



VRIJE
UNIVERSITEIT
BRUSSEL



ML Algorithm Interview Preparation 2020

INTERVIEW PREPARATION

LiangLiang ZHENG

Academic Year: 2019-2020

Promotor:

Advisor:

Set a faculty using \faculty{\textit{Engineering Sciences}}

Contents

1 Machine Learning & Statistics	1
1.1 Linear Regression	1
1.1.1 Probabilistic interpretation	2
1.1.2 K-fold CV & Stratified K-fold CV	3
1.2 Accuracy, Precision, Recall	3
1.3 Statistical Tests Other	3
1.3.1 Null Hypothesis	3
1.3.2 Correlational Test	4
1.3.3 Comparison of Means test	4
1.3.4 Non-parametric Test	4
1.4 ROC	4
1.5 Cross-Validation	4
2 Natural Language Processing	5
2.1 Language Model	5
2.1.1 Unigram, Bi-Gram, etc	5
2.1.2 What are word embeddings?	5
2.1.3 Context Window	6
2.1.4 Two Types of Embedding Models	7
2.1.4.1 CBOW	7
2.1.4.2 Skip-Gram	7
2.2 TF-IDF	7
3 Deep Learning	9
3.1 GNN	9
4 Math	11
5 Algorithm	13
5.1 Time Complexity and Space Complexity	13
5.2 General Algorithmic Knowledge	13
5.2.1 Base Conversion	13
5.2.2 Concatenate bits	13
5.2.3 Max Counter	13
5.2.4 alphabet changing	13
5.2.5 check valid parentheses	13
5.2.6 Time Complexity and Space Complexity	14
5.2.7 Bit operations, Mod and Pow	14
5.2.8 count frequency	14
5.2.9 python: indices even and odd sub list	14

5.2.10	python: even and odd	14
5.2.11	python: sort list of tuple	15
5.2.12	python: functools	15
5.3	Basic Data Structure	15
5.3.1	Coding Template	15
5.3.1.1	Binary Search	15
5.3.1.2	Tree	16
5.3.2	Array	16
5.3.3	Linked List	16
5.3.3.1	commonly used tricks	16
5.3.3.2	Reverse Linked List	18
5.3.3.3	Fast-slow pointer	21
5.3.3.4	Advanced problems, not frequently asked questions	23
5.3.4	Stack	24
5.3.5	HashMap	24
5.3.5.1	TwoSum	24
5.3.5.2	TwoSum: unique pairs	25
5.3.5.3	Prefix and hashmap	25
5.3.6	Queue	26
5.3.7	Deque	26
5.3.8	Tree	26
5.3.8.1	summary	26
5.3.8.2	Traversal	27
5.3.8.3	Binary Tree Pruning	35
5.3.8.4	DFS related questions	35
5.3.8.5	Construct Tree	38
5.3.8.6	LCA	38
5.3.8.7	Tree serialization and de-serialization	40
5.3.8.8	Binary Search Tree	40
5.3.8.9	Balance BST	42
5.4	Advanced Data Structure	42
5.4.1	Priority Queue	42
5.4.2	Trie	42
5.4.3	Segment Tree	42
5.4.4	Greedy	42
5.4.4.1	Job Scheduling Problem	42
5.4.5	Graph	43
5.4.6	Summary	43
5.4.6.1	Connected Component	43
5.5	Set	44
5.5.0.1	Union-Find (Disjoint-set)	44
5.6	Search	45
5.6.1	Backtrack	45

5.6.1.1	combination	46
5.6.1.2	subset	47
5.6.1.3	permutation	48
5.6.2	BFS	49
5.6.3	DFS	55
5.6.3.1	Sudoku, N-Queue	56
5.6.3.2	DFS+BFS	59
5.7	Binary Search	61
5.8	Dynamic Programming	69
5.8.1	Simple Example: Fibonacci	70
5.8.2	Coin Change	70
5.8.3	Longest Increasing Subsequence (LIS) $O(n^2)$	71
5.8.4	Maximum Subarray	73
5.8.5	01 Knapsack problem	76
5.8.6	Subset Knapsack problem	76
5.8.7	Complete Knapsack Problem	78
5.8.8	Edit Distance	79
5.8.9	Longest Common Subsequence(LCS)	80
5.8.10	Longest Palindromic Subsequence	80
5.8.11	Super Egg Drop	83
5.8.12	Burst Balloons	85
5.8.13	Stock Buying Selling Problem	87
5.8.14	House Robber	92
5.8.15	RegExp Match	94
5.8.16	Knuth-Morris-Pratt (KMP)	96
5.9	Two Pointers Sliding Window Prefix Sum	96
6	Tensorflow & PyTorch	99
7	Spark	101
7.1	Rationale of using Spark	101
7.2	Core Concept and How Spark Work	102
7.2.1	Spark Component	103
7.2.2	Spark Architecture	104
7.3	Spark Programming	105
7.3.1	Difference between SparkSession and SparkContext	105
8	Docker & Kubernetes	107
8.1	Useful commands	108
8.2	Docker compose	109
8.3	Docker workflow & Develop with docker	110
8.4	Dockerfile	111
8.5	Private Docker Repository	112
8.6	Deploy and persistence	112

9 Git	113
9.1 Git configuration	113
9.2 Basic Commands	114
9.3 Git branch	115
9.4 Git tag	115
9.5 Git reset	116
9.6 Git workflow	116
9.6.1 Git Flow Step	118
10 SQL	123
11 DevOps	125
11.0.1 Basic Intro	125

Chapter 1

Machine Learning & Statistics

1.1 Linear Regression

The difference between **classification** and **regression** problem is depending on the target variable, if the target variable is continuous, we call it regression, otherwise, it is a classification problem.

To perform a supervised learning, we need to formulate an hypotheses h which approximate our real \mathbf{y} . A simple linear regression of the h is formulated as followed:

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \quad (1.1.1)$$

Here, the θ_i 's are the parameters (also called weights) parameterizing the space of linear functions mapping from \mathcal{X} to \mathcal{Y} . When there is no risk of confusion, we will drop the θ subscript in $h_{\theta}(x)$, and write it more simply as $h(x)$. To simplify our notation, we also introduce the convention of letting $x_0 = 1$ (this is the intercept term).

The cost function is defined as followed, this is call least square cost function, the choice of using least square comes from the fact that it'll be easier to calculate the derivatives of the cost function.

$$J(\theta) = \frac{1}{2} \sum_{i=1}^n \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2 \quad (1.1.2)$$

What we need to computer to do is to minimize $J(\theta)$ as small as possible, we use **Gradient Descent**, what gradient descent dose is starts with an 'initial guess', and and that repeatedly changes to make $J(\theta)$ smaller. α is the learning rate, the reason that is minus instead of add the partial derivative of (θ) is that, you could think of (θ) as the graph of x^2 , you start with a random x , every step you're going downward with the direction of the slope of x^2 .

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \quad (1.1.3)$$

The break down of the partial derivative of $J(\theta)$ is as followed:

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_{\theta}(x) - y)^2 \\
&= 2 \cdot \frac{1}{2} (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_{\theta}(x) - y) \\
&= (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^d \theta_i x_i - y \right) \\
&= (h_{\theta}(x) - y) x_j
\end{aligned} \tag{1.1.4}$$

As you can see from above, when calculating the partial derivatives of $J(\theta)$ wrt θ_j , the least square actually brings some convenience, in the end what we get, intuitively speaking, is the error between hypothesis h and y with respect to the current x we're looking at. Instead of looking at each training set each time of the iteration, **Batch Gradient Descent** looks at every example in the entire training set on every step.

1.1.1 Probabilistic interpretation

In the above of examples we have known that the form of the cost function (least square) could come from that fact it'll be easier for us to calculate the partial derivatives in the gradient descent phase. Another reason could be derived from the probabilistic perspective.

The relationship of the target hypothesis and real value y could be formed as

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)}$ is an error term that captures either unmodeled effects, or random noise. Let us further assume that the $\epsilon^{(i)}$ are distributed IID (independently and identically distributed) according to a Gaussian distribution (also called a Normal distribution) with mean zero and some variance σ^2 . We can write this assumption as " $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ ". I.e., the density of $\epsilon^{(i)}$ is given by

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\epsilon^{(i)})^2}{2\sigma^2}\right).$$

This implies that

$$p(y^{(i)} | x^{(i)}; \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$

When we wish to explicitly view this as a function of θ , we will instead call it the likelihood function:

$$L(\theta) = L(\theta; X, \vec{y}) = p(\vec{y} | X; \theta) \tag{1.1.5}$$

The MLE (Maximization of Likelihood Estimation) is a method we used to do the parameter estimation, here we have the parameter theta to be estimated, in order to

maximize the likelihood, normally we took the log of this likelihood function (because likelihood function involves tons of probability products, using log form to transform it to summation form), so we have:

$$\begin{aligned}
 L(\theta) &= \prod_{i=1}^n p(y^{(i)} | x^{(i)}; \theta) \\
 &= \prod_{i=1}^n \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\
 \ell(\theta) &= \log L(\theta) \\
 &= \log \prod_{i=1}^n \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\
 &= \sum_{i=1}^n \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\
 &= n \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2.
 \end{aligned} \tag{1.1.6}$$

Hence, maximizing $\ell(\theta)$ gives the same answer as minimizing

$$\frac{1}{2} \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2$$

1.1.2 K-fold CV & Stratified K-fold CV

1.2 Accuracy, Precision, Recall

1.3 Statistical Tests Other

1.3.1 Null Hypothesis

Given the test scores of two random samples, one of men and one of women, does one group differ from the other? A possible null hypothesis is that the mean male score is the same as the mean female score: $H_0 : \mu_1 = \mu_2$ where

$$\begin{aligned}
 H_0 &= \text{the null hypothesis,} \\
 \mu_1 &= \text{the mean of population 1, and} \\
 \mu_2 &= \text{the mean of population 2.}
 \end{aligned}$$

A stronger null hypothesis is that the two samples are drawn from the same population, such that the variances and shapes of the distributions are also equal.

1.3.2 Correlational Test

- Pearson correlation: Tests for the strength of the association between two **continuous variables**.
- Spearman correlation: Tests for the strength of the association between two **ordinal variables** (does not rely on the assumption of normal distributed data).
- Chi-square: Tests for the strength of the association between two **categorical variables**.

1.3.3 Comparison of Means test

T-Test Assumptions

- The first assumption made regarding t-tests concerns the scale of measurement. The assumption for a t-test is that the scale of measurement applied to the data collected follows a continuous or ordinal scale, such as the scores for an IQ test.
- The second assumption made is that of a simple random sample, that the data is collected from a representative, randomly selected portion of the total population.
- The third assumption is the data, when plotted, results in a normal distribution, bell-shaped distribution curve.
- The final assumption is the homogeneity of variance. Homogeneous, or equal, variance exists when the standard deviations of samples are approximately equal.

Paired T-test: Tests for difference between two related variables. Independent T-test: Tests for difference between two independent variables. ANOVA: Tests the difference between group means after any other variance in the outcome variable is accounted for.

1.3.4 Non-parametric Test

Wilcoxon rank-sum test: Tests for difference between two independent variables - takes into account magnitude and direction of difference. **Wilcoxon sign-rank test:** Tests for difference between two related variables - takes into account magnitude and direction of difference. **Sign test:** Tests if two related variables are different – ignores magnitude of change, only takes into account direction.

1.4 ROC

1.5 Cross-Validation

Chapter 2

Natural Language Processing

2.1 Language Model

2.1.1 Unigram, Bi-Gram, etc

We need a model that will assign a probability to a sequence of tokens. Let us start with an example: "The cat jumped over the puddle." A good language model will give this sentence a high probability because this is a completely valid sentence, syntactically and semantically. Similarly, the sentence "stock boil fish is toy" should have a very low probability because it makes no sense. Mathematically, we can call this probability on any given sequence of n words:

$$P(w_1, w_2, \dots, w_n)$$

We can take the unary language model approach and break apart this probability by assuming the word occurrences are completely independent:

$$P(w_1, w_2, \dots, w_n) = \prod_{i=1}^n P(w_i)$$

However, we know this is a bit ludicrous because we know the next word is highly contingent upon the previous sequence of words. And the silly sentence example might actually score highly. So perhaps we let the probability of the sequence depend on the pairwise probability of a word in the sequence and the word next to it. We call this the bigram model and represent it as:

$$P(w_1, w_2, \dots, w_n) = \prod_{i=2}^n P(w_i | w_{i-1}) \quad (2.1.1)$$

2.1.2 What are word embeddings?

In their most basic form, word embeddings are a technique for identifying similarities between words in a corpus by using some type of model to predict the co-occurrence of

words within a small chunk of text. By examining the adjacency of words in this space, word embedding models can complete analogies such as “Man is to woman as king is to queen.”

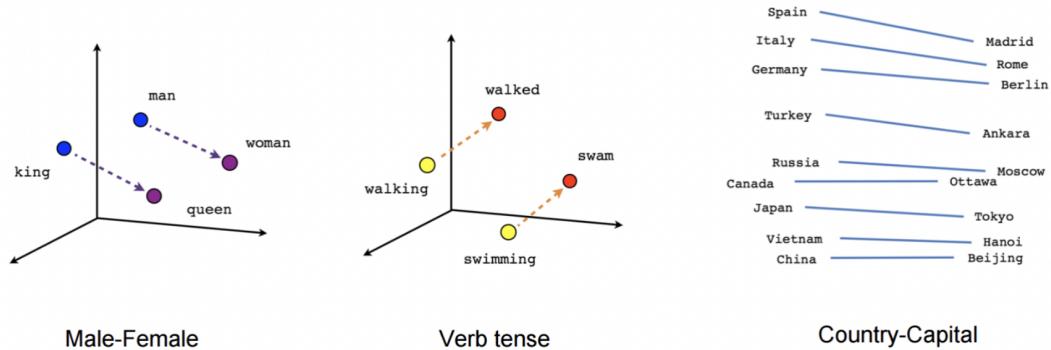


Figure 2.1: word embedding

2.1.3 Context Window

Word embeddings are created by identifying the words that occur within something called a “Context Window.” The Figure below illustrates context windows of varied length for a single sentence. The context window is defined by a string of words before and after a focal or “center” word that will be used to train a word embedding model. Each center word and context words can be represented as a vector of numbers that describe the presence or absence of unique words within a dataset, which is perhaps why word embedding models are often described as “word vector” models, or “word2vec” models.

- : Center Word
- : Context Word

- c=0 The cute **cat** jumps over the lazy dog.
- c=1 The **cute** **cat** jumps over the lazy dog.
- c=2 The **cute** **cat** jumps over the **lazy** dog.

Figure 2.2: context window

2.1.4 Two Types of Embedding Models

Word embeddings are usually performed in one of two ways: “Continuous Bag of Words” (CBOW) or a “Skip-Gram Model.” The figure below illustrates the differences between the two models. The CBOW model reads in the context window words and tries to **predict the most likely center word**. The Skip-Gram Model **predicts the context words given the center word**. The examples above were created using the Skip-Gram model, which is perhaps most useful for people who want to identify patterns within texts to represent them in multidimensional space, whereas the CBOW model is more useful in practical applications such as predictive web search.

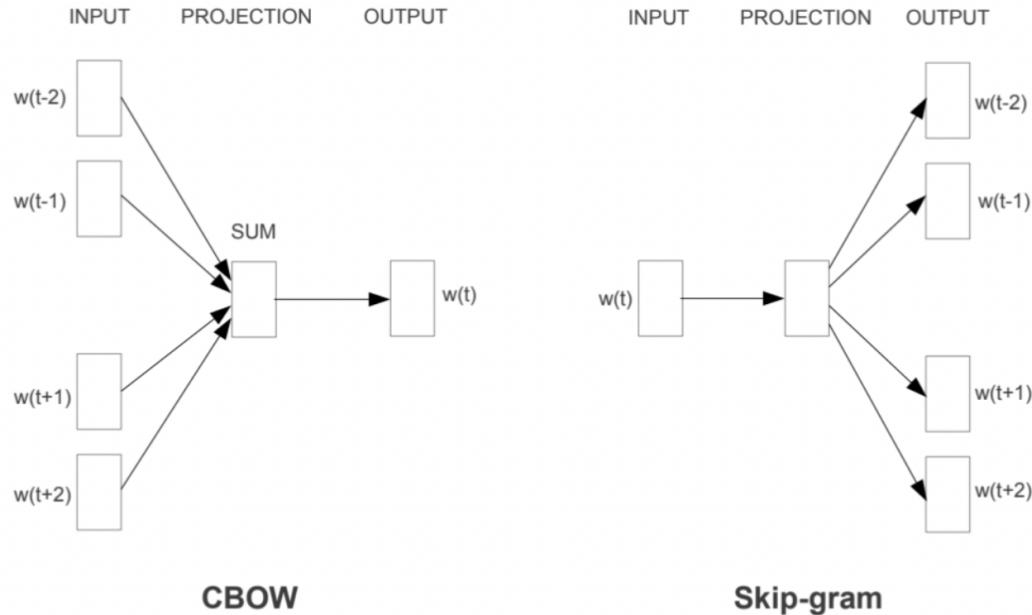


Figure 2.3: CBOW v.s. Skip Gram

2.1.4.1 CBOW

2.1.4.2 Skip-Gram

2.2 TF-IDF

Chapter 3

Deep Learning

3.1 GNN

Representation Learning: Automatically extract or learn the feature, the idea is to map d-dimensional embeddings such that similar nodes in the network are embedded close together.

Chapter 4

Math

Chapter 5

Algorithm

5.1 Time Complexity and Space Complexity

5.2 General Algorithmic Knowledge

5.2.1 Base Conversion

- Binary of a number: `bin(a)`
- Binary to `int(res, 2)`

5.2.2 Concatenate bits

The verticle line | in python is "bitwise or" method, it'll return 1 if one of it is 1.

`s << i.bit_length()|i`

5.2.3 Max Counter

```
l = len(nums)
a = collections.Counter(nums)
return a.most_common(1)[0][0]
```

5.2.4 alphabet changing

To know if a character is a alphabet, `'a'.isalpha()`

5.2.5 check valid parentheses

Check if a sequence of parentheses are valid, count the left paren first and if a right paren occurs and the left paren is 0, which means before this right paren there must be no left paren that could be corresponded to it. The code below is the code, which will be used in #301.

```

def valid(s):
    l = 0
    r = 0
    for i in s:
        if i == '(':
            l += 1
        if i == ')' and l == 0:
            r += 1
        elif i == ')':
            l -= 1
    return l == 0 and r == 0

```

5.2.6 Time Complexity and Space Complexity

5.2.7 Bit operations, Mod and Pow

Alphabet case conversion XOR operation with ' ', for example, converting 'd' to 'D' or do it reversely, simply do $\text{chr}(\text{ord}('d') \oplus \text{ord}(''))$

In #952, mod is always used in circle like number like circular list. A circle like number 0-9 where $0 - 1 = 9$ and $9 + 1 = 0$, the easiest way to know what number to mod could produce 0, in this case only 10 could do that.

5.2.8 count frequency

```

self.count = collections.Counter()
self.inorder(root)
freq = max(self.count.values())
res = []
for item, c in self.count.items():
    if c == freq:
        res.append(item)
return res

```

5.2.9 python: indices even and odd sub list

```

even = nums[::2]
odd = nums[1::2]

```

5.2.10 python: even and odd

The formular is $a \% b = a \& (b - 1)$

```

even = a % 2 == 0 or even = a & 1 == 0
odd = a % 2 == 1 or odd = a & 1 == 1

```

5.2.11 python: sort list of tuple

Sort by a specific element in the tuple

```
courses.sort(key=lambda x:x[1])
```

5.2.12 python: functools

- reduce:
- combination:
- permutation:

5.3 Basic Data Structure

5.3.1 Coding Template

5.3.1.1 Binary Search

```
def left_bound(nums, target):
    l = 0
    r = len(nums)
    while l < r:
        mid = l + (r - 1) // 2
        if nums[mid] >= target:
            r = mid
        elif nums[mid] < target:
            l = mid + 1

    return l

def right_bound(nums, target):
    l = 0
    r = len(nums)
    while l < r:
        mid = l + (r - 1) // 2
        if nums[mid] <= target:
            l = mid + 1
        else:
            r = mid

    return l - 1
```

5.3.1.2 Tree

inorder traversal: recursion
inorder traversal: iterative
pre-order and postorder: recursion
pre-order and postorder: iterative
level-order iterative

dfs build graph: template for turning a tree to a non-directed graph

```
graph = collections.defaultdict(list)
def dfs(parent, child):
    if parent and child:
        graph[parent.val].append(child.val)
        graph[child.val].append(parent.val)

        if child.left: dfs(child, child.left)
        if child.right: dfs(child, child.right)
dfs(None, root)
```

5.3.2 Array

5.3.3 Linked List

5.3.3.1 commonly used tricks

- dummy as previous node of head (#2, #445, #24)
- if not head or not head.next (#206, #24)
- fast-slow pointers (#141, #142, #876)

Example: In **Leetcode#2**, after transferring list to number using the function **l2n**, the result has been transferred to a string, in this way every element inside can be gotten simply by using index number, and in the end the list can be built in a iterative way. A dummy is initialized as head and will be used as return pointer because in iteration head is already changed. For loop is backward, From len - 2 (len - 1 should be assigned to head before) to the end 0.

