u] = 1  # visiting
visited[u] = 2  # visited
```

18.5 Bipartite Graph

A bipartite graph is a graph whose vertices can be split into two disjoint sets U and V such that every edge goes between the sets (none within the same set).



Bus routes. Given an array *routes* representing bus routes where $routes_i$ is a bus route that the i-th bus loops at bus stops forever. For example, if $routes_i = [1,5,7]$.

Find the least number of buses you must take to travel from *source* to *target*.

Core clues:

1. Bus stops are connected through routes, not directly ⇒ bipartite graph as stops U, and routes V.
2. From *source*, we visisted multiple u through the bus route v ⇒ multi-source BFS

```
def numBusesToDestination(self, routes, source, target):
  # graph of stops U, graph of routes V
  U = defaultdict(set)
  V = defaultdict(set)
  for v, us in enumerate(routes):
    for u in us:
      U[u].add(v)
```

```
    V[v].add(u)

# BFS
visited_U = defaultdict(bool)
visited_V = defaultdict(bool)
visited_U[source] = True
for v in U[source]:
  visited_V[v] = True

v_q = []
for v in U[source]:
  v_q.append(v)

step = 1
while v_q:
  new_v_q = []
  for v in v_q:
    if target in V[v]:
      return step

    for u in V[v]:
      if not visited_U[u]:
        visited_U[u] = True
        for nxt_v in U[u]:
          if not visited_V[nxt_v]:
            new_v_q.append(nxt_v)
            visited_V[nxt_v] = True

  v_q = new_v_q
  step += 1

return -1
```

18.6 PATHS

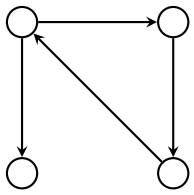
18.6.1 Visit Every Edge Once $\forall e$ - Euler Path

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. **Post-order traversal**. Dfs a vertex v's all neighbors and path. After this, process this current vertex v by `appendleft`.
3. How to avoid circle?
   a. Checking predecessor π? Not sufficient, since π can be reached again if circle.
   b. Visited? $\Rightarrow$ We must **remove** the edge e after being visited.
   c. Or we can record the visited edge $e = (u, v)$ in a dict `visited`, but only if no duplicates in E; otherwise we must use a counter and it becomes the same as removing.



Sorted for lexical order.

```
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
        # thus not for-loop
        nbr = G[cur].popleft()
        self.dfs(G, nbr, ret)

    ret.appendleft(cur)
```

Alternatively, instead of using sorted list, using `heapq`.

```
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)
```

18.6.2 Vist Every Vertex Once $\forall v$ - Hamiltonian Path, NP

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.

18.7 TOPOLOGICAL SORTING

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

18.7.1 Algorithm

Core clues:

1. **DFS neighbors first**. For a given vertex v, if the neighbors of current node is ¬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`.
3. **Acyclic**. Need to detect cycle using *path*; thus the dfs need to construct result queue *ret* and detect cycle simultaneously - by using two sets: *visited* and *path*. The *path* can also be named as `rec_stk` to represent recursion stack in DFS.

Notice:

1. Need to check ascending order or descending order.

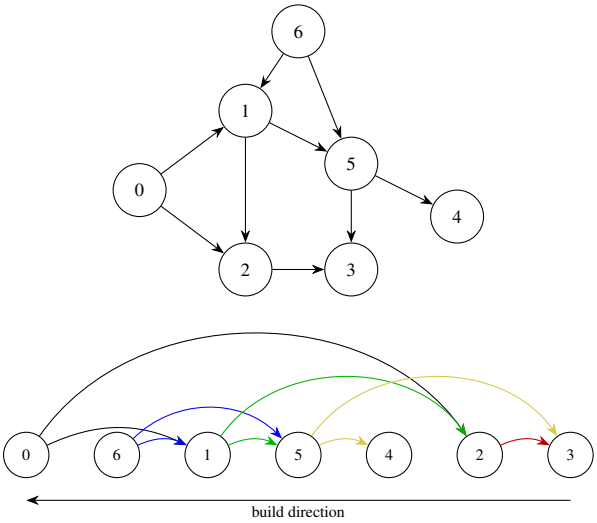


Fig. 18.1: Unsorted DAG to Topologically Sorted

```
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:  # cycle
    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`, i.e. coloring.

```
# encode the visited using 0, 1, 2
def dfs_topo(
  self,
  G: Dict[int, List[int]],
  u: int,
  visited: Dict[int, int],
  ret: Deque,
) -> bool:
  """
  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.

18.7.2 Applications

1. Course scheduling problem with pre-requisite.
2. In a tree, find the closest ancestor y of x s.t. sastisfying some predicate.

In a tree, find the closest ancestor y of node x that has the same value $s[x] = s[y]$ and set x parent to y.

Core clues:

1. Naively dfs updating the tree results in TLE of $O(N^2)$. Thus we need to do a topological dfs on the tree for $O(N)$.
2. During topological dfs, the recursion stack *path* holds **extra** information. The *path* holds the map of a stack of ancestors with a given value, with the last one as the closest.

```
def topo(self, G, cur, path: Dict[List], parents, s):
    # topological dfs
    val = s[cur]
    if len(path[val]) > 0:
        pi = path[val][~0]
        parent[cur] = pi

    path[val].append(cur)
    for v in G[cur]:
        self.topo(G, v, path, parent, s)
    path[val].pop()
```

Alien Dictionary . Given a list of words, and they are lexicially sorted. Return the lexical order of chars. Input: `words = ["wrt","wrf","er","ett","rftt"]`, Output: `"wertf"`.
Core Clues:

1. Relax the problem: single char ⇒ topo sort
2. The difficult parts lie in how to construct the graph G.
3. Multi chars: first char diff btw two words tells the **relative** order
4. When build the graph G, we need to check acyclic
5. When topo sort, we need to check acyclic

```
class Solution:
  def construct_graph(self, words: List[str]):
    G = defaultdict(set)  # use set to avoid duplicate edges

    for w in words:
      for c in w:
        G[c]

    # compare two words
    for i in range(len(words) - 1):
      w1, w2 = words[i], words[i + 1]

      # invalid if w2 is a strict prefix of w1
      if len(w1) > len(w2) and w1.startswith(w2):
        return G, False

      for c1, c2 in zip(w1, w2):
        # first diffs
        if c1 != c2:
          if c2 not in G[c1]:
            G[c1].add(c2)
          break

    return G, True

  def topo_dfs(self, G, u, visited, ret) -> bool:
    visited[u] = 1
    for nbr in G[u]:
      if visited[nbr] == 1:
        return False  # not acyclic
      if visited[nbr] == 0:
        if not self.topo_dfs(G, nbr, visited, ret):
          return False

    visited[u] = 2
    ret.appendleft(u)
    return True

  def alienOrder(self, words) -> str:
    G, acylic = self.construct_graph(words)
    if not acylic:
      return ""

    # coloring: # 0 not visited, 1 visiting, 2 visited
    ret = deque()
    visited = defaultdict(int)

    for u in G.keys():
      if visited[u] == 0:
        if not self.topo_dfs(G, u, visited, ret):
          return ""

    return "".join(ret)
```

18.8 UNION-FIND

18.8.1 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). Don't introduce a dummy node `-1`. Self-equaling is enough.
3. Otherwise, `pi[x] = find(pi[x])`, recursively. Need to find `x`'s `pi` to make progress in the recursive function.
4. When *union*, find the ancestors of both nodes, and set one's ancestor to the other's.
5. Lazily init `pi[x]` during *find* since *find* is invoked inside *union*.

```
class UnionFind:
    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

    # find root
```

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

        # path compression
        pi = self.pi[x]
        if x != pi:
          root = self.find(pi)
          self.pi[x] = root
        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(
    uf.find(x)
    for x in uf.pi.keys()
))
```

Alternatively using iterative:

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

    cur = x
    while cur != self.pi[cur]:
        cur = self.pi[cur]

    self.pi[x] = cur
    return cur
```

Note that it only updates `self.pi[x]`, so nodes between `x` and the root aren't compressed. That's correct but less efficient than full compression.

Improvements:

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

18.8.2 Optimized Union Find

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:  # or DisjointSet
  def __init__(self):
    self.pi = {}  # item -> pi
    self.sz = {}  # root -> size

  def find(self, x):
    """path compression"""
    if x not in self.pi:
```

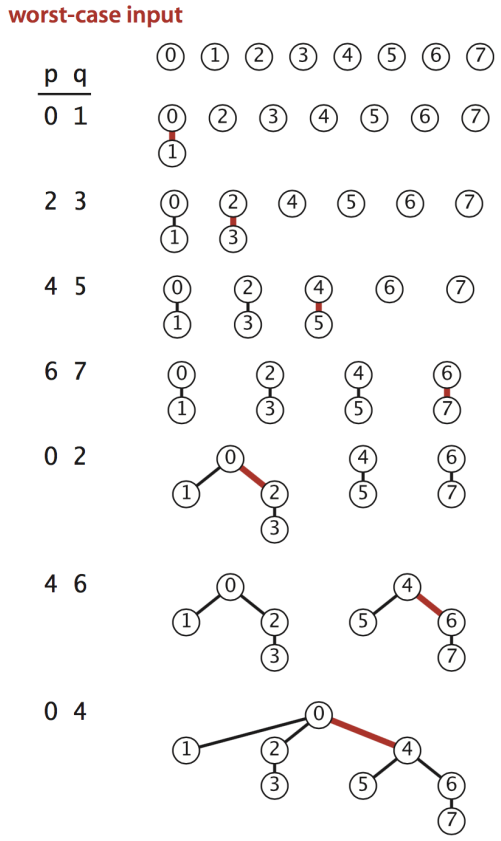


Fig. 18.2: Weighted union find traces, on every union operation

```
      self.pi[x] = x
      self.sz[x] = 1
      return x

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

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

  if pi_x != pi_y:
    if self.sz[pi_x] > self.sz[pi_y]:
      pi_x, pi_y = pi_y, pi_x
      # size balancing, remove the smaller one
    self.pi[pi_x] = pi_y
    self.sz[pi_y] += self.sz[pi_x]
    del self.sz[pi_x]

def is_union(self, x, y):
  if x not in self.pi or y not in self.pi:
    return False
  return self.find(x) == self.find(y)

def __len__(self):
```

```
    """number of sets"""
    return len(self.sz)  # only root nodes have size
```

18.8.3 Complexity

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

18.8.4 MST

Minimum spanning tree. A minimum spanning tree (MST) is a subset of a graph's edges that connects all the vertices while minimizing the total edge weight.

Kruskal's algorithm. Greedily add minimal edge.

Core clues:

1. Vertices $v \in V$ are divided into different sets
2. **Greedily** use min edges to unionize the sets if not in the same union, by sorting the edge weights
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.is_union(u, v):
      A.append((u, v))
      uf.union(u, v)
```

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

Chapter 19
Dynamic Programming

19.1 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* i.e. *up to* index i (i **optional**)

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

Firstly,

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

should be reduced to

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

Secondly,

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

should be reduced to

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

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

Notice:

- $F_{i,o}$ depends on $F_{i-1,o} \Rightarrow$ must iterate i **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 .

$$\pi[i] = j$$

19.2 Sequence

19.2.1 Best Trailing Subarrays - Kadane's Algorithm

Kadane's algorithm tracks the *best* subarray ending AT each position.

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:

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

We can do index rewrite - let F_i be the max subarray sum for $A[:i]$ up to A_{i-1}.

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

Number of 1s-subarrays (1D). Find the total number of 1s-subarrays. For example, for $A = [1,0,1,1,1]$, there are $7 = 1+1+2+3$ 1s-subarrays.

Let F_j be number of 1s-subarrays ending AT $j-1$:

$$F_j = \begin{cases} F_{j-1}+1 & \text{if } A_{j-1}=1, \\ 0 & \text{otherwise.} \end{cases}$$

The total number of 1s-subarrays is $\sum F$.

Number of 1s-submatrix (2D). For a 2-D matrix, find the total number of 1s-submatrices. Consider a 2-row matrix:

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

Project 2D into 1D:

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

$$F_j = \begin{cases} F_{j-1}+1 & \text{if } M_{lo:hi,j-1}=1, \\ 0 & \text{otherwise.} \end{cases}$$

```
def num_submat(mat):
    ret = 0
    M = len(mat)
    N = len(mat[0])
    for lo in range(M):
        # is all ones for mat[lo:hi][j]
        is_ones_col = [True for j in range(N)]
        for hi in range(lo+1, M+1):
            for j in range(N):
                is_ones_col[j] &= mat[hi-1][j]
                # mem saving