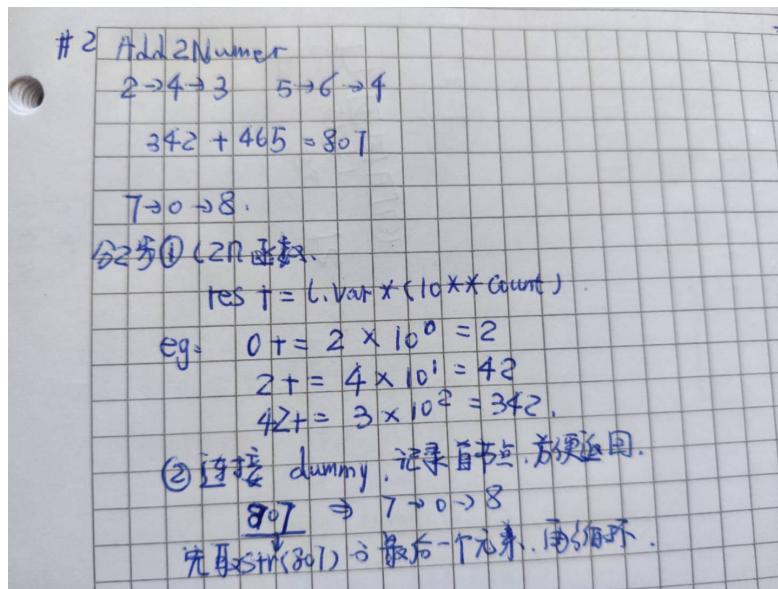


Figure 5.1: two sum

```

class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        def l2n(l):
            res = 0
            count = 0
            while l:
                res += l.val * (10 ** count)
                l = l.next
                count = count + 1

            return res

        str_add = str(l2n(l1) + l2n(l2))
        head = ListNode(str_add[len(str_add) - 1])

        dummy = head

        for i in range(len(str_add) - 2, -1, -1):
            node = ListNode(str_add[i])
    
```

```

head.next = node
head = head.next

return dummy

```

Leetcode#445, add two number II, the only difference is that the function **l2n** is difference. $res = res * 10 + l.val$, and in for loop in index i is forward, from 1 (0 should be assigned to head before) to the end $len - 1$.

5.3.3.2 Reverse Linked List

Leetcode#206 is to reverse linked list, there are iterative way and recursive way to reverse the linked list. The first one below is **iterative** way, the important message is that whenever you want to assign a pointer.next to other node, don't forget to put pointer.next to a new variable in this way we won't lose it. In this problem, it's important to know whether head is None or not, we also need to know head.next for the reason that if this list only has 1 node.

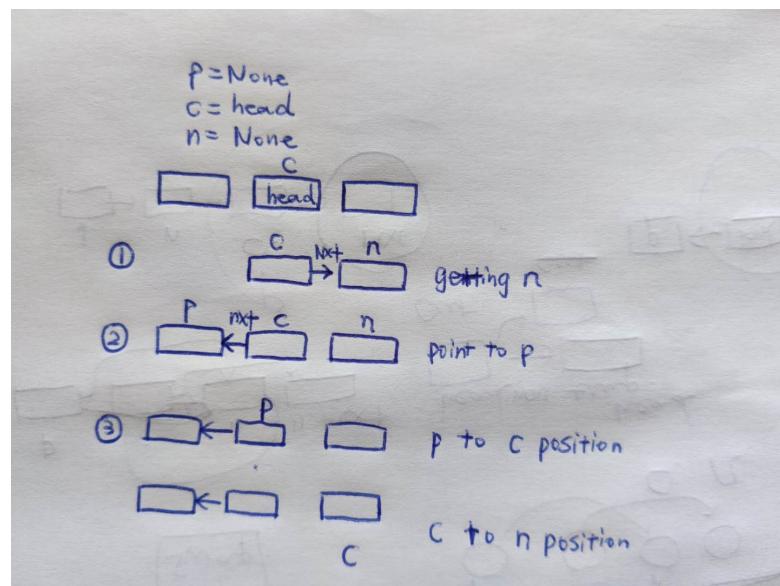


Figure 5.2: reverse list in iterative way

```

class Solution(object):
    def reverseList(self, head):
        if not head or not head.next:
            return head

        p = None
        c = head

```

```

n = None

while c:
    n = c.next
    c.next = p
    p = c
    c = n

return p

```

For **recursive** way, reverseListBlog explained very clearly, the Figure shown below also illustrated the process of reverse a linked list in a recursive way.

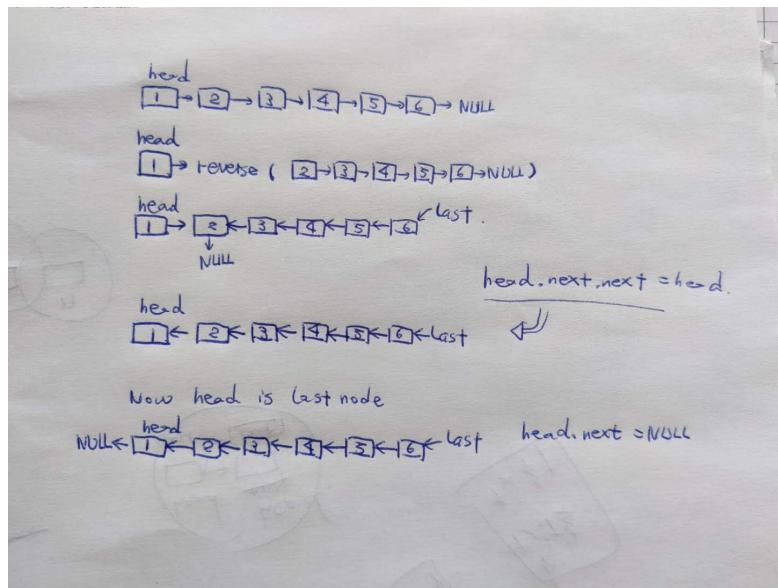


Figure 5.3: reverse list in recursive way

```

def reverseList(self, head):
    """
    :type head: ListNode
    :rtype: ListNode
    """
    if not head or not head.next:
        return head

    last = self.reverseList(head.next)
    head.next.next = head
    head.next = None

```

```
    return last
```

If we want to reverse the first N node, it is similar to the implementation of the recursive one, only need to know the successor, the successor is the one right after node $N + 1$, for example, 1-2-3-4-5-6-None, will return as 3-2-1 4-5-6, but in this case 1.next is not None anymore, it should be 4, so in recursive we need to know the node and assign that in a variable **successor**. The next is the Figure of Reverse N node, and code below the figure is the **Leetcode#92**

```
class Solution(object):
    successor = None
    def reverseBetween(self, head, m, n):
        def reverseN(head, n):
            global successor
            if n == 1:
                successor = head.next
                return head

            last = reverseN(head.next, n - 1)
            head.next.next = head
            head.next = successor

            return last

        if m == 1:
            # same as reverse first N node
            return reverseN(head, n)

        # forward as to the base case m = 1
        head.next = self.reverseBetween(head.next, m - 1, n - 1)
        return head
```

For **Leetcode#24**, swap nodes between nodes should could be solved with iterative way using 2 pointers n1 and n2, the graph and code is shown below.

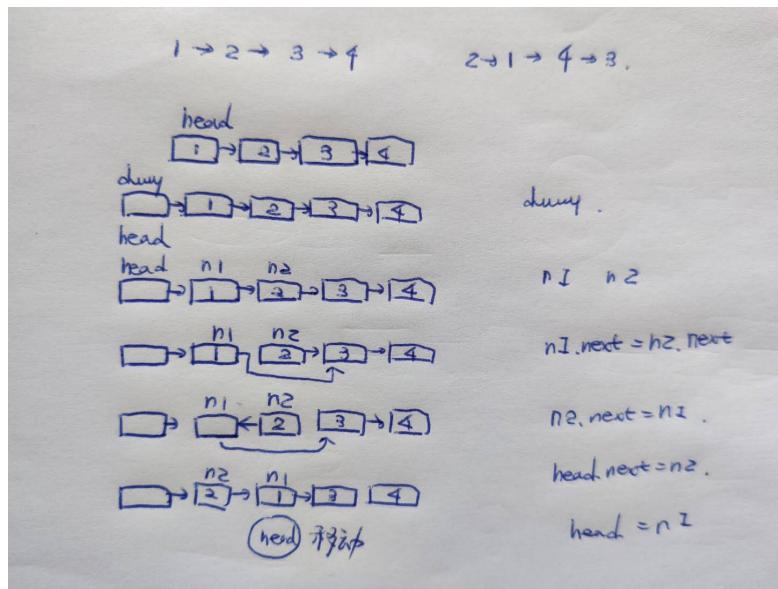


Figure 5.4: swap 2 nodes

```

def swapPairs(self, head):
    if not head or not head.next:
        return head
    dummy = ListNode(-1)
    dummy.next = head
    head = dummy

    while head.next and head.next.next:
        n1,n2 = head.next, head.next.next
        n1.next = n2.next
        n2.next = n1
        head.next = n2
        head = n1

    return dummy.next

```

5.3.3.3 Fast-slow pointer

Leetcode#141 Fast-slow pointer always used in check if a linked list has cycle, the fast pointer will go 2 step at a time while the slow pointer will go 1 step at a time. The initial step is to assign head to both of them, in while loop, the condition is while(fast.next), this is because if reach the second last node, fast.next is the last one, for the iteration after this will end.

```

def hasCycle(self, head):
    slow = head
    fast = head

    while fast and fast.next:
        fast = fast.next.next
        slow = slow.next

    if fast == slow:
        return True

    return False

```

Leetcode#142, the only difference is to know the starting node of the cycle, the following graph shows after k step, slow and fast meet each other in the Meeting Point, the starting point should before m step of Meeting Point, after putting slow back to head, let slow and fast go 1 step at a time, after k - m step, they will meet at the Starting point. (Remeber there are 2 ways to go out of the while loop, the first one is break, the second one is normal running, then in that case there is no Cycle inside, we shoud return None instead).

```

def detectCycle(self, head):
    slow = head
    fast = head

    while fast and fast.next:
        fast = fast.next.next
        slow = slow.next

    if slow == fast:
        break
    else:#No Cycle (out of loop not because of break)
        return None

    slow = head
    while slow != fast:
        slow = slow.next
        fast = fast.next

    return fast

```

Leetcode#876 is an easy one, same as above, when fast reaches the end, slow will be the middle node.

5.3.3.4 Advanced problems, not frequently asked questions

Leetcode#25 could be solved using reverseBetween, 2 passes in solving this, first is to count the length of the list, and the second is to iteratively call reverseBetween with corresponding index m and n. If the k is 2, then it is similar to **Leetcode#24**

```

def reverseKGroup(self, head, k):
    if (head is None or k == 1):
        return head

    # First, get the length of the list
    length = self.getCount(head)

    # Second pass, swap in groups.
    newH = head
    for i in range(1, length+1, k):
        if i+k-1 <= length:
            newH = self.reverseBetween(newH, i, i+k-1)

    return newH

def getCount(self, node):
    n = node
    count = 0
    while n:
        count+=1
        n = n.next
    return count

def reverseN(self, head, n):
    global suc
    if n == 1:
        suc = head.next
        return head

    last = self.reverseN(head.next, n-1)
    head.next.next = head
    head.next = suc
    return last

def reverseBetween(self, head, m, n):
    if m == 1:
        return self.reverseN(head, n)

```

```

head.next = self.reverseBetween(head.next, m - 1, n - 1)

return head

```

5.3.4 Stack

5.3.5 HashMap

HashMap is always used for the problem set of counting given an array and a subjection condition.

5.3.5.1 TwoSum

A classical problem set is Two Sum, to output a pair (it could contains more than 1 pair) where it satisfies $f(x, y) = z$. A brute force way could be to iterate the array in $O(n^2)$, this will be TLE. An alternative way is to use hashmap to mark down the number and index that appear before. And use target - n as an key to search in the hashmap, in this way we could solve it in only 1 loop. $O(n)$.

An example of the process could be found below in 5.5:

方法 2: 查找表法

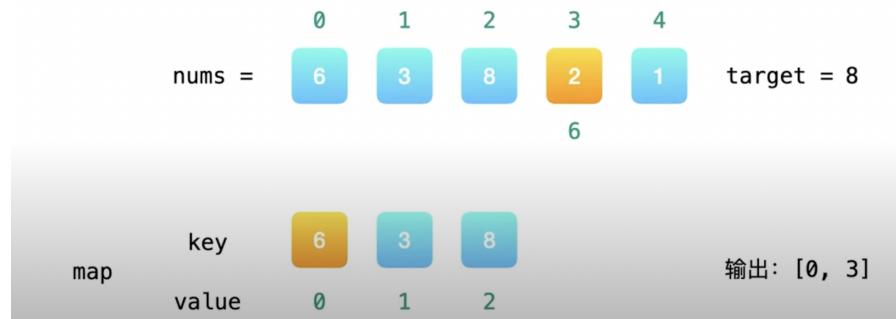


Figure 5.5: twosum

And the corresponding code:

```

def twoSum(self, nums: List[int], target: int) -> List[int]:
    d = {}

    for idx, n in enumerate(nums):
        if (target - n) in d.keys():
            return [d[target - n], idx]
        else:
            d[n] = idx
    return [-1, -1]

```

5.3.5.2 TwoSum: unique pairs

This is a variant of TwoSum problem set and it requires to return all of the unique pairs that satisfy the condition. An intuitive way is to use a set and return the length of it.

```

res = {}
out = set()

for index, value in enumerate(nums):
    if target - value in res:
        out.add((value, target-value))
    else:
        res[value]=index

return len(out)

```

5.3.5.3 Prefix and hashmap

Prefix sum and hashmap could track the sum of sub arrays. See 1524. Number of Subarrays With Odd Sum . Out of all its sub arrays, to find how many of them sums up to Odd. Again like the previous sections, if we want to use brute force, we need to use at least $O(n^2)$ to get the result, so we need to think about using space to compensate the time. First of all, subarrays are continuous, Here we use prefix sum to tackle this problem, concretely.

The important use scenario of prefix is that it could get any range sum in the given array, for example, if $P[i] = A[0] + A[1] + \dots + A[i - 1]$, and $P[j] = A[0] + \dots + A[j - 1]$, $i \leq j$, then sum of $A[i : j]$, $S[i : j] = P[j] - P[i]$. The way is to use hashmap, key will be the prefix sum, value will be the number of counting of the prefix sum.

The second part is the tricky part, it differs for different questions, but easy to be understood. In leetcode 523, we need to see if there sum and number of element in the sub array is larger than 2, so in the dictionary, the value should be index. But in leetcode 1524, it needs to track the number of the sum to odd, then the value of the dict should be number of times it sums to odds.

The third part is to understand what should be the index. In leetcode 523, we could know $\text{sum1} - \text{sum2} = \% \text{target}$, also means $\text{sum1}\%k == \text{sum2}\%k$. Which give us an idea of something like target - value key value find mechanism in twoSum, then we could get the result. And almost all of the prefix sum combined with hashmap problem, $d[0]$ is a special case, if its for sum, we could use collections.defaultdict(int) and it will be initialised as 0 (leetcode 1524), but for leetcode 523, we need to store index instead of count, then we need to initialise as $d = 0 : -1$.

```

def subarraysDivByK(self, nums, k):
    """
    :type nums: List[int]
    :type k: int

```

```

:rtype: int
"""

prefix_sum = nums[::]
for i in range(1, len(nums)):
    prefix_sum[i] += prefix_sum[i-1]

d = collections.defaultdict(int)
d[0] = 1
res = 0
for i in prefix_sum:
    res += d[i%k]
    d[i%k] += 1

return res

```

5.3.6 Queue

5.3.7 Deque

5.3.8 Tree

5.3.8.1 summary

- stack for doing the iterative form of traversal, the pattern of thinking should be reverse. Specifically, for inorder traversal, left-root-right, find left most first, but on the way to find left, stacking root also, since after getting left most, pop out order is left -> root. so stack.append(root) should be applied before finding left most node.**Memorization Trick:Y true not stack**
- For pre-order, root-left-right, unlike pop should be right before stack.append, since the stacking is reverse, stacking right first and then left branch. For postorder, the code is similar, left-right-root, we do it in a reverse way root-right-left, then the question becomes similar to pre-order, after getting res list, reverse it and then return.**Memorization Trick:Y stack**
- For Level-order(BFS), its better to use collections.deque(), which will save more time, the list.pop(0) have complexity of O(n).
- construct trees
- Getting the depth of a tree, normally using recursion, get left and right and consider returning a new value, which normally will be the max(left, right)+1 or other kinds of form. If there are other restriction in the tree depth, like finding the smallest, or finding compare the tree depth, then we to do something after getting and l and r.

- There are several problem set use both children but only return one child when doing the recursion, for example, #124, The definition of the path is **a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections**, means it could go through only left or right, but in the description Example, it shows that add both children only if the starting node of this path is their direct parent. If this is not the case, it can only return one of its children. So in the coding part. the trick is **res can append both children but only return one**. for #534 and #687 are also the same.
- PathSUM problem
- Go through the root or dose not go through the root? From root to the leaf, should give condition on the

Different kind of tree

- Binary Search Tree: the inorder traversal is sorted. Those 2 algorithms are bonded tightly
-

5.3.8.2 Traversal

Leetcode #94 is a classic **inorder** traversal of tree question, the following is the template of recursively do the inorder traversal.

```
def inorderTraversal(self, root):
    # recursive way
    res = []

    def inorder(root):
        # base case
        if not root:
            return root

        inorder(root.left)
        res.append(root.val)
        inorder(root.right)

    inorder(root)
    return res
```

The recursive way is shown below, the intuition behind the algorithm is that we want to get the leftmost child first in the res list, using stack here to push the previous nodes inside until we find the first leftmost child (means it has no left child anymore). **Leetcode #173** is also using iterative way to inorderly traverse a Binary Search Tree.

```

def inorderTraversal(self, root):
    # iterative way
    res = [] # the result oredr
    stack = [] #

    while True:
        while root:
            # find the left most
            stack.append(root)
            root = root.left
        # if there was no element inside stack, similar to if not root in recursive
        if not stack:
            return res

        # found the left most child
        left = stack.pop()
        res.append(left.val)
        root = left.right
    
```

Leetcode #589, Leetcode#114 is the preorder and Leetcode #590 is the postorder.

```

def preorder(self, root):
    # recursive way
    res = []

    def preorder1(root):
        if not root:
            return

        res.append(root.val)

        for c in root.children:
            preorder1(c)

    preorder1(root)
    return res
    
```

The following is the answer for #114, since the output should be a linked list tree, then the res.append should append input TreeNode instead of root.val, and assign the next element inside the list to the right node and the left node to None.

```

def flatten(self, root):
    # preorder recursive way
    res = []
    
```

```

def preorder(root):
    if not root:
        return root

    res.append(root)
    preorder(root.left)
    preorder(root.right)

preorder(root)

for i in range(len(res) - 1):
    res[i].left = None
    res[i].right = res[i+1]

return res

```

And the iterative way is as followed, after `res.append`, we need to reverse the way of the output, specifically it means that stack the one that needs to output later first, in this case it's the last child of the children, so we could simply use `node.children[::-1]` in this case, and use extend to get rid of the square braces problems.

```

def preorder(self, root):
    # iterative way
    res = []

    if root == None:
        return res

    stack = []
    stack.append(root)

    while stack:
        node = stack.pop()
        if not node:
            continue
        # reverse the way to output, so stack in right first
        res.append(node.val)
        # stack.append(node.right)
        # stack.append(node.left)
        stack.extend(node.children[::-1])
    return res

```

The following is the graph shows how it works in general

The serialization of the preorder is also a problem set, for example in **leetcode #331**, the form of the preorder is always follows number # #.

```
def isValidSerialization(self, preorder):
    #the leaves will always follow the form of number # #
    # only need to replace the last 3 element to # and iterate the list

    stack = collections.deque()
    pre = preorder.split(",")

    for i in pre:
        print(i)
        stack.append(i)

    while len(stack) >= 3 and stack[-1] == stack[-2] == '#' and stack[-3] != '#':
        stack.pop()
        stack.pop()
        stack.pop()
        stack.append('#')

    return len(stack) == 1 and stack.pop() == '#'
```

For postorder, the recursive way is the same as inorder and preorder except the order of append root.val, the following is the solution for **Leetcode #590**. The form is similar to the iterative form of preorder, but stacking node.left and then node.right, and reverse all the res list.

```
def postorder(self, root):
    # iterative way
    res = []
    if root == None:
        return res
    stack = []
    stack.append(root)
    while stack:
        node = stack.pop()
        # if not node:
        #     # continue
        res.append(node.val)
        stack.extend(node.children[:])
```

```

    res.reverse()
    return res

```

recursive way for implementing the traversal is actually important in this section, since they are basic framework (or say template) to solve other kinds of problem.

```

def recursive(root):
    if not root:
        return
    # do what you want to do with root
    recursive(root.left)
    recursive(root.right)

```

In **Leetcode #100**, in order to know whether 2 tree are the same, in the recursion, first take out their root and compare. The recursion is also something will be covered in DFS, where the fist line in the recursion are always those end case. In sameTree, the end condition is when 2 nodes are None, if not, then check if one of them is empty. if the structure are the same, the final condition is whether the values are the same. This ensemble the war defense, soldiers set up layer to defend enermies.

```

def isSameTree(self, p, q):
    # both of them are None, only one case with output True
    if p is None and q is None:
        return True
    #one of it is None
    if p is None or q is None:
        return False
    if p.val != q.val:
        return False

    return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)

```

For **Leetcode #101**, same as isSameTree, only difference is change recursion to opposite node. self.isSameTree(p.left, q.right) and self.isSameTree(p.right, q.left)

For **Leetcode #104** finding the maximum depth of the tree, simply three lines could solve it. Same pattern as the traversal recursion. When root is None, return 0, otherwise recursively call the max (left and right) + 1.

```

def maxDepth(self, root):
    if not root:
        return 0
    return max(self.maxDepth(root.left), self.maxDepth(root.right)) + 1

```

For minimum (#111) depth of the tree, after recursively calling the function, there are 2 situations, which are drawn below, when it comes to leave, 2 nodes like the graph should be counted as 2, so $l + r + 1 = 2$, while the other condition is similar to max.

```

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

    l = self.minDepth(root.left)
    r = self.minDepth(root.right)
    # be careful! if there are only 2 nodes
    if l == 0 or r == 0:
        return l + r + 1
    else:
        return min(l, r) + 1

```

Same for #110, get height using maxDepth function as in #104 and add one condition $\text{abs}(l - r) \neq 1$ into the recursion call.