            F = [0 for _ in range(N+1)]
            for j in range(1, N+1):
                F[j] = F[j-1] + 1 if is_ones_col[j-1] \
                       else 0
                ret += F[j]

    return ret
```

Maximum Subarray Gain. Find the maximum subarray gain of A. A gain is defined as followed:

$$gain(A_i) = \begin{cases} +1 & \text{if } A_i = t, \\ -1 & \text{if } A_i = k, \\ 0 & \text{otherwise.} \end{cases}$$

Let F_i be the maximum subarray gain ending at A_i

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

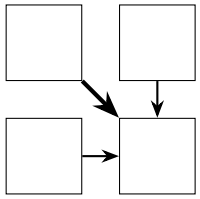
19.2.2 Single-state dp

Longest common subsequence (LCS). Let $F_{i,j}$ be the LCS at string $A[:i+1]$ and $B[:j+1]$, up to A_i and B_j. Note subsequence does not have to be continuous.

We have two situations: $A_i = B_j$ or not.

$$F_{i,j} = \begin{cases} F_{i-1,j-1}+1 & \text{if } A_i = B_j, \\ \max\Big(F_{i-1,j}, F_{i,j-1}\Big) & \text{otherwise.} \end{cases}$$

No need to set 0 since it is subsequence.



```
def lcs_edit(A, B):
    """Return (=, -, +) git diff via classic LCS."""
    M, N = len(A), len(B)
    F = [
        [0]*(M+1)
        for _ in range(N+1)
    ]
    for i in range(M):
        for j in range(N):
            if A[i] == B[j]:
                F[i+1][j+1] = F[i][j] + 1
            else:
                F[i+1][j+1] = max(F[i][j+1], F[i+1][j])
```

To reconstruct the edits of '+/-/=' with A as the base string:

1. Forward reconstruct? Greedy? $aa\alpha\beta b$ vs. $ab\gamma\delta c$, cannot decide by looking ahead by 1 position; thus become a search problem
2. Instead backward reconstruct since $F_{i,j}$ is defined up to A_i, B_j.

```
def reconstruct(A, B, F):
    M, N = len(A), len(B)
    ret = []
    # backward pass
    i, j = M, N
    while i or j:
        if i and j and A[i-1] == B[j-1]:
            ret.append(("=", a[i-1]))
            i -= 1
            j -= 1
        elif j and (i == 0 or F[i][j-1] >= F[i-1][j]):
            # B[:j-1] has longer LCS than A[:i-1]
            ret.append(("+", B[j-1]))
            j -= 1
        else:
            ret.append(("-", A[i-1]))
            i -= 1
    ret.reverse()
    return ret
```

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

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

Because it is not necessary that $F_{i,j} \geq F_{i',j'}, \forall i, j \cdot i > i', j > j'$, as $F_{i,j}$ can be 0, thus $gmax = \max(F)$.

Longest increasing subsequence (LIS). Find the longest increasing subsequence of an array A.

let F_i be the LIS length ends at A_i.

$$F_i = \max(F_j + 1, \forall j < i \cdot A_j < A_i)$$

Then the global *maxa* is:

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

Time complexity: $O(n^2)$

How to improve time complexity?

Notice that F_i is taking max over previous F_j, which makes $F_i > F_j$, although F as a whole is not monotonic increasing.

To binary search to achieve $O(n \log n)$, we need to maintain monotonic states. F records length, can we do the inverse - V_i records some element/value of the LIS of some length i.

Let V_j be the smallest tail value of the LIS of length $j+1$.

$$V_j = \arg\min_k \{F_k = j+1, \forall k < j\}$$

V is monotonic increasing, maintaining those minima is just monotone queue optimization of recurrence of F's formula.

Core Clues:

1. `V`: V_j = min possible tail value of a subseq length $j+1$
2. `idx`: Index where each tail value is (index in A)
3. `pi`: predecessor indices for reconstruction

```
def lis(A):
    V = []
    pi = [-1 for _ in A]  # defaultdict(lambda: -1)
    for i in range(len(A)):
        j = bisect.bisect_left(V, A[i],
            key=lambda e: A[e])

        if j < len(L):
            V[j] = i
        else:
            V.append(i)

        pi[i] = V[j-1] if j > 0 else -1

    # rebuild
    ret = []
    cur = V[~0]
    while cur != -1:
        ret.append(A[cur])
        cur = pi[cur]

    return ret[::-1]

print(lis([10,9,2,5,3,7,101,18]))   # -> [2, 3, 7, 18]
```

Ref search section 8.3.1.

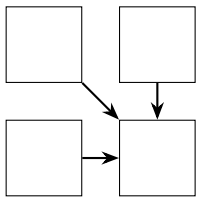
Maximum sum of non-adjacent cells. Get the maximum sum of non-adjacent cells of an array A.

Let F_i be the maximum sum of non-adjacent cells for $A[:i]$, up to A_{i-1}. 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 $B[:j]$.



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

H-index Given an array of citations A of a researcher, write a function to compute the researcher's h-index. Find the highest number h such that the researcher has at least h publications that have each been cited at least h times.

Need some re-representation of information:

1. Relax the problem $\Rightarrow$ exact i citations: let C_i be the #paper with $= i$ citations.
2. The original problem $\Rightarrow \geq i$ citations: let F_i be the #paper with $\geq i$ citations.

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

Backward DP. DP takes $O(n)$.

```
def hIndex(A):
    N = len(A)
    C = [0 for _ in range(N+1)]
    for e in A:
        C[min(e, N)] += 1

    F = [0 for _ in range(N+2)]
    for i in range(N, -1, -1):
        F[i] += F[i+1] + C[i]
        if F[i] >= i:
            return i

    return 0
```

Given $F_i > F_{i+1}$, as F 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]`, a boolean.

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.

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

Maximum Earnings From Taxi. Given n, representing driving from point 1 to point n to make money by picking up passengers. Given *rides*, representing i-th passenger ride from point $start_i$ to point end_i who is willing to give a tip_i dollar tip. The reward is $tip_i + end_i - start_i$. Find the maximal reward.

Core clues:

1. Going from 1 to $n \Rightarrow$ sort the passengers by *start* or *end* $\Rightarrow$ sort through a dict
2. Need to consider vacant $\Rightarrow F_i = max(F_i, F_{i-1})$.
3. One pass from 1 to n

Look-back DP.

Let F_i be the max reward at point i

$$F_i = \max\Big(F_{i-1}, F_j + (tip_j + i - j) \cdot \forall j\Big)$$

```
def maxTaxiEarnings(self, n, rides):
  ends = defaultdict(list)
  for s, e, tip in rides:
    ends[e].append((s, tip))

  F = [0 for _ in range(n+1)]

  for i in range(1, n + 1):
    F[i] = max(F[i], F[i-1])
    if i in ends:
      for s, tip in ends[i]:
        F[i] = max(F[i], F[j] + tip + i - s)

  return F[n]
```

Look-ahead DP.