In the next subsection, we're going to discuss Level order traversal, this is also a sub pre-requisite for Search part BFS. The following code is the template for level order traversal(BFS).

```

def level_order_tree(root, result):
    if not root:
        return
    # using collections.deque()
    # avoid using list.pop(0) since the complexity is O(n)
    queue = collections.deque()
    queue.append(root)
    while queue:
        node = queue.popleft()
        # do somethings
        result.append(node.val)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return result

```

In most of the leetcode problem set, we need to output according to layers, so inside the while, we need to have a for loop and a newQueue which could do it in different layer . #102 and #107 are using the template below.

```

def level_order_tree(root):
    if not root:
        return
    q = [root]
    while q:
        # normally res will be here

```

```

new_q = []
for node in q:
    # do somethins with this layer nodes eg: 103, set flag to append in zigzag fashion
    # classical condiction
    if node.left:
        new_q.append(node.left)
    if node.right:
        new_q.append(node.right)
    # replace as the new queue
    q = new_q
# return things you want
return xxx

```

#987 is a good problem set, it combines the level-order problem with coordinate questions. The first need to watch out is the form inside queue, should be (node, x, y), the order of the value and node is not matter, while inside res, we should put x and y in front since in the end of the while, res should be sorted in this way we'll could put those with the same coordinate in one cell from left to right. up to down, so y should be incremented instead of decremented as mentioned in the problem description.

```

def verticalTraversal(self, root):
    res = []
    if not root:
        return res

    q = []
    #initial coordinates
    q.append((root, 0, 0))

    while q:
        node, x, y = q.pop()
        res.append((x, y, node.val))
        # no need to do it in differen layer
        # so no for loop here
        if node.left:
            q.append((node.left, x-1,y+1))
        if node.right:
            q.append((node.right, x+1,y+1))

    res.sort()
    print(res)
    # put root inside the res1 first
    res1 = [[res[0][2]]]
    # res2 = []

```

```

        for i in range(1, len(res)):
            if res[i][0] == res[i - 1][0]:
                res1[-1].append(res[i][2])
            else:
                res1.append([res[i][2]])

        return res1

#116 and #117

def connect(self, root):
    """
    :type root: Node
    :rtype: Node
    """

    # Level-order way with iterative
    res = []
    if not root:
        return None

    q = collections.deque()
    q.append(root)

    while q:
        newQ = collections.deque()
        for i in range(len(q)):
            n = q.popleft()
            res.append(n.val)
            if n.left:
                newQ.append(n.left)
            if n.right:
                newQ.append(n.right)
        res.append('#')

    q = newQ

    a = Node(res[0])
    p = a
    for i in range(1, len(res)-1):
        if res != '#':
            b = Node(res[i])
        else:
            b = None
        a.next = b

```

```
a = b
return p
```

5.3.8.3 Binary Tree Pruning

Leetcode #814 is still a postorder, could use DFS(recursive way) to do it, since we need to know it from the leaves and then bottom up pruning the tree, the return value should be a node, the one to get the return value (condition) should be root.left and root.right. Trim it in the last step after all the conditions.

```
def pruneTree(self, root):
    if not root:
        return
    root.left = self.pruneTree(root.left)
    root.right = self.pruneTree(root.right)
    # only return node with 1 inside or not leaves
    return root if root.val == 1 or root.left or root.right else None
```

For **Leetcode #669**, trim it when compare to L and R and get left and right in the last condition.

```
def trimBST(self, root, L, R):
    if not root:
        return

    if root.val < L:
        return self.trimBST(root.right, L, R)

    elif root.val > R:
        return self.trimBST(root.left, L, R)

    else:
        root.left = self.trimBST(root.left, L, R)
        root.right = self.trimBST(root.right, L, R)
        return root
```

5.3.8.4 DFS related questions

A lot of problem sets could be solved in a recursive way. The followings are some problems that is typical recursion-based question. It is noticeable to know the nature of recursion, recursion is a structure of tree, every branch (or same iteration) will go through the deepest leave and if found no node after, then backtrack and return. So the first couple lines of DFS are always condition, the first line will also be the last condition, for example in tree the first condition is always to know whether this node is empty or

not to see if this recursion has reached the end of the tree, then next couple of conditions will be based on specific question set, the last is to recursively called the function

Leetcode #112 and **Leetcode #113** are among them, the difference of them is that in 112 there is only one answer while in 113 there are multiple answers, then we need a path + [root.left.val] inside the recursion, since we call the root.left.val which means we assume root.left is indeed exist, we can't wait until the next iteration to check whether root.left is existed, then in this round, it's also necessary to check if root.left is existed. The 2 templates below could be used in any path sum related questions.

```

def hasPathSum(self, root, s):
    if not root:
        return False

    if root.left == None and root.right == None and s == root.val:
        return True
    # either branch is okay, so OR is used here
    return self.hasPathSum(root.left, s - root.val) or self.hasPathSum(root.right, s - root.val)

def pathSum(self, root, s):
    res = []
    if not root:
        return res

    def helper(root, s, path):
        if not root:
            return

        if root.left == None and root.right == None and s == root.val:
            res.append(path)
            return

        if root.left:
            helper(root.left, s - root.val, path + [root.left.val])
        if root.right:
            helper(root.right, s - root.val, path + [root.right.val])

    helper(root, s, [root.val])
    return res

```

Leetcode#437 is also among them. But 437 has several answer and which may not from root to leave, which also need to recurse starting from every node below. then

there are 2 recursion running below. First is the helper which did as the same as what we need above, and called pathSum twice and recursively call root.left and root.right inorder to start with a new child node to search.

```
def pathSum(self, root, s):
    if not root:
        return 0
    return self.helper(root, s) + self.pathSum(root.left, s) + self.pathSum(root.right, s)

def helper(self, root, s):
    if not root:
        return 0
    res = 0
    if s == root.val:
        res += 1

    res += self.helper(root.left, s - root.val)
    res += self.helper(root.right, s - root.val)
    return res
```

The next part is MAX PathSUM related questions: Use both children, but return one, these kind of question is a little bit tricky, but it can also be solved using DFS way. For **Leetcode#124**, the graph has been shown below, we need to find the maximum path sum, the end condition is not go through the leaves, so we need no condition to stop until it reaches the end of the tree, but the tricky part is the result layer could be $\text{root.val}+l+r$ while in recursion, we can only choose one of the path(biggest path) to search. And in case of negative number, l and r should be compared to 0.

```
def maxPathSum(self, root):
    self.res = float('-inf')

def maxPath(root):
    if not root:
        return 0

    l = max(0, maxPath(root.left))
    r = max(0, maxPath(root.right))
    # final answer should add l + r + root.val
    self.res = max(l + r + root.val, self.res)
    # while this return will go the next layer, only need one path
    return max(l, r) + root.val

maxPath(root)
return self.res
```

leetcode #235 and **leetcode #236** is also part of these problems, like what we did in maxPath, divide(recursively called and get l and r first) and conquer (and then do something with the result l and r).

```
def lowestCommonAncestor(self, root, p, q):
    if any((not root,root == p, root ==q)):
        return root

    l = self.lowestCommonAncestor(root.left, p, q)
    r = self.lowestCommonAncestor(root.right, p, q)

    #means there are l and r with same root
    if l and r:
        return root
    # if not, return one of the branch which has and recursively call again
    else:
        return l if l else r
```

508

5.3.8.5 Construct Tree

The approach to construct tree is very simple, given the rule(BST, then mid is root)(max tree, build according to max in every recursive array), and slice the array for left branch and right branch inside the two recursive expression shown below.

```
def buildTree(self, preorder, inorder):
    if not preorder or not inorder:
        return None

    # first element in preorder is root, found the index in preorder and separate
    root = TreeNode(preorder[0])
    index = inorder.index(preorder[0])
    # length of left branch in preorder is 1(root) to index + 1,
    root.left = self.buildTree(preorder[1:index+1], inorder[:index])
    root.right = self.buildTree(preorder[index+1:], inorder[index+1:])
    return root
```

5.3.8.6 LCA

These kinds of problem sets are a little bit difficult. 236 and 235 is the LCA problem, the recursion thinking pattern is to get l and r first, if found it, then could be return. since the return value is the definition of the function. But this process is after the recursion, the condition that could find the result is before the recursion, when the root is None, means not root, then return root, if the root is equal to p or q(remember this is after the

recursion, root.left and root.right has been passed). if both returns a valid node which means p, q are in different subtrees, then root will be their LCA. if only one valid node returns, which means p, q are in the same subtree, return that valid node as their LCA.

```
def lowestCommonAncestor(self, root, p, q):
    if not root or q == root or p == root:
        return root

    l = self.lowestCommonAncestor(root.left, p, q)
    r = self.lowestCommonAncestor(root.right, p, q)

    if l and r:
        return root
    else:
        return l if l else r
```

Iterative Way:

```
def lowestCommonAncestor(self, root, p, q):
    pathP, pathQ = self.findpath(root, p), self.findpath(root, q)
    if pathP and pathQ:
        length = min(len(pathQ), len(pathP))
        ans = None
        for i in range(length):
            if pathP[i] != pathQ[i]:
                break
            ans = pathP[i]
        return ans
    def findpath(self, root, key):
        if not root:
            return
        stack = []
        lastVisit = None
        while stack or root:
            if root:
                stack.append(root)
                root = root.left
            else:
                tmp = stack[-1]
                if tmp.right and tmp.right != lastVisit:
                    root = tmp.right
                else:
                    if tmp.val == key.val:
                        return stack
                    else:
```

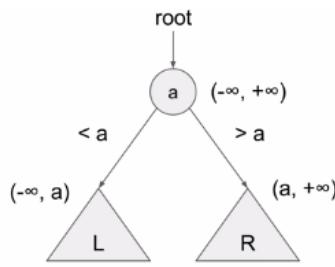
```
lastVisit = stack.pop()
root = None
```

5.3.8.7 Tree serialization and de-serialization

#652 #297

5.3.8.8 Binary Search Tree

#98 To validate a binary search tree, normally BST-related questions could be solved using recursion, so this problem could be solved in 2 ways. The first is give limit condition, condition is shown below.



Property of a binary search tree

- $\text{root.val} > \text{all vals in L}$
- $\text{root.val} < \text{all vals in R}$
- Inorder traversal, vals are sorted

Figure 5.6: Validate Binary Search Tree

```
def isValidBST(self, root):
    return self.isValid(root, float('inf'), float('-inf'))

def isValid(self, root, max, min):
    if not root:
        return True

    if root.val >= max or root.val <= min:
        return False

    return self.isValid(root.left, root.val, min) and \
           self.isValid(root.right, max, root.val)
```

The second way is to use inorder, the inorder traversal of BST is sorted, so just to know in the recursion if prev is greater or equal to root, then return False, it is also noticeable that the assignment of prev should be after the first recursion. class Solution(object): def isValidBST(self, root): self.prev = None
def inorder(root): if not root: return True

```

if not inorder(root.left): return False
if self.prev and self.prev.val != root.val: return False
self.prev = root
return inorder(root.right)
return inorder(root)
#701 combining the concept of binary search and build tree, very elegant solution.

```

```

def insertIntoBST(self, root, val):
    if not root:
        return TreeNode(val)
    if val > root.val:
        root.right = self.insertIntoBST(root.right, val)
    if val < root.val:
        root.left = self.insertIntoBST(root.left, val)
    return root

```

501 is a very good problem, to find the highest frequency element inside the Binary search tree, the first solution is to use collections.Counter() to do that, and there are possibilities that not only 1 highest frequency element, that is why the return should be a list.

```

def findMode(self, root):
    if not root: return []
    self.count = collections.Counter()
    self.inorder(root)
    freq = max(self.count.values())
    res = []
    for item, c in self.count.items():
        if c == freq:
            res.append(item)
    return res

def inorder(self, root):
    if not root: return
    self.inorder(root.left)
    self.count[root.val] += 1
    self.inorder(root.right)

```

#450,

- The deleted node has no left tree, return right tree
- The deleted node has no right tree, return left tree
- Has both left and right tree, to way to do it

- Search the min in right tree, swap the nodes and recursively delete the key again
- Search the max in the left tree, swap the nodes and recursively delete the key again

```

def deleteNode(self, root, key):
    if not root: return root
    if root.val == key:
        # if root is a leave
        if not root.right:
            return root.left
        else: # both left and right exist
            # set the min of right tree to be the new node
            right = root.right
            while right.left:
                right = right.left
            right, root.val = right.val, root.val

            root.left = self.deleteNode(root.left, key)

            root.right = self.deleteNode(root.right, key)

    return root

```

5.3.8.9 Balance BST

Given a un-balanced BST tree, transfer it back to a balanced BST tree where the difference between the deepest left tree and the deepest right tree is at most 1.

Given the properties that the inorder traversal of BST is a sorted array. We could first use a list to save the sorted elements. And then build tree based on BST. So 2 recursive function, the first one is the inorder, and the second one is the buildBST.

5.4 Advanced Data Structure

5.4.1 Priority Queue

5.4.2 Trie

5.4.3 Segment Tree

5.4.4 Greedy

5.4.4.1 Job Scheduling Problem

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time,

so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Sort the jobs based on their deadlines. Use priority queue to keep track of all the taken course, if the duration or time is larger than the deadline, then do a heappop.

Leetcode 630

```
def scheduleCourse(self, courses):
    """
    :type courses: List[List[int]]
    :rtype: int
    """

    import heapq
    courses.sort(key=lambda x:x[1])
    pq = []
    time = 0
    for duration, end in courses:
        heapq.heappush(pq, -duration)
        time += duration

        if time > end:
            time += heapq.heappop(pq)

    return len(pq)
```

5.4.5 Graph

5.4.6 Summary

Classical graph problems like BFS or DFS could be found in the chapter 5.6.

5.4.6.1 Connected Component

Connected component (graph theory), a set of vertices in a graph that are linked to each other by paths. In leetcode, there are some problems that ask about number of disconnected-connected component / connected components. We could use DFS to iterate every node and helper list **visit** to denote if a node is visted or not. Example is leetcode 323 (Locked problem).

```
def ConnectedComponent(self, n: int, c: List[List[int]]) -> int:
    # create graph
    graph = collections.defaultdict(set)
    visit = set()
    for u,v in c:
        graph[u].add(v)
        graph[v].add(u)
```

```

def dfs(g, v, i):
    # mark as visited
    v.add(i)
    if i in g:
        for j in g[i]:
            if j not in visit:
                dfs(g, v, j)

res = 0
for i in range(n):
    if i not in visit:
        # iterating over all nodes as start point
        dfs(graph, visit, i)
        res += 1

return res

```

This type of problem sets could also be solved by union find. See section Set below

5.5 Set

5.5.0.1 Union-Find (Disjoint-set)

Union Find: In computer science, a disjoint-set data structure, also called a union–find data structure or merge–find set, is a data structure that stores a collection of disjoint (non-overlapping) sets. Equivalently, it stores a partition of a set into disjoint subsets.

In Union find, it provides operations for adding new sets, merging sets (replacing them by their union), and finding a representative member of a set. The last operation makes it possible to find out efficiently if any two elements are in the same or different sets.

The following is the template of UnionFind, about the path compression or union based on rank could be find in the book Introduction to Algorithm Chapter 21 or huahua union find

```

class UnionFindSet:
    def __init__(self, n):
        self._parents = [i for i in range(n + 1)]
        self._ranks = [1 for i in range(n + 1)]

    def find(self, u):
        while u != self._parents[u]:
            self._parents[u] = self._parents[self._parents[u]]
            u = self._parents[u]

```

```

    return u

def union(self, u, v):
    pu, pv = self.find(u), self.find(v)
    if pu == pv: return False

    if self._ranks[pu] < self._ranks[pv]:
        self._parents[pu] = pv
    elif self._ranks[pu] > self._ranks[pv]:
        self._parents[pv] = pu
    else:
        self._parents[pv] = pu
        self._ranks[pu] += 1

    return True

```

5.6 Search

5.6.1 Backtrack

Summarize from labladong:

- subset: start
- combination: start
- permutation used
- bfs:
- dfs:

Backtracking is the traversal of decision tree in essence. There are 3 things:

- Path: decisions have been made
- Choice list: the choices you can choose now
- End condition: reach to the end of the decision tree

General framework of the backtracking is to do **recursion in for loop, do a choice before the recursion and undo the choice after the recursion**

```

res = []
def backtrack(path,choice_list):
    if end_condition:
        result.append(path)

```

```

    return

    for c in choice_list:
        do the choice (select the choice)
        backtrack(path, choice_list)
        undo the choice c (remove the choice from choice_list)

```

5.6.1.1 combination

Input 2 number n and k, output the all combinations of k number in [1..n]. This is also a backtracking problem, but k limit the tree depth, and n limit the width of the tree. Which means the algorithm is similar to subset problem, the only difference **is that k limit the depth, so the end condition should take k into account and end the recursion.** Other than this, combination and subset problem should include **start** index to indicate that the element used before can not be used. There are 2 cases.

- `backtrack(path + [something], i)`: means if combining [1,2,3,4], i = 1 then element 2 can be used in the next round but 1 cannot be used (different from permutation)
- `backtrack(path + [something], i + 1)`: means if combining [1,2,3,4], i = 1 then element 2 can not be used in the next round but 1 cannot be used, it could only start from element 3(index = 2)

Figure 5.7 shown below shows the tree like search process, in some cases, the combination will have constraints like there is **duplicate** inside the list choices but the output cannot have duplicate result, this Figure below(lowest tree) shows it will have same result when it contains duplicate, then an expression `if i > start and c[i] == c[i - 1] continue` should be used, and the choice list should be **sorted** beforehand. Also path + [i] is a technique here work as DFS, the traditional way is to `path.append()` and `path.pop()`. #17 #39 #40 are the combination problem, the template below is the solution of #40 Combination Sum II.

```

def combinationSum2(self, c, t):
    res = []
    if c is None or t == 0:
        return res

    def backtrack(path, target, start):
        if target == 0:
            res.append(path)
            return

        for i in range(start, len(c)):
            if c[i] > target:
                continue

            path.append(c[i])
            backtrack(path, target - c[i], i + 1)
            path.pop()

```

```

        if i > start and c[i] == c[i - 1]:
            continue
        backtrack(path + [c[i]], target - c[i], i + 1)

c.sort()
backtrack([], t, 0)
return res

```



images/combination.jpg

Figure 5.7: Combination

5.6.1.2 subset

concept of these questions: given an array, return its all subsets. eg: `nums = [1,2,3]`, the subsets are `[[],[1],[2],[3],[1,3],[2,3],[1,2],[1,2,3]]`, the order doesn't matter. The question is similar to combination, the only difference is that it doesn't need to have any specific rule to append the path into res.

```

def subset(path, start):
    res.append(path)

```

```
for i in range(start, len(nums)):
    subset(path + [nums[i]], i+1)
```

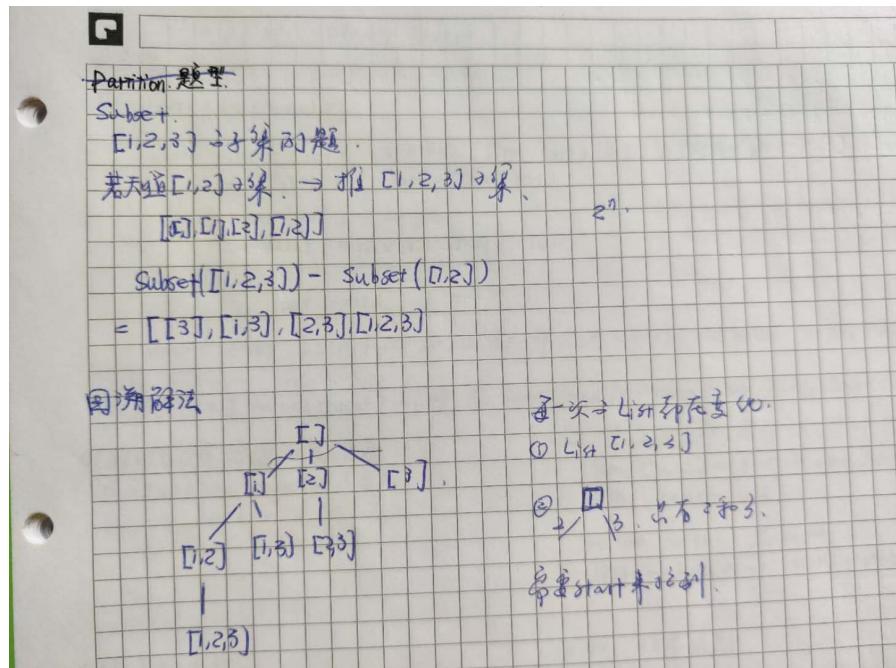


Figure 5.8: subset

5.6.1.3 permutation

The only difference with subset and combination is that the number before could be used, so be careful when doing this, we only need a 'used' array to denote whether the element have been used in this recursion. The followng is the solution for #47

An excellent explaination of whether to use 'used[i]' or 'not used[i]' when doing pruning could be found in <https://www.cxyxiaowu.com/2946.html>here.

```
def permuteUnique(self, nums):
    res = []
    if nums == None:
        return res
    used = [0] * len(nums)
    def backtrack(path):
        if len(path) == len(nums):
            res.append(path)

        for i in range(len(nums)):
```

```

        if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
            continue
        if used[i]:
            continue
        used[i] = 1
        backtrack(path + [nums[i]])
        used[i] = 0

nums.sort()
backtrack([])
return res

```

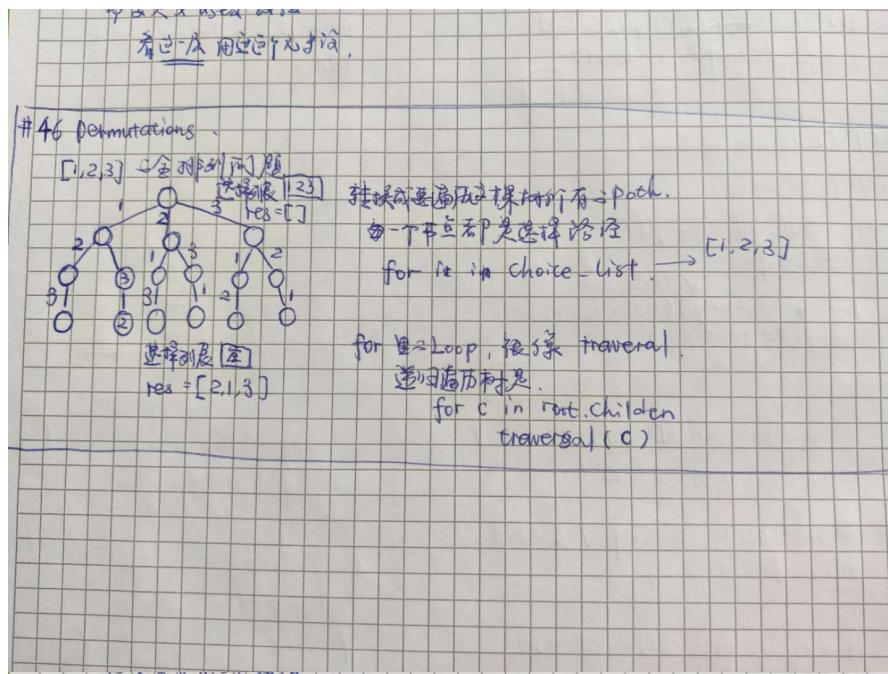


Figure 5.9: Permutation

5.6.2 BFS

The template of BFS is similar to level order traversal of a binary tree, using queue to append root and while loop the root, since we need to get the node level by level, so the size of nodes inside queue should be gained in order to for loop all the nodes inside queue. BFS is a typical **Start to Target** problem, and always is to find the **Shortest Path from Start to the Target**. Recall in level order traversal there is no end condition in for loop, while in BFS, if we found the **Target**, then we should return the step. A typical return value is the value(num of levels to get to the target). Other than that, since this is not a tree anymore, the search direction is not stricted in downward(left

node or right node), it could search all of the adjacent nodes, so a visited **set** should be initialized to ensure the node has not been visted yet. BFS is usually used for solving the least step to solve some search based problem.

The following is the template for BFS:

```

1 // 计算从起点 start 到终点 target 的最近距离
2 int BFS(Node start, Node target) {
3     Queue<Node> q; // 核心数据结构
4     Set<Node> visited; // 避免走回头路
5
6     q.offer(start); // 将起点加入队列
7     visited.add(start);
8     int step = 0; // 记录扩散的步数
9
10    while (q not empty) {
11        int sz = q.size();
12        /* 将当前队列中的所有节点向四周扩散 */
13        for (int i = 0; i < sz; i++) {
14            Node cur = q.poll();
15            /* 划重点：这里判断是否到达终点 */
16            if (cur is target)
17                return step;
18            /* 将 cur 的相邻节点加入队列 */
19            for (Node x : cur.adj())
20                if (x not in visited) {
21                    q.offer(x);
22                    visited.add(x);
23                }
24        }
25        /* 划重点：更新步数在这里 */
26        step++;
27    }
28 }
```

Figure 5.10: BFS

126 word ladder is a typical BFS problem. But in typical BFS problem, normally inside the for loop, we will see if the node is in visited, if it is in visied, then stop this loop with **continue**, while in this case we want to find the node which is in the visited, so those are not in the visited should be ignored. So strictly speaking, this **set** should be declared as wordListSet instead of visited.

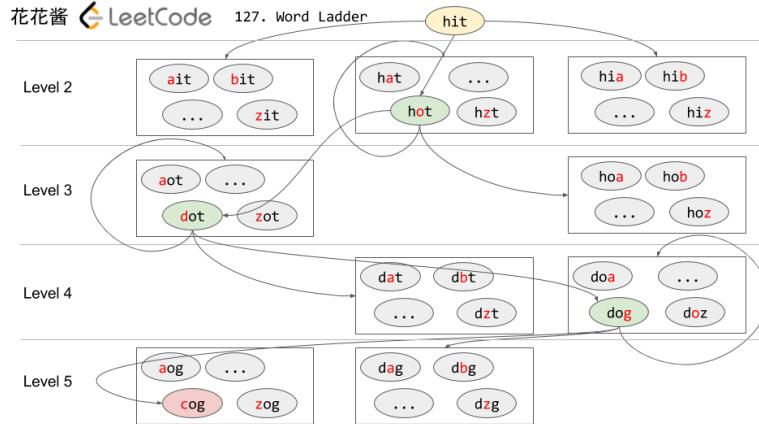


Figure 5.11: wordLadder

```

def ladderLength(self, beginWord, endWord, wordList):
    visted = set(wordList)
    q = collections.deque()
    q.append(beginWord)
    if endWord not in visted: return 0
    step = 0

    while q:
        qSize = len(q)
        step += 1
        for k in range(qSize):
            node = q.popleft()
            for i in range(len(node)):
                for j in "abcdefghijklmnopqrstuvwxyz":
                    word = node[:i] + j + node[i+1:]
                    if endWord == word and word in visted:
                        return step+1

                    if word not in visted:
                        continue

                    if word in visted and word != node:
                        visted.remove(word)
                        q.append(word)
    return 0

```

#752 Open the lock, the first code "0000" should also be put into the deadends set to avoid go back to the origin lock set. simple, but the point of the circle number is important to be remembered. 0 to 9 and 9 to 0, which could be done by mod 10.

```

def openLock(self, deadends, target):
    if "" == target or "0000" in deadends or target in deadends:
        return -1

    q = collections.deque()
    q.append("0000")

    visited = set(deadends)
    visited.add("0000")

    step = 0
    while q:
        step += 1
        qSize = len(q)

        for i in range(qSize):
            node = q.popleft()
            for j in range(len(node)):
                for k in (1, -1):
                    num = (int(node[j]) + k) % 10
                    new = node[:j] + str(num) + node[j+1:]

                    if new == target:
                        return step

                    if new in visited:
                        continue

                    q.append(new)
                    visited.add(new)

    return -1

```

#542 01 Matrix, get every step into the dist function. the interesting thing here is when search in 4 different directions, we could use dirs = [0,-1,0,1,0] to reduce the amount of code we need to write, simply loop in range(4) and add the first to x and add the second to y. This solution below is the one that will not exceed the time limit

The idea of this question is first enqueue all of the positions that are 0, the rest of them are initialized as float('inf'), then use BFS with 4 different directions to find the distance difference that is bigger than 1, then update the dist + 1.

```

def updateMatrix(self, matrix):
    n = len(matrix)
    m = len(matrix[0])

```

```

q = collections.deque()
dist = [[float("inf") for i in range(m)] for j in range(n)]

for i in range(n):
    for j in range(m):
        if matrix[i][j] == 0:
            dist[i][j] = 0
            q.append((i, j))

dirs = [0,-1,0,1,0]
while q:
    x, y = q.popleft()

    for i in range(4):
        a = x + dirs[i]
        b = y + dirs[i+1]
        if 0<=a<n and 0<=b<m and dist[a][b] - dist[x][y] > 1:
            dist[a][b] = dist[x][y] + 1
            q.append((a, b))

return dist

#675 Cut Off Trees for Golf Event.

def cutOffTree(self, f):
    if f == None:
        return -1
    h = []
    # sort the height
    m = len(f)
    n = len(f[0])
    for i in range(m):
        for j in range(n):
            if f[i][j] != 0:
                h.append((f[i][j], i, j))

def bfs(f, sx, sy, tx, ty):
    q = collections.deque()
    q.append((sx, sy))
    visited = [[0 for i in range(n)] for j in range(m)]

    step = 0
    dirs = [0, -1, 0, 1, 0]

```

```

while q:
    qSize = len(q)

    for i in range(qSize):
        (x, y) = q.popleft()
        if x == tx and y == ty:
            return step
        for j in range(4):
            nx = x + dirs[j]
            ny = y + dirs[j + 1]

            # Out of bound or unwalkable
            if (nx < 0 or nx == m or ny < 0 or ny == n or not f[nx][ny] or
                visited[nx][ny] == 1):
                continue
            q.append((nx, ny))
            visited[nx][ny] = 1
        step += 1

    return float('inf')

# sort the height
h.sort()
sx = 0
sy = 0
ts = 0

# loop the height an call BFS
for i in range(len(h)):
    tx = h[i][1]
    ty = h[i][2]

    s = bfs(f, sx, sy, tx, ty)
    if s == float('inf'):
        return -1

    ts += s
    sx = tx
    sy = ty

return ts

```

5.6.3 DFS

DFS is backtracking problem in essence, for #301, this is a hard question, the template is the same as the backtrack problem, the difference is that we need to know how many l and r should be deleted(kinda like target in this case), for every element is list_choice, just simply remove the element at this level and continue the recursion. curr = s[:i] + s[i+1:] will do this action. While the way to verified if a sequence of parentheses are valid is shown in the section 5.2.5.

```

def removeInvalidParentheses(self, s):
    res = []
    if s == "":
        return [""]
    l = 0
    r = 0
    for i in s:
        if i == '(':
            l += 1
        if i == ')' and l == 0:
            r += 1
        elif i == ')':
            l -= 1

#2. if a string is valid
def valid(s):
    l = 0
    r = 0
    for i in s:
        if i == '(':
            l += 1
        if i == ')' and l == 0:
            r += 1
        elif i == ')':
            l -= 1
    return l == 0 and r == 0

def backtrack(s, l, r, start):
    if l == 0 and r == 0 and valid(s):
        res.append(s)
        return

    for i in range(start, len(s)):
        if i > start and s[i] == s[i - 1]:
            continue

```

```

# if s[i] is parenthesis
if s[i] == '(' or s[i] == ')':
    curr = s[:i] + s[i+1:]
# start from right parenthesis
if r > 0:
    # start is i because i is removed, the i+1 now becomes
    # i, so start again with i instead of i + 1
    backtrack(curr, l, r-1, i)
elif l > 0:
    backtrack(curr, l-1, r, i)
backtrack(s, l, r, 0)
return res

```

For #22, same backtracking problem, but the key thing is to understand for loop in backtracking, in this case, there is no need to use for loop, for loop only used in the situation that there are several list choices, in this case, the choices are left or right parentheses, but the actions are taken in different condition, which means it has all choices made already.

```

def generateParenthesis(self, n):
    res = []
    if n == 0:
        return [""]

    def backtrack(path, l, r):
        if l > r:
            return

        if l == 0 and r == 0:
            res.append(path)
            return
        if l > 0:
            backtrack(path + '(', l-1, r)
        if r > 0:
            backtrack(path + ')', l, r-1)
    backtrack("", n, n)
    return res

```

5.6.3.1 Sudoku, N-Queue

#37 Sudoku, this is similar to permutation, this is a brute force DFS, the used array have 3, cols, rows and boxes, the only thing need to be remember is the when found 1 solution, it could return True already, so it won't search other results. And the index of box is also important in this case $x*3 + y$.

```

def solveSudoku(self, board):
    if not board:
        return
    # initialize the board

    rows = [[0 for i in range(10)] for j in range(9)]
    cols = [[0 for i in range(10)] for i in range(9)]
    box = [[0 for i in range(10)] for j in range(9)]

    for i in range(9):
        for j in range(9):
            if board[i][j] != '.':
                a = int(board[i][j])
                rows[i][a] = 1
                cols[j][a] = 1
                box_x = i // 3
                box_y = j // 3
                box[box_x*3 + box_y][a] = 1

    def backtrack(x, y):
        if x == 9:
            return True

        ny = (y + 1) % 9
        if ny == 0:
            nx = x + 1
        else:
            nx = x

        if board[x][y] != '.':
            return backtrack(nx, ny)

        # backtracking
        for i in range(1, 10):
            box_x = x // 3
            box_y = y // 3
            box_index = box_x * 3 + box_y
            if not rows[x][i] and not cols[y][i] and not box[box_index][i]:
                # take actions
                rows[x][i] = 1
                cols[y][i] = 1
                box[box_index][i] = 1

```

```

board[x][y] = str(i)

if backtrack(nx, ny):
    return True

board[x][y] = '.'
rows[x][i] = 0
cols[y][i] = 0
box[box_index][i] = 0
return False

backtrack(0, 0)

```

#51 N-Queue, the ‘used’ array also have 3, rows, diag1 and diag2, in this case, the bracktrack parameter x means cols(rows is also fine). For Sudoku, the choice list have 1 to 9, so for in range(1, 10), but in N-Queue, the choice now becomes cols, so only for in range(n). and we can get corresponding index of those ‘used’ array but only one index ‘i’, so rows, diags and diag2 is only a one dimentional array.

```

def solveNQueens(self, n):
    res = []
    if n == 0:
        return res

    rows = [0] * n
    diag1 = [0] * (2*n - 1)
    diag2 = [0] * (2*n - 1)

#initialize the board
board = [['.' for _ in range(n)] for _ in range(n)]

def backtrack(x):
    if x == n:
        res.append(["".join(i) for i in board])
        return

    for i in range(n):
        if not rows[i] and not diag1[x+i] and not diag2[x - i + n - 1]:
            rows[i] = 1
            diag1[x+i] = 1
            diag2[x - i + n - 1] = 1
            board[x][i] = 'Q'

            backtrack(x+1)

```

```

rows[i] = 0
diag1[x+i] = 0
diag2[x - i + n - 1] = 0
board[x][i] = '.'

backtrack(0)
return res

```

Word Search, same backtracking, but be careful when used this element in the board and set it to nil and set it back after the backtrack. Otherwise this element might be used again in the next round.

```

def exist(self, board, word):
    if word == "":
        return True
    m = len(board)
    n = len(board[0])
    def backtrack(x, y, word):
        if len(word) == 0:
            return True

        if x < 0 or x >= m or y < 0 or y >= n or board[x][y] != word[0]:
            return False

        t = board[x][y]
        board[x][y] = '#'
        res= backtrack(x+1, y, word[1:]) or backtrack(x, y+1, word[1:]) or backtrack(x-1,
        board[x][y] = t
        return res
    for i in range(m):
        for j in range(n):
            if backtrack(i, j, word):
                return True

    return False

```

5.6.3.2 DFS+BFS

#934 Shortest Bridge, this is a very good problem because it covers both DFS and BFS, to find the shortest path of 2 island, the first step is to find one of the island and marked it as differnt element(2 in my case), because from the perspective of finding the same 1 in the deepest path, this is something DFS usually dose, so dfs is used here to search an island thoroughly, and the same time enqueue all of the node found in dfs, this is kinda like enqueue all node in the same level(From the perspective of another island).

The second step is to use classical BFS, the node marked as 2 could be re-used as ‘visited’ set in BFS.

```

def shortestBridge(self, A):
    # dfs and mark an island to 2
    m = len(A)
    n = len(A[0])

    q = collections.deque()
    # dfs
    def backtrack(x, y):
        # end condition
        if x == m or x < 0 or y == n or y < 0 or A[x][y] != 1:
            return
        A[x][y] = 2
        q.append((x, y))
        for a, b in ((x+1, y), (x-1, y), (x, y+1), (x, y-1)):
            backtrack(a, b)

    Found = False
    for i in range(m):
        for j in range(n):
            if A[i][j] == 1:
                backtrack(i, j)
                Found = True
                break
        if Found == True:
            break

    # bfs
    dirs = [0, -1, 0, 1, 0]
    step = 0
    while q:
        qSize = len(q)
        for i in range(qSize):
            x, y = q.popleft()
            for j in range(4):
                a = x + dirs[j]
                b = y + dirs[j + 1]

                if a == m or a < 0 or b == n or b < 0 or A[a][b] == 2:
                    continue
                A[a][b] = 2
                q.append((a, b))
                step += 1

```

```

if A[a][b] == 1:
    return step

A[a][b] = 2
q.append((a, b))

step += 1

return -1

```

5.7 Binary Search

The Binary Search Algorithm is very easy, but the detail is very important, lower bound, upper bound, return mid + 1 or mid - 1. The open interval and closed interval are also a question that needs to be remember.

The template for the Binary Search is as follows:

```

# return the first value that is not less than value in [first, last)
def lower_bound(array, first, last, value):
    while first < last:
        mid = first + (last - first) // 2
        if array[mid] < value: first = mid + 1
        else: last = mid
    return first # last is also fine

```

There are 2 cases here:

First is to find lower bound, which means to find $x \geq value$ or $x > value$, the index of x is the minimum. Use $[first, last)$ interval.

Second is to find the upper bound, which is to find the $x \mid value$ or $x \leq value$, where the index of x is the largest.