```
def maxTaxiEarnings(self, n, rides):
  starts = defaultdict(list)
  for s, e, tip in rides:
    starts[s].append((e, tip))

  F = [0 for _ in range(n + 1)]

  for i in range(1, n + 1):
    F[i] = max(F[i], F[i-1])
    if i in starts:
      for e, tip in starts[i]:
        F[e] = max(F[e], F[i] + tip + e - i)

  return F[n]
```

19.2.3 Dual-state dp

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

- Maximal product $\Rightarrow$ Let L_i be the largest product end at A_i.
- Product can be negative $\Rightarrow$ Let S_i be the smallest product end at A_i.
- The states can be negative.

$$S_i = \min\Big(A_i,\ S_{i-1} \cdot A_i,\ L_{i-1} \cdot A_i\Big)$$
$$L_i = \max\Big(A_i,\ S_{i-1} \cdot A_i,\ L_{i-1} \cdot A_i\Big)$$

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. 19.1: Trapping Rain Water

Let L_i be the $\max(A[:i])$; let R_i be the $\max(A[i:])$. The dp of obtaining L, R is trivial.

The the total volume *vol*:

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

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.

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

Notice in python implementation, the two states are interleaved and interdependent.

```
def maxzigzag(self, A):
    N = len(A)
    U = [1 for _ in range(N)]
    D = [1 for _ in range(N)]
    gmax = 1
    for i in range(1, N):
        for j in range(i):
            if A[i] > A[j]:
                U[i] = max(U[i], D[j] + 1)
            elif A[i] < A[j]:
                D[i] = max(D[i], L[j] + 1)

            gmax = max(gmax, U[i], D[i])

    return gmax
```

Greedy compression. Let D_i be the length of longest zigzag subsequence up to A_I, with last step downward.

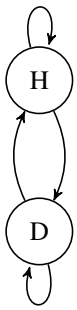
$$D_i = \begin{cases} U_{i-1} + 1 & \text{if } A_i < A_{i-1} \\ D_{i-1} & \text{otherwise} \end{cases}$$

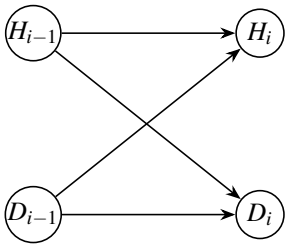
For D_i, we only need to check $A_{i-1} > A_i$ since otherwis $A_{i-3}, A_{i-2}, A_{i-1} \ldots$ keeps going up.

$$H_i = \begin{cases} H_{i-1} & \text{if upward trend continues} \\ D_{i-1} + 1 & \text{otherwise, i.e. } A_i > A_{i-1} \end{cases}$$

Note that

$$|H_i - D_i| \leq 1 \cdot \forall i$$






```
def maxzigzag(self, A):
    N = len(A)
    U = [1 for _ in range(N)]
    D = [1 for _ in range(N)]
    for i in range(1, N):
        D[i] = U[i-1] + 1 if A[i] < A[i-1] else D[i-1]
        U[i] = D[i-1] + 1 if A[i] > A[i-1] else U[i-1]

    return max(U[N-1], D[N-1])
```

Buy low sell high. Given a stock price timeseries A, find the maximum profit, with at most k transactions. Let $L_{i,j}$ be the global max ending at A_i with j transactions. Let $G_{i,j}$ be the global max ending at or before (i.e. up to) A_i with j transactions. Kadance-style.

$$\Delta = A_i - A_{i-1}$$
$$L_{i,j} = \max(G_{i-1,j-1} + \Delta, L_{i-1,j} + \Delta)$$
$$G_{i,j} = \max(L_{i,j}, G_{i-1,j})$$

```
def dp(A, k):
    n = len(A)
    L = [0 for _ in range(k+1)]  # local max
    G = [0 for _ in range(k+1)]  # global max
    ret = 0
    for i in range(1, n):
        delta = A[i] - A[i-1]
        for j in range(k, 0, -1):
            L[j] = max(G[j-1] + delta, L[j] + delta)
            G[j] = max(L[j], G[j])
            ret = max(ret, G[j])

    return ret
```

19.2.4 Synchronized States

Cherry pickup. Given a $n \times n$ matrix M, 0 is a pass through block, 1 is a reward, -1 is block. Find the maximum reward going from $(0,0)$ to $(\sim 0, \sim 0)$ and then back to $(0,0)$.

Core clues:

1. If the problem is relaxed to two passes in two parallel matrics, it is trivial. However, the reward can only take once. Thus union the reward, not add the rewards.

2. In forward path, we take k steps from the start. In backward path, we take k steps to reach the start.
3. Equivalently, from start to end and then back to start $\Rightarrow$ treat them as start to end twice

The basic transition for one pass

$$F_{i,j} = \max(F_{i,j}, F_{i-1,j}, F_{i,j-1})$$

Given k, we can use i to determine j as $\boxed{i+j=k}$. We only need to know the forward pass row index i and backward i'.

$$\begin{aligned} J_{i,i',k} = \max(&J_{i,i',k-1}, J_{i-1,i',k-1}, \\ &J_{i,i'-1,k-1}, J_{i-1,i'-1,k-1}) \\ &+ reward(M_{i,j}, M_{i',j'}) \end{aligned}$$

The *reward* function is

$$reward(M_{i,j}, M_{i',j'}) = \begin{cases} -\infty & \text{if } M_{i,j} = -1 \vee M_{i',j'} = -1 \\ M_{i,j} + M_{i',j'} & \text{if } i \neq i' \\ M_{i,j} & \text{if } i = i' \end{cases}$$

19.2.5 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 $s[:i]$, end at $s[i-1]$.

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

Regex. TODO

19.3 GRAPH

19.3.1 Binary Graph

Maximal square. Find the largest square 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).

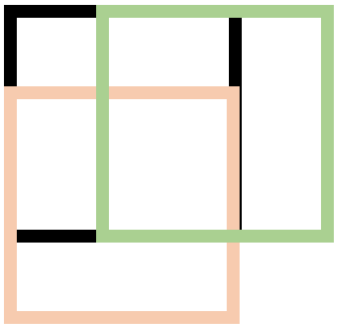


Fig. 19.2: Expand the maximal square

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

19.3.2 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 (up to) n vertices.

Then we can two options:

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

, 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 $-\infty$.

19.4 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, i.e. ending at s_{i-1}.

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

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

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

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

Reconstruct the sentence from F_i with backtracking:

```
def build(self, F, i, cur, ret):
    if cur_index == 0:
        ret.append(" ".join(cur[::-1]))
        return

    # backtracking
    for word in F[i]:
        cur.append(word)
        self.build(F, i-len(word), cur, ret)
        cur.pop()
```

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

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

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

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

- If i depends on $i-1 \Rightarrow$ forward builidng.
- If j depends on $j+1 \Rightarrow$ backward building.

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

for i in range(n-2, -1, -1):
    for j in range(i+2, n+1):
        F[i][j] = F[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 A 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 segments.

Let P_i be the number of violations (i.e. `"b"`) in prefix $A[:i]$. Let S_i be the number of violations (i.e. `"a"`) in suffix $A[i:]$.

$$\min(P_i + S_i \cdot \forall i)$$

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

Define a cost function:

$$\text{cost}(chr, x) = \begin{cases} 0 & \text{if } chr = x, \\ 1 & \text{if } chr \neq x \end{cases}$$

1. Let $F_{i,0}$ be the cost of changing $A[:i]$ into `"a*"`, ending with `"a"`.
2. Let $F_{i,1}$ be the cost of changing $A[:i]$ into `"a*b*"`, ending with `"b"`.
3. Let $F_{i,2}$ be the cost of changing $A[:i]$ into `"a*b*c*"`, ending with `"c"`.

The transition functions:

$$\begin{aligned} F_{i,0} &= F_{i,0} + \text{cost}(A_{i-1}, a), \\ F_{i,1} &= \min\left(F_{i-1,0},\ F_{i-1,1}\right) + \text{cost}(A_{i-1}, b), \\ F_{i,2} &= \min\left(F_{i-1,1},\ F_{i-1,2}\right) + \text{cost}(A_{i-1}, c). \end{aligned}$$

The result:

$$\min(F_{0,0}, F_{0,1}, F_{0,2})$$

ab subsequence. Given a string A, a string pattern B that is a subsequence of A. We define an operation as removing a character at an index *idx* from A such that:

1. *idx* is an element of *removable*.
2. pattern B remains as a subsequence of A.

Find the maximal possible number of removals.

Forward DP. Let $F_{i,j}$ be the maximal operations at $A[:i]$ and $B[:j]$

$$F_{i,j} = \max \begin{cases} F_{i-1,j} + 1 & \text{take } A_{i-1} \text{ if } i-1 \text{ in } removable \\ F_{i-1,j} & \text{skip } A_{i-1} \text{ if } i-1 \text{ not in } removable \\ F_{i-1,j-1} & \text{skip } B_{j-1} \text{ with } A_{i-1} \text{ if } A_{i-1} = B_{j-1} \end{cases}$$

Backward DP. Let $F_{i,j}$ be the maximal operations at $A[i:]$ and $B[j:]$

$$F_{i,j} = \max \begin{cases} F_{i+1,j} + 1 & \text{take } A_i \text{ if } i \text{ in } removable \\ F_{i+1,j} & \text{skip } A_i \text{ if } i \text{ not in } removable \\ F_{i+1,j+1} & \text{skip } B_j \text{ with } A_i \text{ if } A_i = B_j \end{cases}$$

19.5 Divide & Conquer

19.5.1 Tree

Number of different BSTs. It can be solved using Catalan number (Section 13.4), 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,

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

19.5.2 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]` reward. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

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

Core clues:

1. Simplify the problem: what if A only contains on element?
2. Sub-problem: What is the reward of bursting `A[i:j]` $\Rightarrow$ Divide & Conquer. Burst A_k where $k \in [i, j)$.

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

$$F_{i,j} = \max\left(F_{i,k} + F_{k+1,j} + A_{i-1} \cdot A_k \cdot A_j \cdot \forall k \in [i, j)\right)$$

, where k is the one to burst.

Since $F_{i,j}$ derived from smaller $F_{i',j'}$, we need to expand the F from smaller-length F.

```
def burst(self, A):
    A = DefaultList(A)
    N = len(A)
    F = [
        [0 for _ in range(N+1)]
        for _ in range(N+1)
    ]
    for l in range(1, N+1):
        for i in range(N-l+1):
            j = i + l
            F[i][j] = max(
                F[i][k] + F[k+1][j] + A[i-1]*A[k]*A[j]
                for k in range(i, j)
            )

    return F[0][N]

def burst(self, A):
    A = DefaultList(A)
```

```
    N = len(A)
    F = [
        [0 for _ in range(N+1)]
        for _ in range(N+1)
    ]
    for i in range(N+1, -1, -1):
        # j determines i's range
        for j in range(i+1, N+1):
            F[i][j] = max(
                F[i][k] + F[k+1][j] + A[i-1]*A[k]*A[j]
                for k in range(i, j)
            )

    return F[0][N]

def get(A, i):
    return A[i] if 0 <= i < len(A) else 1

# alternatively
from collections import UserList

class DefaultList(UserList):
    def __getitem__(self, i):
        if 0 <= i < len(self.data):
            return self.data[i]
        return 1
```

19.6 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.

19.6.1 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.

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

Advanced backpack problem[4].

[4] Nine Lectures in Backpack Problem.

19.6.2 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: $O(n^2k)$

19.7 Local and Global Extremes

19.7.1 Long and short stocks

19.7.1.1 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 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.

19.7.1.2 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 before)

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 $i-1$.

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

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

For M_i, simply,

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

19.8 GAME THEORY - MULTI PLAYERS

Assumption: the opponent take the optimal strategy for herself.

19.8.1 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.

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

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

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

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

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

Chapter 20
Interval

20.1 Interval Merger

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 & merge 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 = list(filter(lambda x: x.end < s, itvls))
    right = list(filter(lambda x: x.start > e, itvls))
    if len(left) + len(right) != len(itvls):
        s = min(s, itvls[len(left)].start)
        e = max(e, itvls[-len(right)-1].end)

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

20.2 Meeting Rooms

Each meeting has a start and an end $[start, end)$.

Meeting room requried. Given an array of meeting time $itvls$.

Core Clues:

1. Sort by *start*
2. Put *end* into heap
3. Record the max heap size