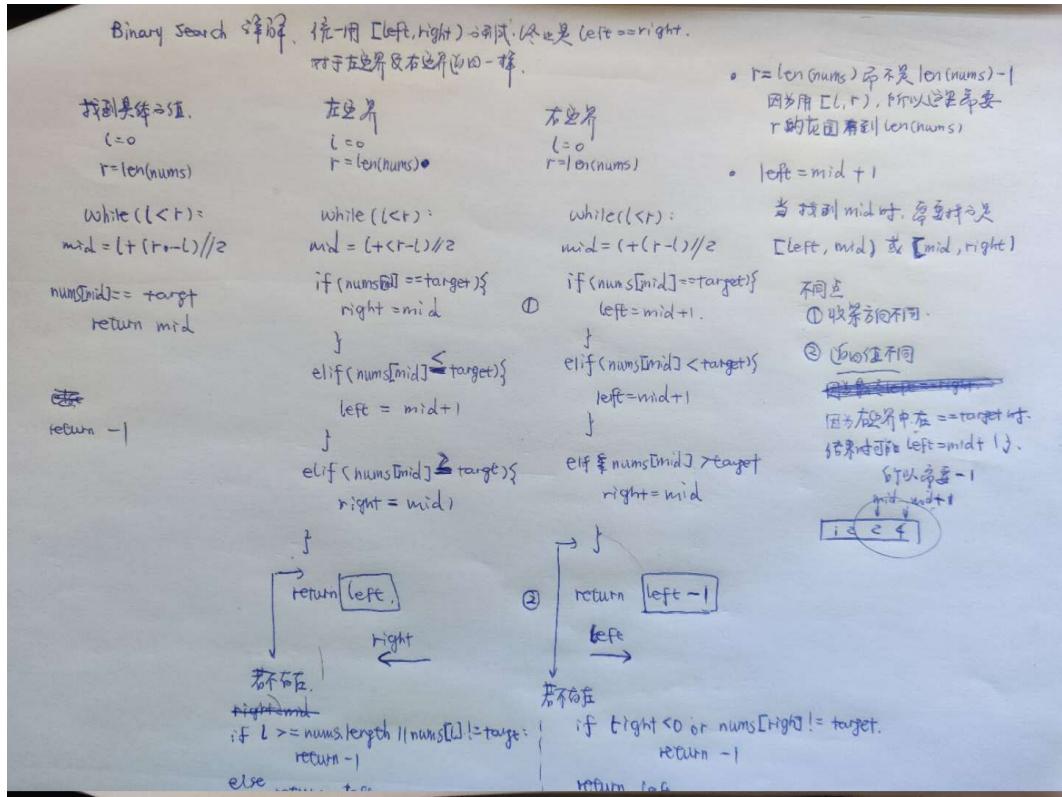


Figure 5.12: peak number

leetcode #33 want to find a value inside a rotated array, this actually could be solved by using indexof but since the problem should be solved in nlogn, then binary search should be used here. The important part is **to know which part is strictly increasing sequence**, for example, a sequence is 4,5,6,7,0,1,2. say if we want to find value 0, if mid is in 7, then we simply compare mid with r and l to see which part is sorted, obviously, 4,5,6,7 is sorted, while 7,0,1,2 is not sorted, after know which part is sorted, just simply compare it with **target to see whether target is in this fields**. if not, then shrink the mid. Remember we need to compare our mid target with the right value, so we will use closure $[l, r]$ instead of $[l, r)$ in this one.

```

def search(self, nums, target):
    """
    :type nums: List[int]
    :type target: int
    :rtype: bool
    """

    N = len(nums)
    l, r = 0, N - 1
    while l <= r:

```

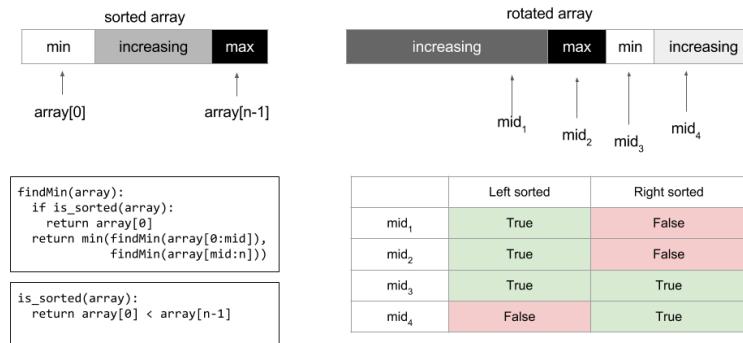
```

while l < r and nums[l] == nums[r]:
    l += 1
mid = l + (r - l) / 2
if nums[mid] == target:
    return mid
if nums[mid] >= nums[l]:
    if nums[l] <= target < nums[mid]:
        r = mid - 1
    else:
        l = mid + 1
elif nums[mid] <= nums[r]:
    if nums[mid] < target <= nums[r]:
        l = mid + 1
    else:
        r = mid - 1
return -1

```

in leetcode #153, try to find the minimum of a rotated array, just use recursion+binary search to do that.

153. Find Minimum in Rotated Sorted Array



花花酱 LeetCode

Figure 5.13: Minimum number of Rotated array 1

153. Find Minimum in Rotated Sorted Array

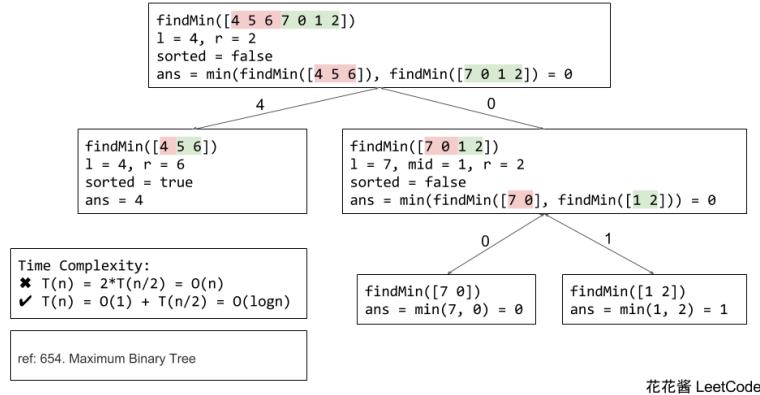


Figure 5.14: Minimum number of Rotated array 1

```

def findMin(self, nums):
    def bsMin(nums, l, r):
        # bs, end condition is l==r(only 1 element), if 2 left, l+1==r
        if l + 1 >= r:
            return min(nums[l], nums[r])

        # sorted
        if nums[l] < nums[r]:
            return nums[l]

        mid = l + (r - 1) // 2
        return min(bsMin(nums, mid + 1, r), bsMin(nums, l, mid))

    l = 0
    r = len(nums) - 1
    return bsMin(nums, l, r)

```

leetcode #162 peak number, need to find a peak number in an array, use 2 pointers in this case. The process is shown below.

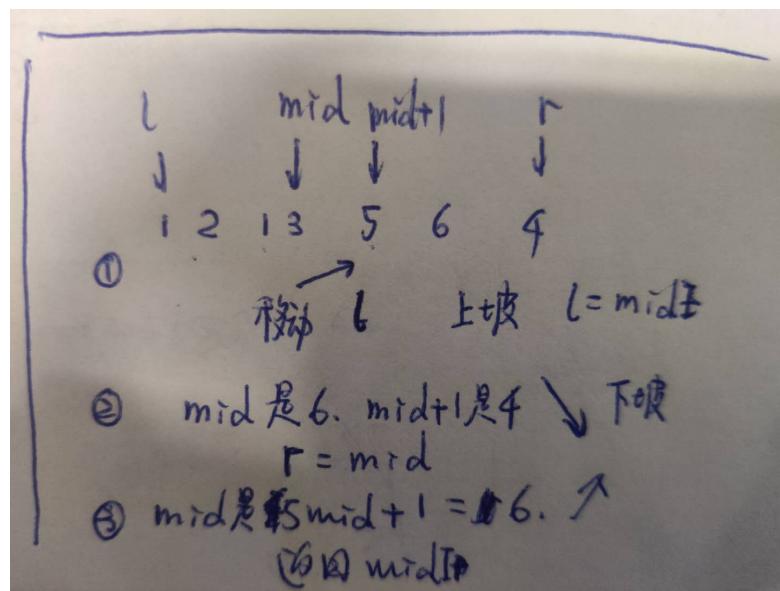


Figure 5.15: peak number

```
def findPeakElement(self, nums):
    l = 0
    r = len(nums) - 1
    while l < r:
        mid = l + (r - l) // 2
        mid1 = mid + 1

        if nums[mid] < nums[mid1]:
            l = mid1
        else:
            r = mid

    return l
```

#69 upper bound, be careful that the l should be 1 instead of 0, the reason to find upper bound instead of lower bound is that **to find the first that is bigger than the result or say the rightest one that close to the result**

```
def mySqrt(self, x):
    l = 1
    r = x

    while l < r:
        mid = l + (r - l) // 2
        if mid ** 2 == x:
```

```

        return mid
    elif mid ** 2 > x:
        r = mid
    else:
        l = mid + 1

    return l - 1

```

#378 Find the smallest x(lower bound Binary Search), such that there are k elements that are smaller or equal to x(upper bound).

```

def kthSmallest(self, matrix, k):
    if not matrix:
        return -1

    l = matrix[0][0]
    r = matrix[-1][-1] + 1

    while l < r:
        mid = l + (r - 1) // 2

        loc = sum(bisect_right(a, mid) for a in matrix)

        if loc == k:
            r = mid
        elif loc < k:
            l = mid + 1
        else:
            r = mid

    return l

```

#668, it is a little bit different from the #378, the matrix is just the multiplication of col and row, if use the way above(bisect_right) will cause memory limit, so instead of using sum over the bisect_right, we use a function called lex, for every rows(m), compare the col with mid/i , i is every row here, we take advantage of the fact that each cell in matrix is actually just a multiplication, mid/i can calculate the boundary in this line, for example, [1,2,3],[2,4,6],[3,6,9], the first mid will be 5, then when scan the first row, n=3, $mid/1 = 5$, so in the first line there are $\min(3,5)$ 3 numbers that is smaller than the mid, for the second line, m=2, mid=5, n=3, $mid//m = 2.5$, then in this case, there are 2 numbers that is smaller than 5 (2 and 4). The following slide shows the detail reasoning.

   花花酱  huahualeetcode

668. Kth Smallest Number in Multiplication Table

Solution 1: Brute Force (MLE) Generate the table, and find k-th element Time complexity: $O(m \cdot n)$ Space complexity: $O(m \cdot n) = 0.9 \cdot 4 = 3.6 \text{ GB}$	how many #s are $\leq x$ in the table. Do it row by row 1 2 3 ... $n \cdot 1$ 2 4 6 ... $n \cdot 2$... i $2 \cdot i$ $3 \cdot i$... $n \cdot i \leq x$... m $2 \cdot m$ $3 \cdot m$... $n \cdot m$ 1, 2, 3, ..., n $\leq x/i$ (both /i) $\#s \leq x = \min(x/i, n)$ def lex(x): # O(m) return sum(min(n, x/i)) i=1..m	Why use $\text{lex}(x) \geq k$, not $\text{lex}(x) == k$? There are duplications! n = 3, m = 3, k = 7 1 2 3 2 4 6 3 6 9 Sorted: 1, 2, 2, 3, 3, 4, 6, 6, 9 $\text{lex}'(9) = 9 \geq 7$ lex'(6) = 8 >= 7 $\text{lex}'(4) = 6$
Solution 2: Binary Search Target: the k-th element x. Find smallest x such that there are at least k elements $\leq x$ in the table	<pre> l = 1, r = m * n + 1 while l < r: x = l + (r - 1) // 2 if lex(x) >= k: r = x else: l = x + 1 return l </pre>	Time complexity: $O(m \cdot \log(m \cdot n))$ Space complexity: $O(1)$
Related Problems: k-th element LC378 <= LC668 < LC786		Pruning: 1. sum is already $\geq k$, $\text{lex}(x) \geq k$ is already true 2. x/i is already 0, i goes up x/i goes down, sum will not change after that

<https://youtube.com/c/huahualeetcode> <https://space.bilibili.com/9880352> <https://zxi.mytechroad.com/blog/> 225

Figure 5.16: Kth Smallest Number in Multiplication Table (From huahua)

```

def findKthNumber(self, m, n, k):
    l = 1
    r = m*n+1

    def lex(mid, m, n):
        count = 0
        for i in range(1, m+1):
            count += min(n, mid // i)
        return count

    while l < r:
        mid = l + (r - 1) // 2
        loc = lex(mid, m, n)

        if loc >= k:
            r = mid
        else:
            l = mid + 1

    return l

```

#786, a hard question, should take advantage of the property of the matrix, for each row, found the first col that is $A[i]/A[j] \leq m$, in other words, $A[i] < A[j] * m$, so the rest will all smaller. The detail is shown below.

LeetCode YouTube 花花酱 huahualeetcode

Solution 0: Brute Force / Sorting TLE

Time complexity: $O(n^2 \log n)$
Space complexity: $O(n^2)$

Solution 1: Binary Search Accepted

Related Problems: LC 378/719

Search m , $0 < m < 1$
Such that there are exact K pairs of (i,j) that $A[i] / A[j] \leq m$

Create a virtual matrix to speed up the search process

Time complexity: $O(\log(\max(n))n^2)$ TLE
 $O(\log(\max(n))n)$
Space complexity: $O(1)$

786. K-th Smallest Prime Fraction

			smallest
1/2	1/3	1/5	
-	2/3	2/5	
-	-	3/5	

A = [1, 2, 3, 5], K = 3

def: $M[i][j] = A[i] / A[j]$
 $0 \leq i \leq n - 1, i < j \leq n - 1$
 $M[i][j] < M[i][j - 1]$
 $M[i][j] > M[i][j + 1]$

if $M[i][j] > m \Rightarrow M[i + 1][j] \geq m$

l	r	m	$A[i] / A[j] \leq m$	total
0	1	0.5	1/2, 1/3, 1/5, 2/5	$4 > K$
0	0.5	0.25	1/5	$1 < K$
0.25	0.5	0.375	1/3, 1/5	$2 < K$
0.375	0.5	0.4375	1/3, 1/5, 2/5	$3 = K$

<http://zxi.mytechroad.com/blog/>

Figure 5.17: K-th Smallest Prime Fraction

```

def kthSmallestPrimeFraction(self, A, K):
    l = 0.0
    r = 1.0
    n = len(A)

    while l < r:
        mid = (l + r) / 2
        max_f = 0.0
        total = 0
        j = 1
        for i in range(n - 1):
            while j < n and A[i] > A[j] * mid:
                j += 1
            if j == n:
                break
            total += n - j
            f = A[i] / A[j]
            if f > max_f:
                p, q, max_f = i, j, f

        if total == K:
            return [A[p], A[q]]
        elif total > K:
            r = mid
    
```

```

else:
    l = mid

return []

```

5.8 Dynamic Programming

Normally the problem of DP is the problem of **finding maximum or minimum solutions**, that is to say, to find optimal solution in operation research, for example, longest increasing sub-sequence, minimum edit distance. To find the extremum, the core is **exhaustive method**, a little different from pure exhaustive method, DP related questions should be optimized using DP table, after the dp table, the dynamic transition function and optimal sub structure is the key to the answer. So the most difficult part is to write down the **dynamic transition function**. The process to write down DTF is as follows:

- base case
- know the state space
- choices
- defined the dp matrix

```

1 # 初始化 base case
2 dp[0][0][...] = base
3 # 进行状态转移
4 for 状态1 in 状态1的所有取值：
5     for 状态2 in 状态2的所有取值：
6         for ...
7             dp[状态1][状态2][...] = 求最值(选择1, 选择2...)

```

Figure 5.18: DP vague template

5.8.1 Simple Example: Fibonacci

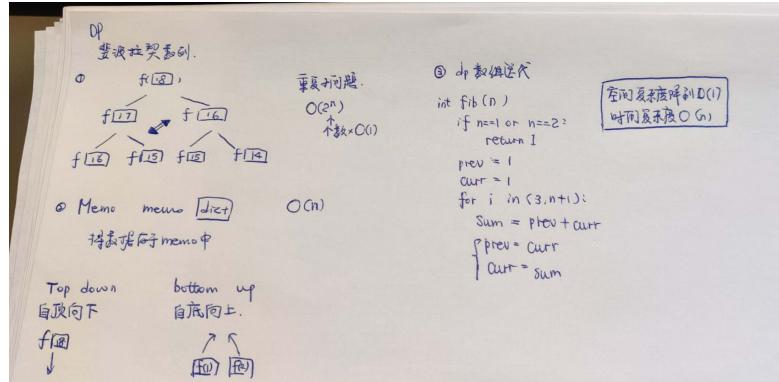


Figure 5.19: DP Example

5.8.2 Coin Change

#322 You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount.

The base case here is when amount is 0 then we only need 0 coin. State is the target amount, and choice is the coins you have, the dp array is defined as: input n aim amount, the dp array will return the fewest amount of coins

LeetCode
YouTube
花花酱
huahualeetcode
322. Coin Change

Solution: DP

Def: $dp[i][j]$: min coins to make up j amount using first i types of coins.
Init: $dp[-1][0] = 0$, $dp[-1][i] = \infty$
Transition: $dp[i][j] = \min(dp[i-1][j], dp[i-1][j - k * coin_i] + k)$
 $= \min(dp[i][j], dp[i][j - coin_i] + 1)$
Ans: $dp[n-1][amount]$
Time complexity: $O(n * amount^2) \rightarrow O(n * amount)$
Space complexity: $O(n * amount) \rightarrow O(amount)$

num	$dp[0]$	$dp[1]$	$dp[2]$	$dp[3]$	$dp[4]$	$dp[5]$	$dp[6]$	$dp[7]$	$dp[8]$	$dp[9]$	$dp[10]$	$dp[11]$
-	0	∞										
1	0	1	2	3	4	5	6	7	8	9	10	11
2	0	1	1	2	2	3	3	4	4	5	5	6
5	0	1	1	2	2	1	2	2	3	3	2	3

<http://zxi.mytechroad.com/blog/>

Figure 5.20: Coin Change

It is noticeable that defined the space of dp as amount + 1 instead of amount, because 0 to amount has the number of amount + 1.

```
def coinChange(self, coins, amount):
```

```

dp = [float('inf')] * (amount + 1)

dp[0] = 0
for i in range(amount+1):
    for coin in coins:
        if coin > i:
            continue
        dp[i] = min(dp[i], dp[i - coin] + 1)

return -1 if dp[-1] == float('inf') else dp[-1]

```

For #377, it is the same the only difference is that the base case $dp[0]$ is 1, which means there is 1 way to sum up to target 0, and since it is not the case of getting the min, then sum up all of the possibilities.(The dp defines as given the target index, return the number of possibilities that could sum up to target index)

```

class Solution(object):
    def combinationSum4(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        """
        dp = [0] * (target+1)
        dp[0] = 1
        for i in range(target+1):
            for num in nums:
                if i - num >= 0:
                    dp[i] += dp[i - num]
        return dp[-1]

```

5.8.3 Longest Increasing Subsequence (LIS) $O(n^2)$

Problem statement: given a non sorted array, find the longest increasing subsequence. be careful about the concept of subsequence and substring. substring should be continuous while subsequence does not have to be continuous.

define dp array, the $dp[i]$ array means the longest subsequence of a certain number $nums[i]$, the dp array should be initialized as 1 since itself count as 1. The process is as follows:

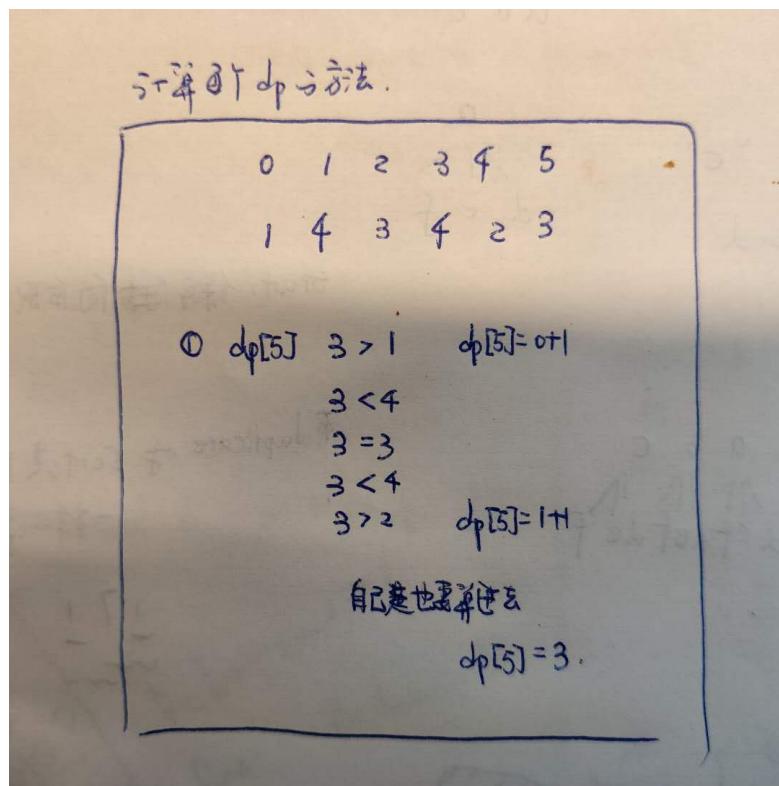


Figure 5.21: LIS

Remember that in the process shown above, $dp[5]$ should be $1 + 1$ or $dp[5] = 2 + 1$, the 1 and 2 actually is the result from $dp[0]$ and $dp[4]$. That explains the part of $dp[i] = \max(dp[i], dp[j] + 1)$. #310

```
def lengthOfLIS(self, nums):
    """
    :type nums: List[int]
    :rtype: int
    """
    if not nums:
        return 0

    l = len(nums)
    dp = [1] * l
    for i in range(1, l):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

#673 is to know the number of LIS, when found the longest one, update LISN[i], if found the same, then sum them up. for example (1,3,4,7) and (1,3,5,7)

```
def findNumberOfLIS(self, nums):
    """
    :type nums: List[int]
    :rtype: int
    """
    if not nums:
        return 0
    l = len(nums)
    LIS = [1] * l
    LISN = [1] * l
    for i in range(1, l):
        for j in range(i):
            if nums[i] > nums[j]:
                if LIS[j] + 1 > LIS[i]:
                    LIS[i] = LIS[j] + 1
                    LISN[i] = LISN[j]
                elif LIS[j] + 1 == LIS[i]:
                    LISN[i] += LISN[j]
    total = 0
    a = max(LIS)
    for i in range(l):
        if LIS[i] == a:
            total += LISN[i]

    return total
```

Follow up, for #673, to find the Number of Longest Increasing Subsequence, then only compare to the one before the current element i.