```
def minMeetingRooms(self, itvls) -> int:
  itvls.sort(key=lambda x: x.start)

  ret = 0
  end_h = []

  for s, e in itvls:
    if end_h and end_h[0] <= s:
      heapq.heappop(end_h)

    heapq.heappush(end_h, e)
    ret = max(ret, len(end_h))

  return ret
```

Delayed meetings. Given n as number of rooms, and a list of meetings $itvls$. Return the number of the room that held the most meetings.

1. Each meeting will take place in the unused room with the lowest number.
2. If there are no available rooms, the meeting will be delayed until a room becomes free. The meeting duration unchange.
3. When a room becomes unused, meetings that have an earlier original start time should be given the room.

Core Clues:

1. Intervals $\Rightarrow$ sort intervals by *start* and use a heap of busy rooms (processing) to hold their ending time *end*.
2. n rooms $\Rightarrow$ need to keep tracks of currently available rooms, and number of ever hosted meetings per room.
3. We don't limit busy heap size by n.
4. How to handle overflow $\Rightarrow$ Overflow are scheduled ahead into the busy heap with delayed time.
5. Get the smallest room index to schedule a meeting $\Rightarrow$ use another heap to maintain the indices.

```
def mostBooked(self, n, itvls) -> int:
    itvls.sort()  # by start time
    avail_h = [i for i in range(n)]
    heapq.heapify(avail_h)   # rooms sort by index
    busy_h = []  # [(end, room)]
    counters = [0 for _ in range(n)] # meeting count per room

    for s, e in itvls:
        # process start
        while busy_h and busy_h[0][0] <= s:
            end, room = heapq.heappop(busy_h)
            heapq.heappush(avail_h, room)

        # process end
        if avail_h:
            room = heapq.heappop(avail_h)
            heapq.heappush(busy_h, (e, room))
            counters[room] += 1
        else:
            end_earliest, room = heapq.heappop(busy_h)
            delayed_e = end_earliest + (e-s)  # delay meeting
            heapq.heappush(busy_h, (delayed_e, room))
            counters[room] += 1

    return counters.index(max(counters))  # argmax
```

Alternatively:

1. Intervals $\Rightarrow$ sort by *start* and use a heap to hold *end*.
2. n rooms $\Rightarrow$ limit the heap size to n, then need to handle overflow.
3. Need to know the room that hold the max $\Rightarrow$ we need to maintain room indices.
4. Overflow $\Rightarrow$ a pending heap to hold meetings waiting for rooms, sort by start time
5. How to get new start time $\Rightarrow$ earliest end time of a room, and the new end time is naturally known by the duration.
6. This is complicated since it requires heaps, time pointers and meeting indices.

20.3 EVENT-DRIVEN ALGORITHMS

20.3.1 Introduction

The core philosophy of event-driven algorithm:

1. **Events**: define *events*; the events are sorted by time of appearance.
2. **Accumulator**: define *accumulator* as the accumulated impacts of the event.
3. **Transition**: define *transition functions* among events impacting the accumulator.

20.3.2 Line Sweeping

Maximal Overlaps. Given a list of intervals, find the max number of overlapping intervals. This is the same as number of meeting rooms required.

Core clues:

1. **Events**: Every new start of an interval is an event. Sort intervals by *start*.
2. Need to maintain a list of *ends* that are covered by the current *start*, i.e. $end_i < start, \forall i$.
3. **Accumulator**: When iterating the current element, maintain the maximum number of overlaps. The smallest *end* should be covered by the current *start*; $\Rightarrow$ sort the *ends* .
4. **Transition**: Sorted by a heap, stores the *ends* of the intervals. Put the *end* 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
```

The horizontal line balancing above and below. Given a 2D integer array *squares*. Each $squares_i = [x_i, y_i, l_i]$, representing the coordinates of the bottom-left point and the side length of a square parallel to the x-axis.

Find the minimum y-coordinate value of a horizontal line such that the total area of the squares above the line equals to that below the line. Note: Squares may overlap. Overlapping areas should be counted multiple times.

Core clues:

1. **Events**: Every y_i repreents a new square, with a begining and an end.
2. **Accumulator**: Define a quantity $q(y) = areaBelow(y)$, by the horizontal line y.
3. **Transition**: The rate of change of $q(y)$ is $rate$. The $rate$ is the accumulated squares intersecting at y. Each event contains a $\Delta rate$ that impacts the accumulated rate. More formally,

$$\frac{\mathrm{d}}{\mathrm{d}y} q(y) = rate(y)$$

$$rate(y) = \sum_{\substack{\text{all squares } i \\ y_i \le y < y_i + l_i}} l_i$$

$$rate(y) \ge 0$$

$$\frac{\mathrm{d}}{\mathrm{d}y} rate(y) = \Delta rate_y \text{ from } events$$

```
def separateSquares(self, squares):
    """
    Event driven. Line Sweep.
    q(y) = areaBelow(y)
    dq/dy = slope, the rate of change of q(y)
    q(y) is monotonically increasing
    """
    events = []
    total_area = 0
    for x, y, l in squares:
        # (y-coordiate, delta_rate)
        # x not relevant
        events.append((y, l))
        events.append((y+l, -l))
        total_area += l*l

    events.sort()
    target = total_area / 2

    q = 0
    rate = 0
    prev_y, _ = events[0]
    for y, d_rate in events:
        q += (y - prev_y) * rate
        if q >= target:
            return y - (q - target) / rate
            # eager, won't fall below prev_y

        rate += d_rate
        prev_y = y

    return prev_y
```

20.4 RANGE BY PIVOT

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

$$| \leftarrow p \rightarrow |$$

Dedup. 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.

$$| \leftarrow p$$

Appendix A
Balanced Search Tree

A.1 2-3 Search Tree

A.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.

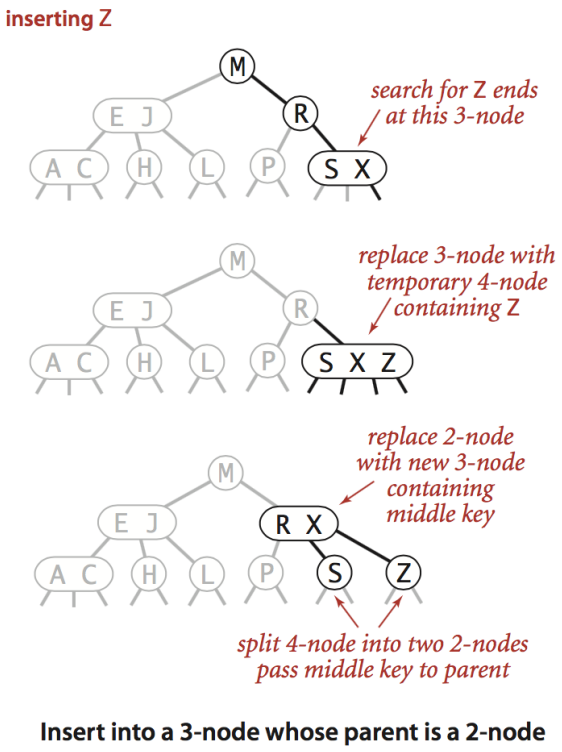


Fig. A.1: Insertion 1

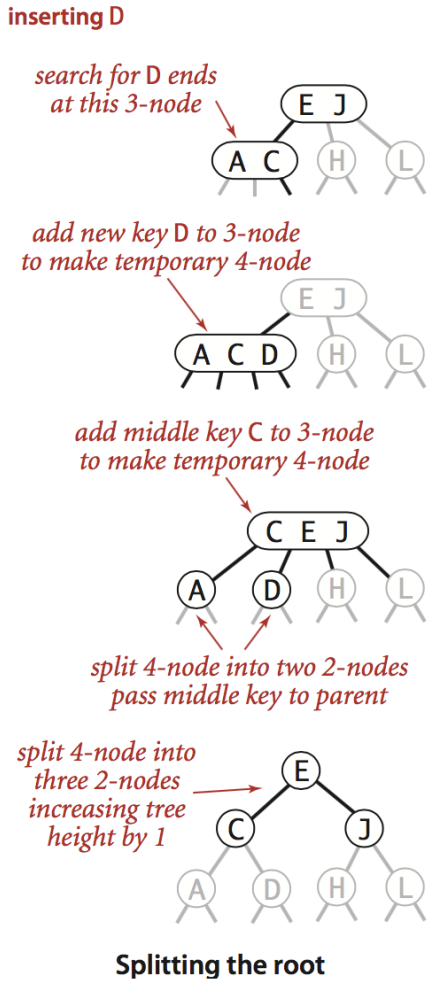


Fig. A.2: insert 2

A.1.2 Splitting

Summary of splitting the tree.

A.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

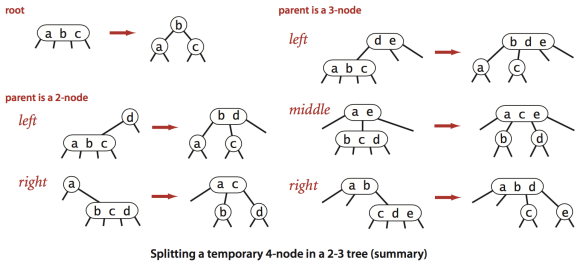


Fig. A.3: Splitting temporary 4-ndoe summary

tree increase by one? When every node on the search path from the root is a 3-node

A.2 Red-Black Tree

A.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.

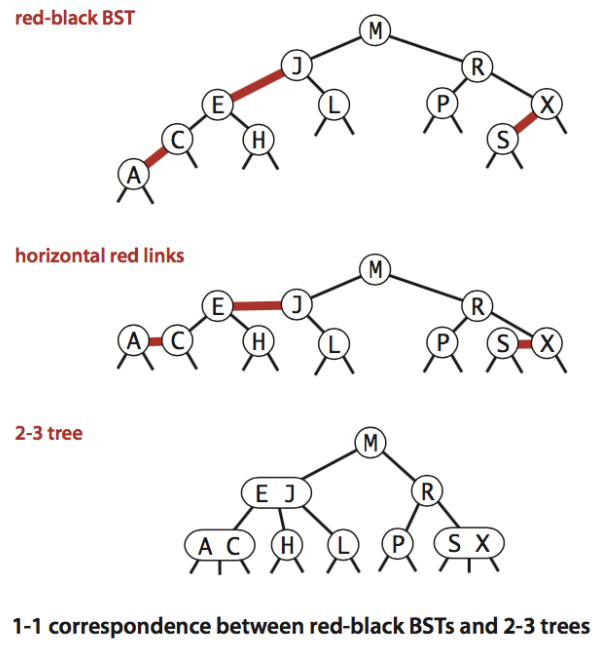


Fig. A.4: RB-tree and 2-3 tree

A.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.

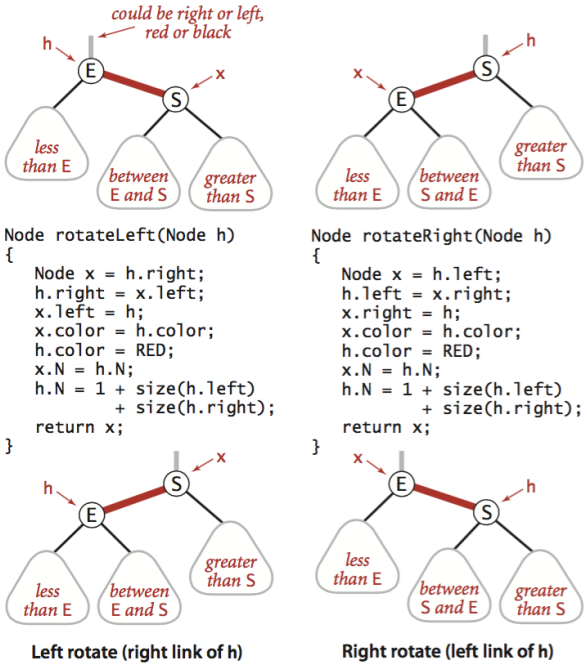


Fig. A.5: Rotate left/right

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:

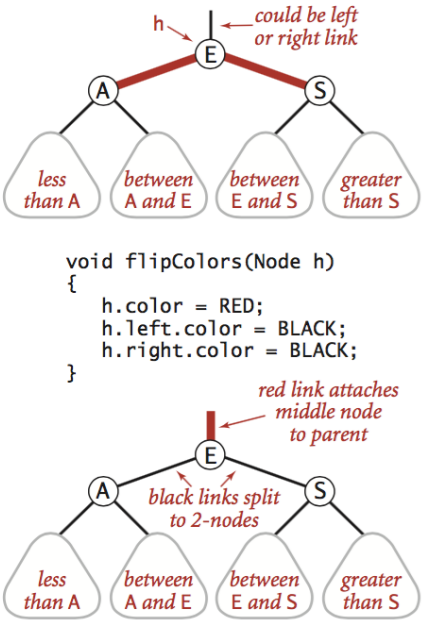


Fig. A.6: Flip colors

```
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-A.7. Insert into a single 3-node: Figure-A.8

Deletion. Deletion is more complicated.

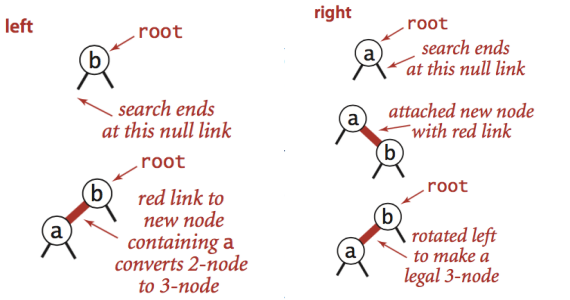


Fig. A.7: (a) smaller than 2-node (b) larger than 2-nod

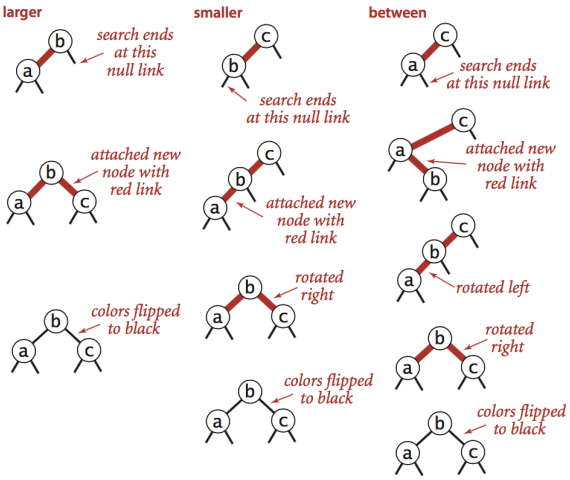


Fig. A.8: (a) larger than 3-node (b) smaller than 3-node (c) between 3-node.

A.3 B-TREE

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

A.3.1 Basics

Half-full principle:

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

Table A.1: Nodes at least half-full

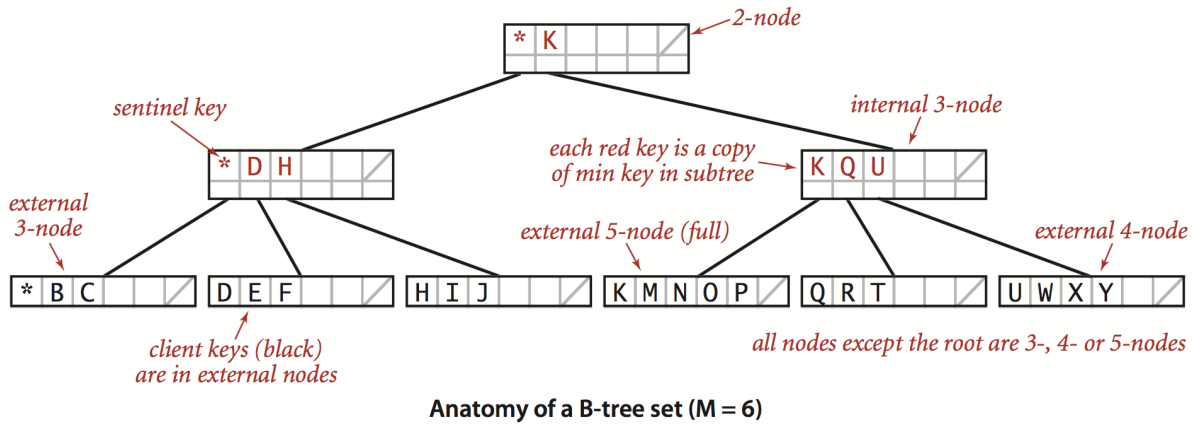


Fig. A.9: B-Tree

A.3.2 Operations

A.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.

A.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.

A.4 AVL Tree

TODO

RB-Tree is preferred since shorter implementation code.

A.5 Cartesian Tree

A.5.1 Basics

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

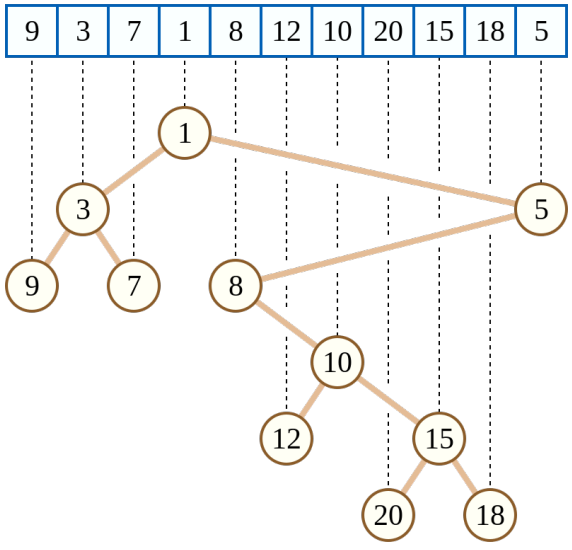


Fig. A.10: Cartesian Tree

```
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 9.3.1 Mono Stack.

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 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.

A.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.

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.

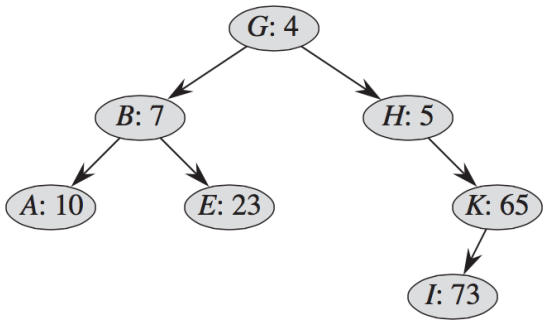


Fig. A.11: Treap. Each node x is labeled with x.key: x.priority.

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

Appendix B
General

B.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.

Appendix C
Divide & Conquer

C.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.

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

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