```
def findLengthOfLCIS(self, nums):
    if not nums:
        return 0
    dp = [1] * len(nums)
    for i in range(1, len(nums)):
        if nums[i] > nums[i-1]:
            dp[i] = max(dp[i], dp[i-1]+1)
    # print(dp)
    return max(dp)
```

5.8.4 Maximum Subarray

Problem: given an array and return the maximum subarray (sum up), this sumarray should be continuous. The dp array should be defined as dp[i] stands for the maximum

subarray. The transition function is reasoning like this: if we know the $dp[i-1]$, then $dp[i]$ will be known by taking 2 choices, take the next element or not.

Then the transition is:

The algorithm could be written as

```
def maxSubArray(self, nums):
    if not nums:
        return 0

    dp = [0] * len(nums)
    dp[0] = nums[0]
    for i in range(1, len(nums)):
        dp[i] = max(dp[i], dp[i-1] + nums[i])

    return max(dp)
```

Notice that $dp[i]$ only relies on $dp[i-1]$, then we can shrink the space to $O(1)$

```
def maxSubArray(self, nums):
    if not nums:
        return 0

    dp0 = nums[0]
    dp1 = 0
    max1 = dp0
    for i in range(len(nums)):
        dp1 = max(nums[i], dp0 + nums[i])
        dp0 = dp1
        max1 = max(max1, dp0)
    return max1
```

Notice that

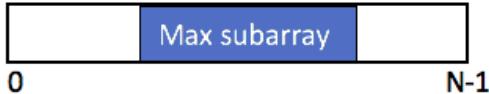
```
def maxSubarraySumCircular(self, A):
    """
    :type A: List[int]
    :rtype: int
    """

    maxSum, curMax= A[0], 0
    for a in A:
        curMax = max(curMax + a, a)
        maxSum = max(maxSum, curMax)

    return maxSum
```

Now change this problem to a circular list, then there are 2 cases that should be taken into account. 893 Maximum Sum Circular Subarray

Case 1: max subarray is not circular.



Case 2: max subarray is circular.

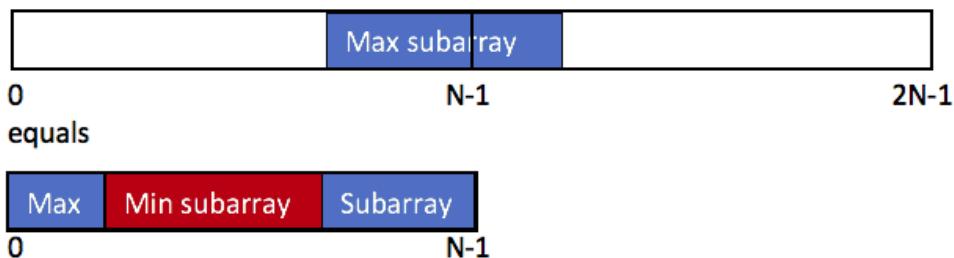


Figure 5.22: Maximum Sum Circular Subarray

```
def maxSubarraySumCircular(self, A):
    """
    :type A: List[int]
    :rtype: int
    """
    if not A:
        return 0

    total, maxSum, curMax, minSum, curMin = 0, A[0], 0, A[0], 0
    for a in A:
        curMax = max(curMax + a, a)
        maxSum = max(maxSum, curMax)
        curMin = min(curMin + a, a)
        minSum = min(minSum, curMin)
        total += a
    return max(total - minSum, maxSum) if maxSum > 0 else maxSum
```

If all numbers are negative, $\text{maxSum} = \max(A)$ and $\text{minSum} = \text{sum}(A)$. In this case, $\max(\text{maxSum}, \text{total} - \text{minSum}) = 0$, which means the sum of an empty subarray. According to the decription, We need to return the $\max(A)$, instead of sum of am empty subarray. So we return the maxSum to handle this corner case.

5.8.5 01 Knapsack problem

Problem description: given a knapsack that can load w weight and N things, every object has two properties, object i is $wt[i]$ and the value is $val[i]$, what's the most valuable things combination you could have given this knapsack?

First of all, two points must be clarified, **status** and **choice**. As long as the capacity of a few things and a backpack is limited, a backpack problem is formed. When it comes to status, it's easy to think about, for each item, what can you choose? The choice is to "pack in a backpack" or "do not pack in a backpack".

#474. Ones and Zeroes. 2 status inside: number of m and number of n. And the choices is form this word or not. The dp array here will be defined as given m zeros and n ones, what is the maximum number of strings that can formed.

```
def findMaxForm(self, strs, m, n):
    dp = [[0 for _ in range(n+1)] for _ in range(m+1)]

    for s in strs:
        # count zeros and ones
        zeros, ones = 0, 0
        for c in s:
            if c == '0':
                zeros += 1
            if c == '1':
                ones += 1

        # status and select
        for i in range(m, zeros-1, -1):
            for j in range(n, ones-1, -1):
                # choice: form this word or not
                # if form this word, the number of zeros in this word will be deducted
                dp[i][j] = max(dp[i - zeros][j - ones] + 1, dp[i][j])

    return dp[m][n]
```

5.8.6 Subset Knapsack problem

#416, given an non empty array with only positive numbers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

This problem is also a Knapsack problem, given a sum/2 weight knapsack with n number, is there a way to fulfill this knapsack. The dp array could be defined as $dp[i][j]$, where i denotes the first i numbers, and j denotes the volume of the knapsack. If $dp[i][j]$ is true then it can be fulfilled with the first i numbers. The states are those number and choices are choose the i number or not.

transition function then becomes:

- if $\text{num}[i]$ is not included, then $\text{dp}[i][j] = \text{dp}[i-1][j]$, same as the last one
- if $\text{num}[i]$ is included, then $\text{dp}[i][j] = \text{dp}[i-1][j-\text{nums}[i]]$

如果不把 `nums[i]` 算入子集，或者说你不把这第 `i` 个物品装入背包，那么是否能够恰好装满背包，取决于上一个状态 `dp[i-1][j]`，继承之前的结果。

如果把 `nums[i]` 算入子集，或者说你把这第 `i` 个物品装入了背包，那么是否能够恰好装满背包，取决于状态 `dp[i - 1][j-nums[i-1]]`。

Figure 5.23: Knapsack

```
def canPartition(self, nums):
    nums.sort()
    sum1 = sum(nums)
    aim = sum1/2
    if sum1%2 != 0:
        return False

    dp = [[0 for _ in range(aim+1)] for _ in range(len(nums) + 1)]
    dp[0][0] = True

    for i in range(1, len(nums) + 1):
        for j in range(1, aim+1):
            if nums[i - 1] > j:
                # not enough j, can't put the i number
                dp[i][j] = dp[i-1][j]
            else:
                # put in or not put in
                dp[i][j] = dp[i-1][j] or dp[i-1][j-nums[i-1]]

    return dp[-1][-1]
```

Notice that the $\text{dp}[i][j]$ only depends on the last state $\text{dp}[i-1][...]$, then we could do a space shrink, and j should be traversed backwards. The current position $\text{dp}[i][j]$ before actually is replied on the last state $+\text{nums}[i]$, if we do it forward, then $\text{dp}[i-2]$ will influence $\text{dp}[i-1]$, and then $\text{dp}[i]$ will also be influenced, which means a $\text{nums}[i]$ is used several times.

```
def canPartition(self, nums):
    nums.sort()
    sum1 = sum(nums)
```

```

aim = sum1/2
if sum1%2 != 0:
    return False

dp = [0 for _ in range(aim+1)]
dp[0] = True

for i in range(1, len(nums) + 1):
    for j in range(aim, -1, -1):
        if nums[i - 1] <= j:
            # put in or not put in
            dp[j] = dp[j] or dp[j-nums[i-1]]

return dp[-1]

```

5.8.7 Complete Knapsack Problem

The complete Knapsack problem is similar to the problems mentioned before, the only difference is that we could used the things or numbers infinite times.

#518, CoinChange 2, amount 5, coins=[1,2,5]. count how many times it has to sum up to 5 using the coins.

- states: amount and coins
- dp defined as $dp[num][amount]$, using only the first num of coin, the all possibilities.
- base cases: $dp[0][...] = 0$, $dp[...][0] = 1$

```

def change(self, amount, coins):
    if not coins and amount == 0:
        return 1

    dp = [[0 for i in range(amount+1)] for j in range(len(coins) + 1)]
    for i in range(len(coins)+1):
        dp[i][0] = 1

    for i in range(1, len(coins) + 1):
        for j in range(1, amount+1):
            # put it or not
            if j >= coins[i-1]:
                dp[i][j] = dp[i][j-coins[i-1]] + dp[i-1][j]
            else:
                dp[i][j] = dp[i-1][j]

    return dp[-1][-1]

```

```

def change(self, amount, coins):
    if not coins and amount == 0:
        return 1

    dp = [0 for i in range(amount+1)]
    dp[0] = 1

    for i in range(1, len(coins) + 1):
        for j in range(1, amount+1):
            # put it or not
            if j >= coins[i-1]:
                dp[j] = dp[j-coins[i-1]] + dp[j]

    return dp[-1]

```

5.8.8 Edit Distance

Given 2 string s1 and s2, calculate the fewest step that transform s1 to s2. The operations are insert, delete and replace. (Actually there is a hidden operation, that is skip)

To solve 2 strings problem, especially in DP, normally we will use 2 pointers. The pseudocode will be like

```

if s1[i] == s2[j]:
    skip
    i, j move forward
else:
    3 operations:
        |insert|: dp(i,j-1): s1 insert, then j-1 continue compare
        |delete|: dp(i-1,j): delete i and continue compare
        |replace|: dp(i-1,j-1): replace, but this position still here, move i and j forward

```

Initialization: if one string is empty and the other is not, then the min edit distance is the length of the non empty string.

```

def minDistance(self, s1, s2):
    n1 = len(s1)
    n2 = len(s2)
    dp = [[0 for i in range(n2 + 1)] for j in range(n1 + 1)]

    # base case
    for i in range(n1+1):
        dp[i][0] = i
    for j in range(n2+1):
        dp[0][j] = j

```

```

    for i in range(1, n1+1):
        for j in range(1, n2+1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1

    return dp[-1][-1]

```

5.8.9 Longest Common Subsequence(LCS)

#1143 Given two strings text1 and text2, return the length of their longest common subsequence. Using 2 pointers, if $s1[i] == s2[j]$, then this letter is definitely in LCS, otherwise, at least one of the letter inside s1 or s2 should be not in lcs. take the max one.

```

def longestCommonSubsequence(self, s1, s2):
    n1 = len(s1)
    n2 = len(s2)

    dp = [[0 for i in range(n2+1)] for j in range(n1+1)]

    # initialization one is empty then LCS is 0
    # dp[i][0] = 0   dp[0][j] = 0
    for i in range(1, n1+1):
        for j in range(1, n2+1):
            # found common
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[-1][-1]

```

5.8.10 Longest Palindromic Subsequence

#516 Given a string s, return the Longest Palindromic Subsequence. The template is similar to LCS, dp array defined as $dp[i][j]$: the LPS($nums[i]$ $nums[j]$). The reasoning of the transition function is illustrated below.

The 2 pointers i and j will go from 2 directions. i will go forward and j will go backward.

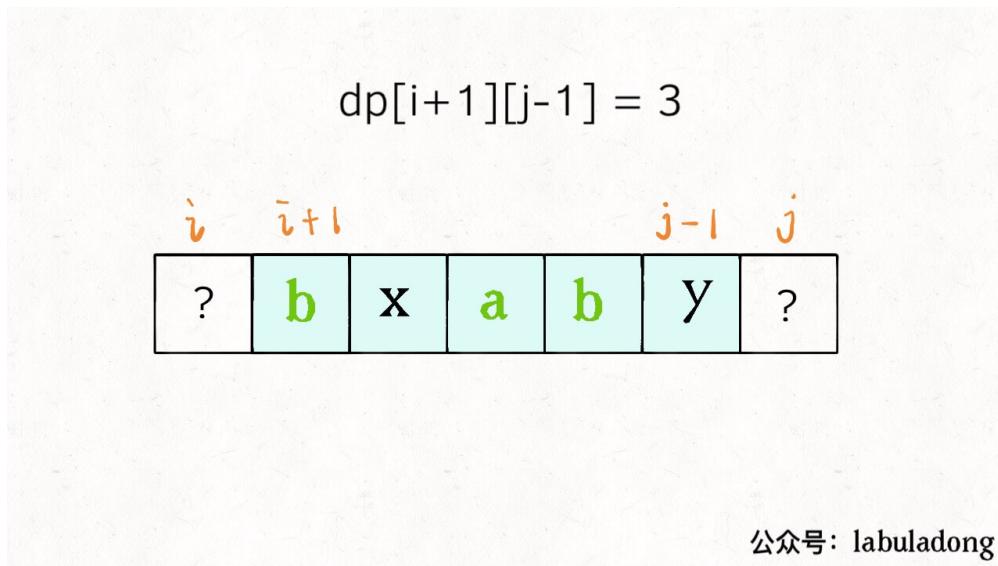


Figure 5.24: Ex1

If the two nums are the same, simply plus 2.

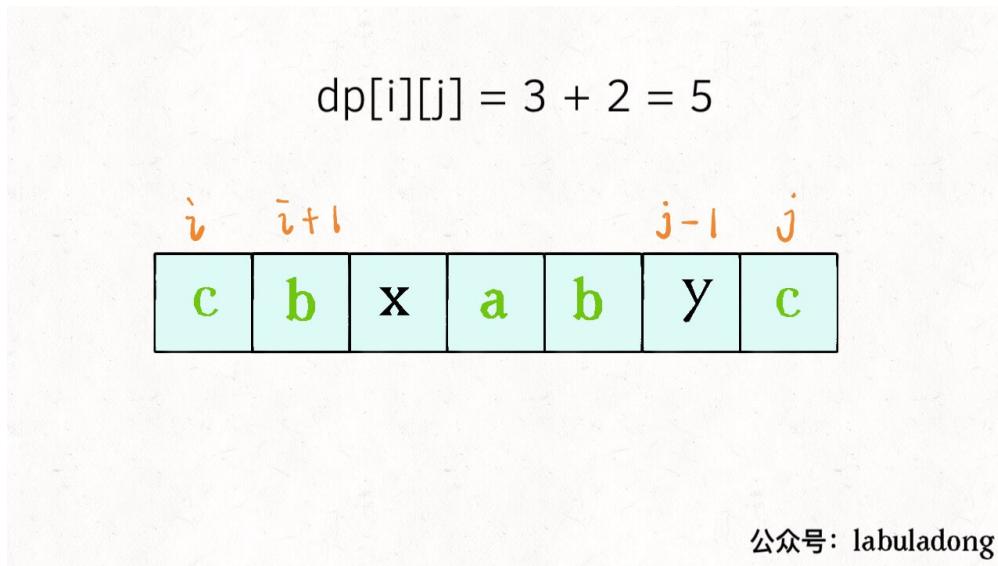


Figure 5.25: Ex2: when 2 nums equal

If not the same, take the maximum of one of them.

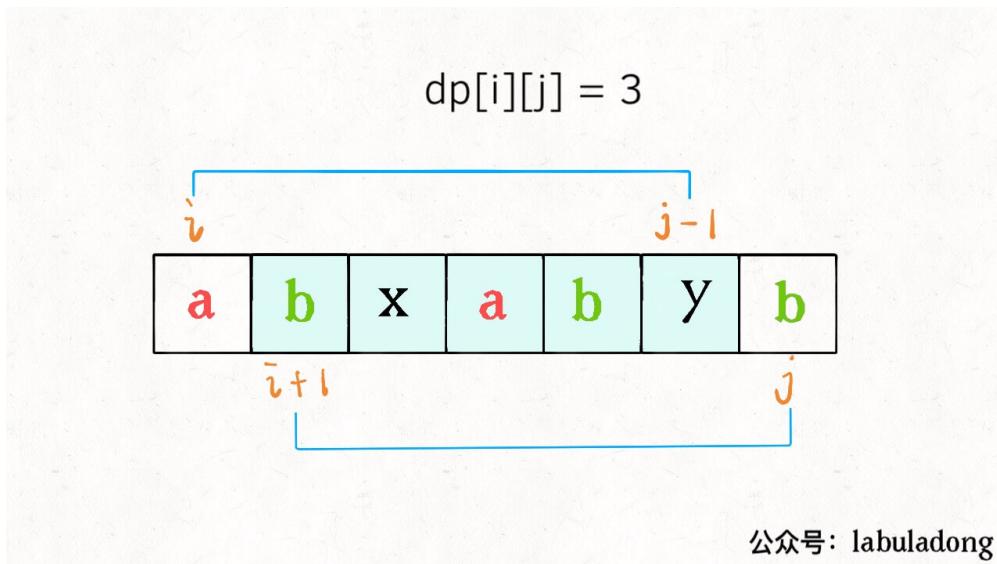


Figure 5.26: Ex3: When 2 nums not the same

Base Cases: If $i == j$, then only 1 elements. when $i < j$, there is no elements.

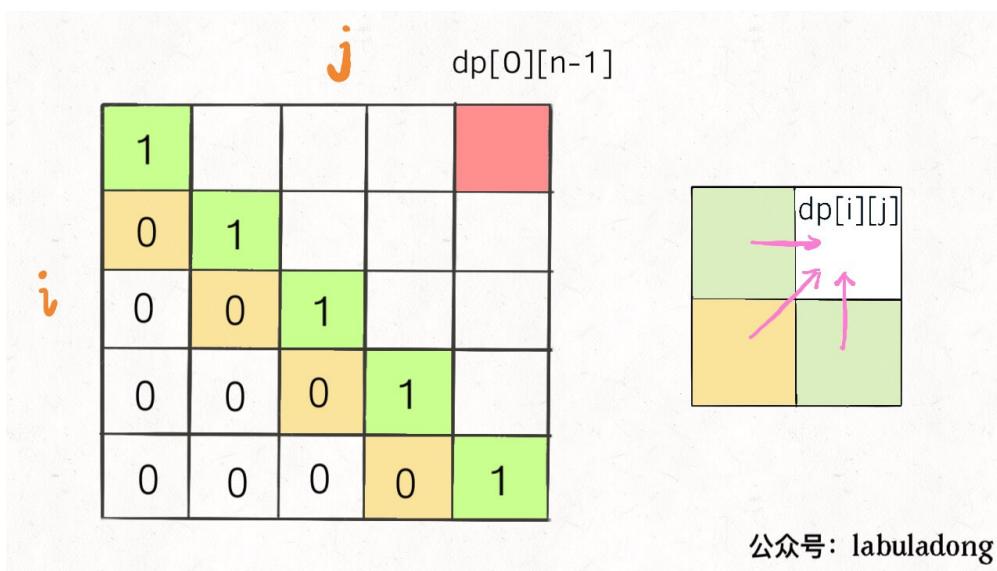


Figure 5.27: Base Cases

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

    dp = [[0 for _ in range(n)] for _ in range(n)]
    # base cases
```

```

for i in range(n):
    for j in range(n):
        if i == j:
            dp[i][j] = 1
        # to make sure dp[i][j] can be updated
        # i should be start from n-1
    for i in range(n-1,-1,-1):
        for j in range(i+1, n):
            if s[i] == s[j]:
                dp[i][j] = dp[i+1][j-1] + 2
            else:
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])

return dp[0][n-1]

```

5.8.11 Super Egg Drop

#887, You are given K eggs, and you have access to a building with N floors from 1 to N. You know that there exists a floor F with $0 \leq F \leq N$ such that any egg dropped at a floor higher than F will break, and any egg dropped at or below floor F will not break. Your goal is to know with certainty what the value of F is. What is the **minimum** number of moves that you need to know with certainty what F is

Actually this problem could be solved using Binary Search, but the problem is that the number of eggs is given, which means you can't do experiments whatever times you want.

- states: Number of eggs K and Number of floor N.
- Choices: which floor to drop the egg. The result after choosing this floor i also has 2 situations, the egg breaks or not. If the egg breaks, K-1, And the search space should be transformed from [1..N] to [1..i-1], if not, K. the space is then [i+1..N]

$$dp(K, N) = \min_{0 <= i <= N} \{ \max\{dp(K - 1, i - 1), dp(K, N - i)\} + 1 \}$$

Figure 5.28: Drop Egg Transition function

```

def superEggDrop(self, K, N):

    memo = collections.defaultdict()
    def dp(K, N):
        # base case

```

```

if K == 1: return N
if N == 0: return 0
#
if (K, N) in memo:
    return memo[(K, N)]

res = float('INF')
#
for i in range(1, N + 1):
    res = min(res,
               max(
                   dp(K, N - i),
                   dp(K - 1, i - 1)
               ) + 1
    )
#
memo[(K, N)] = res
return res

return dp(K, N)

```

This could be optimized with binary search:

```

memo = dict()
def dp(K, N):
    if K == 1: return N
    if N == 0: return 0
    if (K, N) in memo:
        return memo[(K, N)]

    # for 1 <= i <= N:
    #     res = min(res,
    #                max(
    #                    dp(K - 1, i - 1),
    #                    dp(K, N - i)
    #                ) + 1
    #     )

    res = float('INF')
    #
    lo, hi = 1, N
    while lo <= hi:
        mid = (lo + hi) // 2

```

```

broken = dp(K - 1, mid - 1) #
not_broken = dp(K, N - mid) #
# res = min(max() + 1)
if broken > not_broken:
    hi = mid - 1
    res = min(res, broken + 1)
else:
    lo = mid + 1
    res = min(res, not_broken + 1)

memo[(K, N)] = res
return res

return dp(K, N)

```

5.8.12 Burst Balloons

#312 Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get $\text{nums}[\text{left}] * \text{nums}[i] * \text{nums}[\text{right}]$ coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent. Find the maximum coins you can collect by bursting the balloons wisely.

312. 戳气球

难度 困难 271 收藏 分享 切换为英文 关注 反馈

有 n 个气球，编号为 0 到 $n-1$ ，每个气球上都标有一个数字，这些数字存在数组 `nums` 中。

现在要求你戳破所有的气球。每当你戳破一个气球 i 时，你可以获得 $\text{nums}[\text{left}] * \text{nums}[i] * \text{nums}[\text{right}]$ 个硬币。这里的 left 和 right 代表和 i 相邻的两个气球的序号。注意当你戳破了气球 i 后，气球 left 和气球 right 就变成了相邻的气球。

求所能获得硬币的最大数量。

说明:

- 你可以假设 $\text{nums}[-1] = \text{nums}[n] = 1$ ，但注意它们不是真实存在的所以并不能被戳破。
- $0 \leq n \leq 500, 0 \leq \text{nums}[i] \leq 100$

示例:

```
输入: [3,1,5,8]
输出: 167
解释: nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
      coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167
```

Figure 5.29: Burst Balloons Example

This is a hard problem simply because the sub problems are not independent. the score of bursting $\text{nums}[i]$ is related to $\text{nums}[i-1]$ and $\text{nums}[i+1]$. The first is to define dp array. To simplify this problem, add 2 element $\text{nums}[-1] = 1$ and $\text{nums}[n] = 1$.

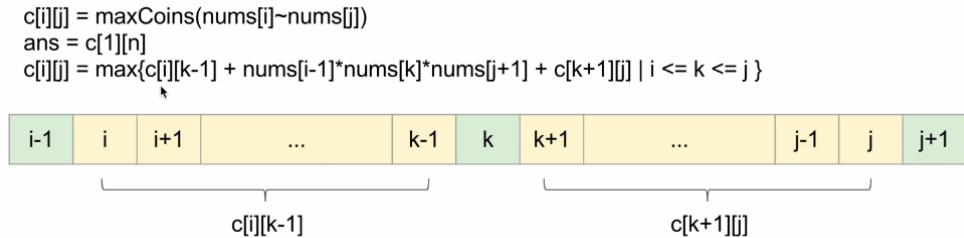


Figure 5.30: Burst Balloons TTransition Function

```
def maxCoins(self, nums):
    n = len(nums)
    nums.insert(0, 1)
    nums.append(1)
```

```

# dp[i][j] means maxCoin(nums[i]~nums[j])
c = [[0 for _ in range(n+2)] for _ in range(n+2)]

for l in range(1, n+1):
    # i could be in 1~n
    for i in range(1, n-l+2):
        j = i+l-1
        # k in between
        for k in range(i, j+1):
            c[i][j] = max(c[i][j], c[i][k-1] + nums[i-1]*nums[k]*nums[j+1] + c[k+1])

return c[1][n]

```

5.8.13 Stock Buying Selling Problem

Say you have an array for which the i th element is the price of a given stock on day i . If you were only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock), design an algorithm to find the maximum profit. A more general problem is given a list of prices, but the transition k could be more than 1, but in this case, only 1 transaction is allowed.

- States: The first is days, the second is the maximum transaction time k (in this case is 1). The third state is whether you have stock in your hand.
- Choices: Buy, Sell and hold(do nothing)

Base Case: Since we want to make this kind of problems more general to be solved, in the base case, therefore the detail will be elaborated. 1 in the 3 dimension means have stock, 0 stands for no stock

- $dp[-1][k][0] = 0$: i starts from 0. -1 means haven't started, initialize the profit to 0.
- $dp[-1][k][1] = \text{float}('-\infty')$: this situation is not possible, set it to inf because it is impossible to hold stock before the first day
- $dp[i][0][0] = 0$. Not possible to buy or sell.
- $dp[i][0][1] = \text{float}('-\infty')$: not possible

So the dp array could be defined as $dp[i][k][0$ or $1]$, i is the day, k is the transaction time. Then it is just to compare 3 different choices with max. For example, $dp[3][2][1]$ means on the third day, I have stock, and I could only transact 2 times at most.

The result we want to know is $dp[n-1][K][0]$.

```

1  dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
2          max( 选择 rest , 选择 sell )
3
4  解释：今天我没有持有股票，有两种可能：
5  要么是我昨天就没有持有，然后今天选择 rest，所以我今天还是没有持有；
6  要么是我昨天持有股票，但是今天我 sell 了，所以我今天没有持有股票了。
7
8  dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
9          max( 选择 rest , 选择 buy )
10
11 解释：今天我持有股票，有两种可能：
12 要么我昨天就持有股票，然后今天选择 rest，所以我今天还持有股票；
13 要么我昨天本没有持有，但今天我选择 buy，所以今天我就持有股票了。

```

Figure 5.31: Stock buy sell psudocode

```

# 121, k = 1:

def maxProfit(self, prices):
    """
    :type prices: List[int]
    :rtype: int
    """

    if not prices:
        return 0
    n = len(prices)
    dp = [[0 for _ in range(2)] for _ in range(n)]

    for i in range(n):
        if i-1 == -1:
            dp[i][0] = 0
            dp[i][1] = -prices[0]
            continue
        #dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
        #dp[i][1][1] = max(dp[i-1][1][1], dp[i-1][0][0] - prices[i])
        #           = max(dp[i-1][1][1], -prices[i])
        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
        dp[i][1] = max(dp[i-1][1], -prices[i])

    return dp[n-1][0]

```

By shrinking the space we can get:

```

def maxProfit(self, prices):

    if not prices:

```

```

        return 0
n = len(prices)
dp_0 = 0
dp_1 = float('-inf')
for i in range(n):
    dp_0 = max(dp_0, dp_1 + prices[i])
    dp_1 = max(dp_1, -prices[i])

return dp_0

#122

def maxProfit(self, prices):
    """
    :type prices: List[int]
    :rtype: int
    """
    n = len(prices)
    if not prices:
        return 0

    #dp=[[0 for i in range(2)] for j in range(n)]

    dp_0 = 0
    dp_1 = -prices[0]
    for i in range(n):
        temp = dp_0
        dp_0 = max(dp_0, dp_1 + prices[i])
        dp_1 = max(dp_1, temp - prices[i])

    return dp_0

#309

def maxProfit(self, prices):
    if not prices:
        return 0

    n = len(prices)
    dp = [[0 for _ in range(2)] for _ in range(n)]

    for i in range(n):
        if i - 1 == -1:

```

```

dp[i][0] = 0
dp[i][1] = -prices[0]
continue
if i - 2 == -2:
    dp[i][0] = 0
    dp[i][1] = -prices[0]
    continue
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
return dp[n-1][0]

def maxProfit(self, prices):
    if not prices:
        return 0

    n = len(prices)

    dp_0 = 0
    dp_1 = -prices[0]
    dp_pre = 0
    for i in range(n):
        temp = dp_0
        dp_0 = max(dp_0, dp_1 + prices[i])
        dp_1 = max(dp_1, dp_pre - prices[i])
        dp_pre = temp
    return dp_0

#714

def maxProfit(self, prices, fee):
    n = len(prices)
    if not prices:
        return 0

    dp_0, dp_1 = 0, float('-inf')
    for i in range(n):
        temp = dp_0
        dp_0 = max(dp_0, dp_1 + prices[i] - fee)
        dp_1 = max(dp_1, temp - prices[i])
    return dp_0

#123

def maxProfit(self, prices):

```

```

if not prices:
    return 0
n = len(prices)
dp = [[[0 for i in range(2)] for j in range(2+1)] for k in range(n)]

for i in range(n):
    for k in range(1, 3):
        if i == 0:
            dp[i][k][0] = 0
            dp[i][k][1] = -prices[0]
        continue

        dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
        dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);
return dp[n-1][2][0]

#188

def maxProfit(self, k, prices):
    if not prices:
        return 0
    n = len(prices)

    max_k = k
    if max_k > n / 2:
        return self.maxProfit_inf(prices)
    dp = [[[0 for _ in range(2)] for _ in range(k+1)] for _ in range(n)]
    for i in range(n):
        for k in range(1, max_k+1):
            if i == 0:
                dp[i][k][0] = 0
                dp[i][k][1] = -prices[0]
            continue
            dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
            dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
    return dp[n-1][k][0]

def maxProfit_inf(self, prices):
    n = len(prices)
    dp_0=0
    dp_1=-prices[0]
    for i in range(n):
        temp = dp_0
        dp_0 = max(dp_0, dp_1 + prices[i])

```

```

        dp_1 = max(dp_1, temp - prices[i])
    return dp_0

```

5.8.14 House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

- states: the index of the current house(element)
- choices: rob or not

dp array is defined as: $dp[i]=x$, with only 0-i element, what is the maximum profit.

#198

```

def rob(self, nums):
    if not nums:
        return 0
    n = len(nums)
    dp = [0 for i in range(n)]
    if n == 1:
        return nums[0]
    if n == 2:
        return max(nums[0], nums[1])
    # basecases
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])

    for i in range(2, n):
        dp[i] = max(dp[i-2]+nums[i], dp[i-1])

    return dp[-1]

```

After shrinking the space

```

def rob(self, nums):
    if not nums:
        return 0
    n = len(nums)
    if n == 1:
        return nums[0]
    if n == 2:
        return max(nums[0], nums[1])

```

```

# basecases
dp_2 = nums[0]
dp_1 = max(nums[0], nums[1])
for i in range(2, n):
    dp_i = max(dp_2+nums[i], dp_1)
    dp_2 = dp_1
    dp_1 = dp_i
return dp_i

```

#213 In this case, the selection becomes a cycle, then the figure shown below shows the situations. we only compare situation 2 and 3 since all of the elements are non-negative, situation 1 will therefore no need to compare with the other because it has fewer elements.

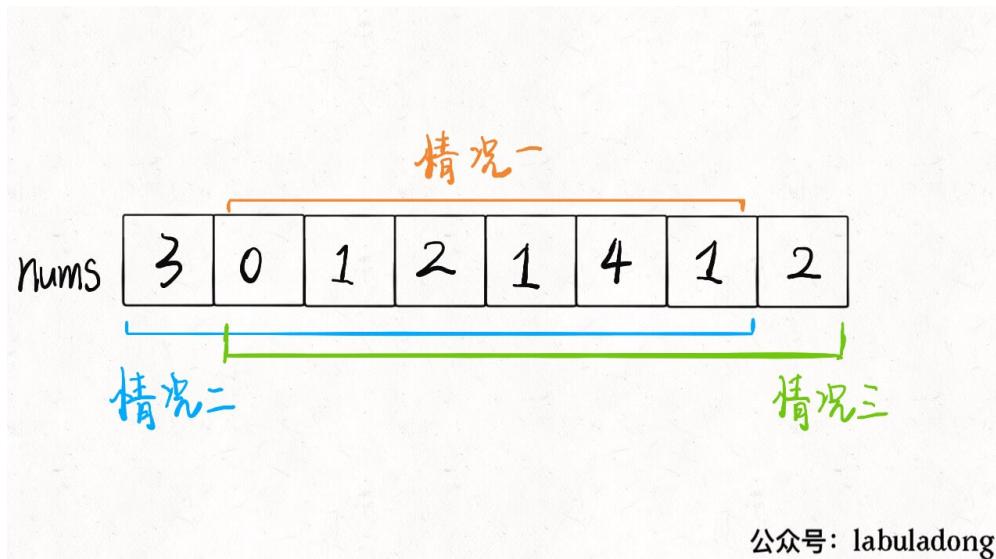


Figure 5.32: Robber 3

```

def rob(self, nums):
    if not nums:
        return 0
    n = len(nums)
    if n == 1:
        return nums[0]
    if n == 2:
        return max(nums[0], nums[1])

    return max(self.helper(nums[:-1]), self.helper(nums[1:]))

def helper(self, nums):

```

```

n = len(nums)
if n == 2:
    return max(nums[0], nums[1])
dp_2 = nums[0]
dp_1 = max(nums[0], nums[1])
dp_i = 0
for i in range(2, n):
    dp_i = max(dp_2+nums[i], dp_1)
    dp_2 = dp_1
    dp_1 = dp_i
# print(dp_i)
return dp_i

# 337

def rob(self, root):
    #Compare grandparent + max of grandchildren(l.l + l.r + r.l + r.r) vs max of
def dfs(root):
    if not root:
        return 0,0,0

    l,ll,lr = dfs(root.left)
    r,rl,rr = dfs(root.right)

    return max(root.val + ll + lr + rl + rr, l + r), l, r
return dfs(root)[0]

```

5.8.15 RegExp Match

Given an input string (s) and a pattern (p), implement regular expression matching with support for '.' and '*'.

- '.' Matches any single character.
- '*' Matches zero or more of the preceding element.

此外，这个问题也可以用动态规划的思路来解决。我们定义状态 $P[i][j]$ 为子串 $s[0, i)$ 和 $p[0, j)$ 是否匹配，能匹配为真，反之为假，然后状态转移方程则可以分为以下三种情况：

1. 如果 $p[j-1] \neq '*' \&& (s[i-1] == p[j-1] \text{ || } p[j-1] == '.')$ ，说明当前两个字符可以匹配且没有遇到 '*'，那么此时 $P[i][j] = P[i-1][j-1]$ ，也即若 $s[0, i-1)$ 和 $p[0, j-1)$ 匹配，则 $s[0, i)$ 和 $p[0, j)$ 也能匹配；
2. 如果 $p[j-1] == '*'$ ，说明当前字符为 '*'，并且我们用其匹配零个字符，那么 $P[i][j] = P[i][j-2]$ ，也即若 $s[0, i)$ 和 $p[0, j-2)$ 匹配，则跳过 $p[j-2], p[j-1]$ 两个元素后 $s[0, i)$ 和 $p[0, j)$ 也能匹配；
3. 如果 $p[j-1] == '*'$ ，说明当前字符为 '*'，并且我们用其匹配至少一个字符，而且还需满足 $s[i-1] == p[j-2] \text{ || } p[j-2] == '.'$ ，那么 $P[i][j] = P[i-1][j]$ ，也即若 $s[0, i-1)$ 和 $p[0, j)$ 匹配，那么我们用这个 '*' 来匹配 $s[i-1]$ 后， $s[0, i)$ 和 $p[0, j)$ 也就能匹配。至少一次是说这里 '*' 已经被用了一次，而前面 $s[0, i-1)$ 和 $p[0, j)$ 的匹配也有可能会用到。

其中 2 和 3 只需满足一个即可匹配，代码如下。

Figure 5.33: RegExp Transition Functions

```
def isMatch(self, s, p):
    n1 = len(s)
    n2 = len(p)

    if n1 != 0 and n2 == 0:
        return False

    dp = [[False for _ in range(n2+1)] for _ in range(n1 + 1)]
    dp[0][0] = True

    for j in range(2, n2+1):
        if p[j - 1] == '*':
            dp[0][j] = dp[0][j - 2]

    for i in range(1, n1 + 1):
        for j in range(1, n2 + 1):
            if p[j - 1] != '*':
                dp[i][j] = (p[j - 1] == '.' or p[j - 1] == s[i - 1])
                and dp[i - 1][j - 1]
            else:
                dp[i][j] = dp[i][j - 1] or dp[i][j - 2] or ((p[j - 2] == '.')
                    or p[j - 2] == s[i - 1]) and dp[i-1][j])

    return dp[-1][-1]
```

5.8.16 Knuth-Morris-Pratt (KMP)

Original Text: s, length: n1, Pattern sequence: p, length is n2. The KMP algorithm is to find substring p inside s. If not found return -1.

First, to understand what is KMP, it's necessary to figure out what is Partial Match Table (PMT). Look at the table below.

char:	a	b	a	b	a	b	c	a
index:	0	1	2	3	4	5	6	7
value:	0	0	1	2	3	4	0	1

Figure 5.34: PMT

The value inside is the PMT, how is it calculated? The first is to know prefix and suffix. For a string 'aba', the prefix set is 'a', 'ab'. And the suffix set is 'ba', 'a'. The intersection is a. Then the value is 1. So in the table above. The value is calculated like the intersection.

Having known the meaning of this table, how do we boost the search of finding a certain string?

5.9 Two Pointers Sliding Window Prefix Sum

Sliding window, generally, it is type of questions where you need to construct a window with l and r pointer to slide the array. Every iteration, r will go right, l pointer will go right based on condition. It is always related to subarray or substring questions.

The generation code skeleton will be like following:

```
def slidingwindow(self, nums):
    """
    :type nums: List[int]
    :rtype: List[int]
    """

    n = len(nums)
    l, res = 0, 0
    for r in range(n):
```

```

# do something
while condition l should go right:
    l += 1
# do something
# always res = min max
return res

```

Example: leetcode 1695 You are given an array of positive integers `nums` and want to erase a subarray containing **unique elements**. The score you get by erasing the subarray is equal to the sum of its elements.

Return the maximum score you can get by erasing exactly one subarray.

Here you want the subarray, the subarray need to be unique. then you need sliding to construct subarray and use a set to track the uniqueness, so we initialize a set `window` first. The condition for the left to move right is when the is the difficult part, every iteration, r go into the set window first, so the condition is actually when you see a new coming `nums[r]` already in our `window`, you need to remove `nums[l]` and do `l += 1`

```

def maximumUniqueSubarray(self, nums: List[int]) -> int:

    n = len(nums)
    m = 0
    res, l = 0, 0
    window = set()
    for r in range(n):
        # while condition not met
        while window and nums[r] in window:
            window.remove(nums[l])
            m -= nums[l]
            l += 1
        m += nums[r]
        window.add(nums[r])

        res = max(m, res)

    return res

```


Chapter 6

Tensorflow & PyTorch

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special contents, but the length of words should match the language.

Chapter 7

Spark

7.1 Rationale of using Spark

Engineered from the bottom-up for performance, Spark can be 100x faster than Hadoop for large scale data processing by exploiting in memory computing and other optimizations. Spark is also fast when data is stored on disk, and currently holds the world record for large-scale on-disk sorting. Therefore Spark is suitable for the big data processing.

Spark has easy-to-use APIs for operating on large datasets. This includes a collection of over 100 operators for transforming data and familiar data frame APIs for manipulating semi-structured data.

Spark comes packaged with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing. These standard libraries increase developer productivity and can be seamlessly combined to create complex workflows.

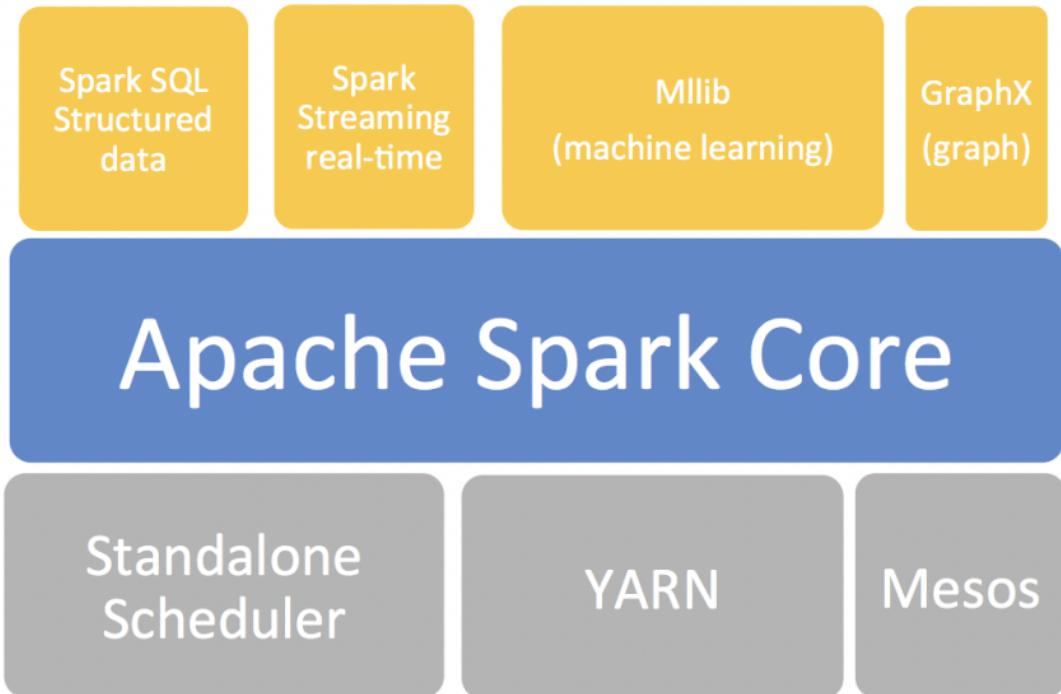


Figure 7.1: Spark Stack

7.2 Core Concept and How Spark Work

- Job: A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- Stages: Jobs are divided into stages. Stages are classified as a Map or reduce stages (It's easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- Tasks: Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- DAG: DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.
- Executor: The process responsible for executing a task.
- Master: The machine on which the Driver program runs
- Slave: The machine on which the Executor program runs

7.2.1 Spark Component

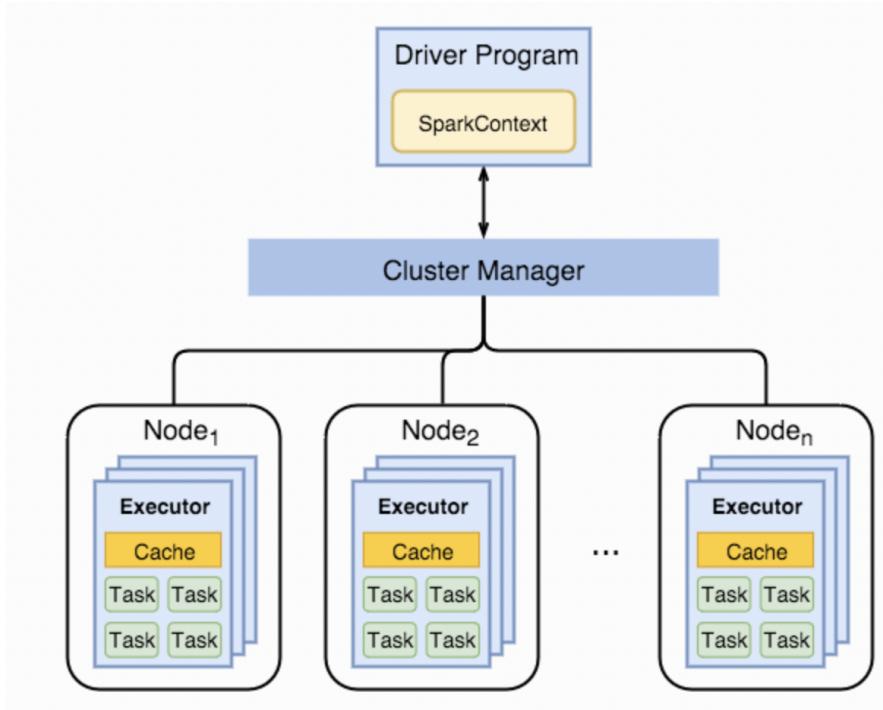


Figure 7.2: Spark Component

- Driver: Separate process to execute user applications, creates SparkContext to schedule jobs execution and negotiate with cluster manager
- Executors: run tasks scheduled by driver, store computation results in memory, on disk or off-heap and interact with storage systems
- Cluster Manager: Mesos, YARN, SparkStandalone

Spark Driver contains more components responsible for translation of user code into actual jobs executed on cluster.

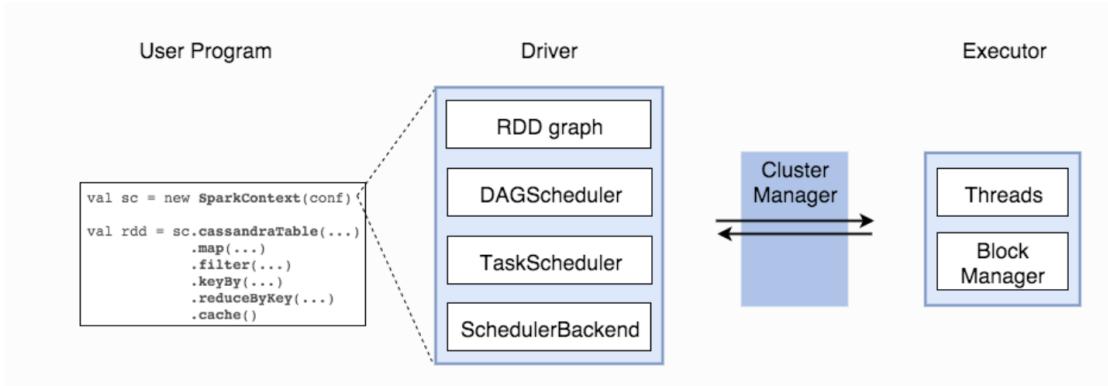


Figure 7.3: spark driver

- **SparkContext**
 - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster
- **DAGScheduler**
 - computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs
- **TaskScheduler**
 - responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers
- **SchedulerBackend**
 - backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)
- **BlockManager**
 - provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

7.2.2 Spark Architecture

Spark has a small code base and the system is divided in various layers. Each layer has some responsibilities. The layers are independent of each other.

The first layer is the interpreter, Spark uses a Scala interpreter, with some modifications. As you enter your code in spark console (creating RDD's and applying operators),

Spark creates a operator graph. When the user runs an action (like collect), the Graph is submitted to a DAG Scheduler. The DAG scheduler divides operator graph into (map and reduce) stages. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together to optimize the graph. For e.g. Many map operators can be scheduled in a single stage. This optimization is key to Sparks performance. The final result of a DAG scheduler is a set of stages. The stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager. (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies among stages.

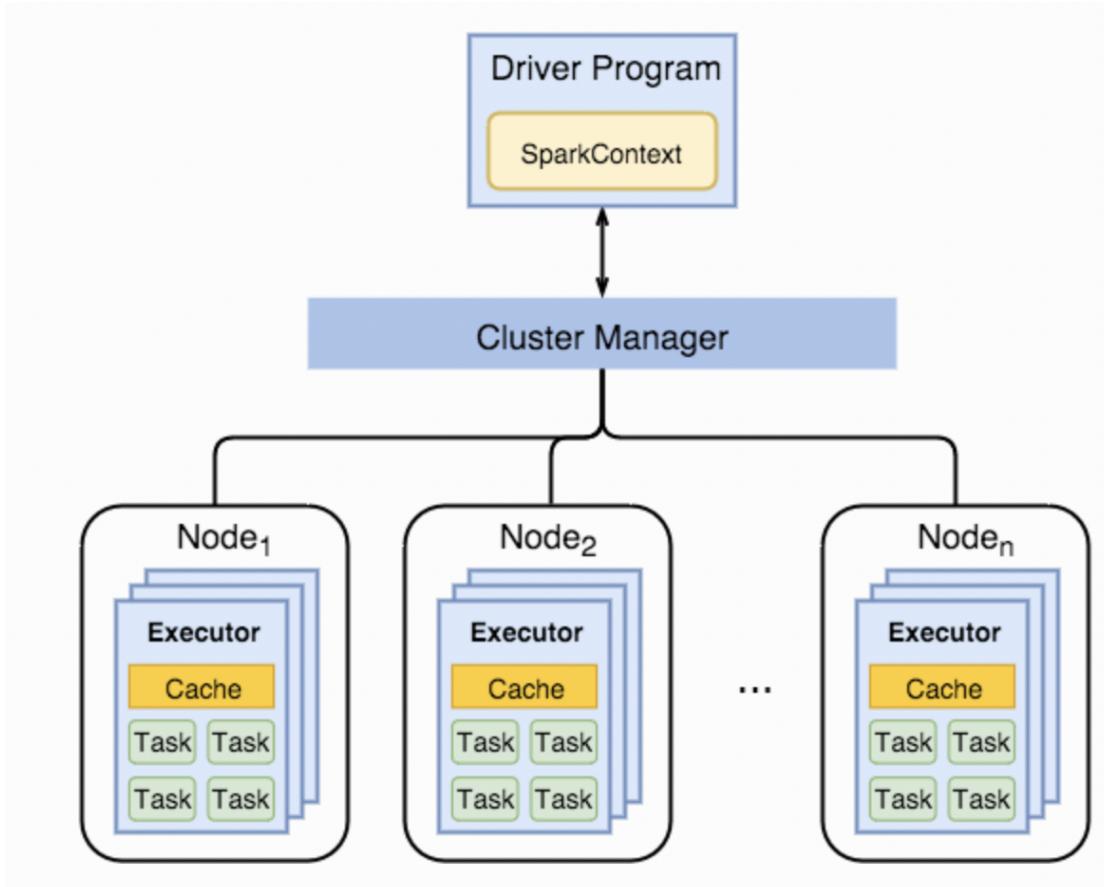


Figure 7.4: spark architecture

7.3 Spark Programming

7.3.1 Difference between SparkSession and SparkContext

Prior to spark 2.0.0, SparkContext was used as a channel to access all spark functionality. The spark driver program uses spark context to connect to the cluster through a resource

manager (YARN orMesos..).

SparkConf is required to create the spark context object, which stores configuration parameter like appName (to identify your spark driver), application, number of core and memory size of executor running on worker node

In order to use APIs of SQL, HIVE , and Streaming, separate contexts need to be created.

```
val conf = new SparkConf()  
  
val sc = new SparkContext(conf)  
  
val hc = new hiveContext(sc)  
  
val ssc = new streamingContext(sc).
```

SPARK 2.0.0 onwards, SparkSession provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with Dataframe and Dataset APIs. All the functionality available with sparkContext are also available in SparkSession.

In order to use APIs of SQL, HIVE, and Streaming, no need to create separate contexts as sparkSession includes all the APIs.

Once the SparkSession is instantiated, we can configure Spark's run-time config properties.

Chapter 8

Docker & Kubernetes

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

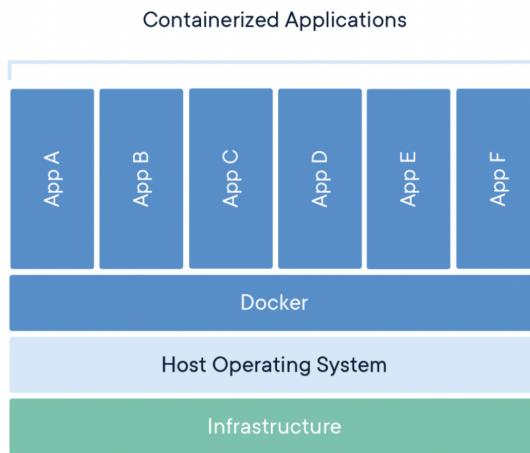


Figure 8.1: Docker in the OS

The operating system support of Virtual machine and Docker container is very different. From the image above, you can see each virtual machine has its guest operating system above the host operating system, which makes virtual machines heavy. While on the other hand, Docker containers share the host operating system, and that is why they are lightweight.

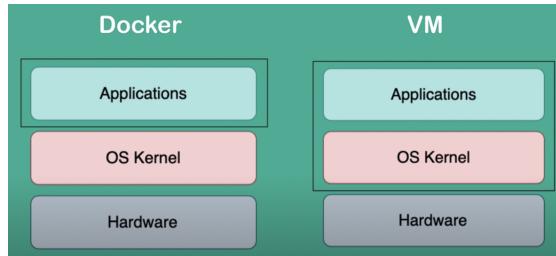


Figure 8.2: Docker v.s. Virtual Machine

8.1 Useful commands

The docker commands could be found in docker doc or docker cheatsheet [pdf]

Most of your images will be created on top of a base image from the Docker Hub registry. Docker Hub contains many pre-built images that you can pull and try without needing to define and configure your own. To download a particular image, or set of images (i.e., a repository), use docker pull

\$ docker pull [OPTIONS] NAME[:TAG|@DIGEST]

List all images that are locally stored with the Docker Engine

\$ docker image ls

List the running containers (add –all to include stopped containers)

\$ docker container ls

Delete an image from the local image store

\$ docker image rm alpine:3.4

The docker run command first creates a writeable container layer over the specified image, and then starts it using the specified command. That is, docker run is equivalent to the API /containers/create then /containers/(id)/start. A stopped container can be restarted with all its previous changes intact using docker start. See docker ps -a to view a list of all containers. You could also use –name option to specify the container name.

\$ docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

An example could be run container in background and print container ID and bind the container's port (see from docker ps -a) and your own OS's port

\$ docker run -p 127.0.0.1:80:8080/tcp -d redis

Show all the running and stopped containers:

```
$ docker ps -a
```

Stop or start a container, to specify a container, you could find the container id from docker ps.

```
$ docker stop [OPTIONS] CONTAINER [CONTAINER...]
$ docker start [OPTIONS] CONTAINER [CONTAINER...]
```

List port mappings or a specific mapping for the container

```
$ docker port CONTAINER [PRIVATE_PORT[/PROTO]]
```

In order to debug our container, there are 2 useful commands we could use, the first one is log, for example, we want to print the last 100 lines of a container's logs

```
$ docker logs --tail 100 web
```

The docker exec command runs a new command in a running container.

```
$ docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

An example could be execute commands in container 'web', since this is a lightweight execution bash session, so you won't have all of the command (e.g. curl), but simple ls and cd are supported.

```
$ docker exec -it web /bin/bash
```

8.2 Docker compose

The docker compose file (.yml or .yaml) is a way for us to configure everything we want to run using docker run command, this Compose file is super convenient as we do not have to type all the parameters to pass to the docker run command. We can declaratively do that using a Compose file.

The figure 8.3 shows the mapping from docker run and .yml file. The another good thing about using the compose file is that if you are going to run 2 container in the same network, in the original docker run function, you need to specify -net option and link between this two, but in the compose file it'll create these 2 container within the same network by default.

The command to use is:

```
$ docker compose [-f <arg>...] [options] [COMMAND] [ARGS...]
```

Example:

```
$ docker compose -f docker-compose.yml
```

docker run command	mongo-docker-compose.yaml
<pre>docker run -d \ --name mongodb \ -p 27017:27017 \ -e MONGO-INITDB_ROOT_USERNAME \ =admin \ -e MONGO-INITDB_ROOT_PASSWORD \ =password \ --net mongo-network \</pre>	<pre>version: '3' services: mongodb: image: mongo ports: - 27017:27017 environment: - MONGO..._USERNAME=admin</pre> 

Figure 8.3: docker compose

8.3 Docker workflow & Develop with docker

An good example of how to develop with docker could be found in the doc, The step could be as followed:

- Create a sample Python application
- Create a new Dockerfile which contains instructions required to build a Python image
- Build an image and run the newly built image as a container
- Set up volumes and networking
- Orchestrate containers using Compose
- Use containers for development
- Configure a CI/CD pipeline for your application using GitHub Actions
- Deploy your application to the cloud

The graph 8.4 shows the workflow with docker mainly in the CICD workflow.

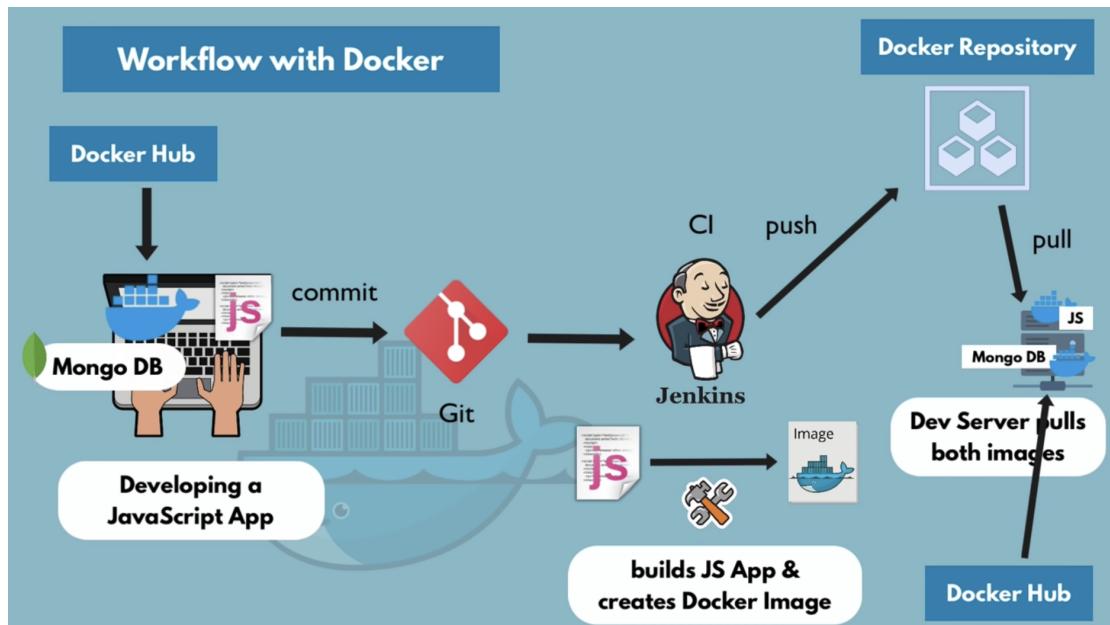


Figure 8.4: Docker CICD work flow

8.4 Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using **docker build** users can create an automated build that executes several command-line instructions in succession.

Image Environment Blueprint	DOCKERFILE
install node	FROM node
set MONGO_DB_USERNAME=admin	ENV MONGO_DB_USERNAME=admin \
set MONGO_DB_PWD=password	MONGO_DB_PWD=password
create /home/app folder	RUN mkdir -p /home/app
copy current folder files to /home/app	COPY . /home/app
start the app with: "node server.js"	CMD ["node", "server.js"]

Figure 8.5: An dockerfile example

8.5 Private Docker Repository**8.6 Deploy and persistence**

Chapter 9

Git

Git is a **distributed version control** system for tracking changes in source code during software development. Now, what is a distributed version control system? **Version control** is basically a system that records changes to a file or set of files over time so that you can recall specific versions later. And **distributed** means that complete codebase — including its full version history — is mirrored on every developer's computer.

The notes only contains important git command and some explanation in the workflow, therefore, it assumes that you have basic understanding of the feature in git.

9.1 Git configuration

Git config is used to set our username, email, password, credential way and so on, this is the first step we will do before start using git. There are 3 levels of config in this command.

```
$ git config [--local|--global|--system]
```

- By default, git config will write to a local level if no configuration option is passed. Local level configuration is applied to the context repository git config gets invoked in. Local configuration values are stored in a file that can be found in the repo's .git directory: .git/config
- Global level configuration is user-specific, meaning it is applied to an operating system user. Global configuration values are stored in a file that is located in a user's home directory. ~/.gitconfig on unix systems.
- System-level configuration is applied across an entire machine. This covers all users on an operating system and all repos. The system level configuration file lives in a gitconfig file off the system root path. \$(prefix)/etc/gitconfig on unix systems.

Config the username and user email

```
$ git config --global user.name "Liangliang Zheng"  
$ git config --global user.email liangliangzheng@example.com
```

9.2 Basic Commands

First, to get started, you need to run the command `git init`. Git will create a hidden `.git` directory and use it for keeping its files organized in other subdirectories.

```
$ git init
```

Before sharing the file we have to select the file to be shared and to select the files we use the command `git add`.

```
$ git add file_name
```

To save the selected file in we use the `git commit` command. A snapshot is created when we make a commit. Saving the changes = Making a commit. To edit the commit message we can use the `--amend` option.

```
$ git commit [--amend] -m "commit message"
```

To check the status of the files we use the `git status` command. It will show the untracked, unmodified, modified, etc files.

```
$ git status
```

To view the commit history we can use the `git log` command.

```
$ git log
```

Adding a remote repository:

- Repository: A repository is like a folder for your project. Your project's repository will contain all of your project's files.
- Remote Repositories: Remote repositories are versions of your project that are hosted online/cloud.

Git uses the remote repository to store the shared file. To add a remote repository `git remote add` command is used and for that, a remote URL is also required. We can easily get the remote URL from hosting platforms like GitHub. And we could have several remote repository by using `add remote url`

```
$ git remote add remote_name URL
```

You could view the remote repo detail by using:

```
$ git remote -v
```

Change remote name:

```
$ git remote rename current_name changed_name
```

Uploading files to a remote After making the changes to the repository we need to upload the files so that developers can work on it. And to upload the files to a remote we use the **push** command. To get the latest copy of the repository we download the files. To download the files from a remote we can use the **pull** command.

```
$ git [push|pull] remote_name branch_name
```

9.3 Git branch

List all the local branches, option -r could list all the remote branches. -a could list all the local and remote branches.

```
$ git branch
```

Create a new branch [-b] and change to the branch

```
$ git checkout -b [branch]
```

Create a new branch and point to a specific commit.

```
$ git branch [branch] [commit]
```

The merge command in git could be read as merge from, means merge a branch to our current checked out branch.

```
$ git merge [branch]
```

Git merge with no fast forward option, The --no-ff flag prevents git merge from executing a "fast-forward" if it detects that your current HEAD is an ancestor of the commit you're trying to merge. A fast-forward is when, instead of constructing a merge commit, git just moves your branch pointer to point at the incoming commit. This commonly occurs when doing a git pull without any local changes.

```
$ git merge --no-ff myfeature
```

Cherry pick one to several commit and merge into current branch.

```
$ git cherry-pick [commit]
```

Delete a branch.

```
$ git branch -d [branch-name]
```

Delete a remote branch. Could do a push operation.

```
$ git push origin --delete [branch-name]
```

9.4 Git tag

Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1.0.1). A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits. It is used after git commit.

List all the tags

```
$ git tag
```

Create a tag in the current commit.

```
$ git tag [tag]
```

Create a tag in a specific commit.

```
$ git tag [tag] [commit]
```

Delete a local tag

```
$ git tag -d [tag]
```

Delete a remote tag

```
$ git push --delete origin tagname
```

Show information of a tag

```
$ git show [tag]
```

Push a tag

```
$ git push [remote] [tag]
```

9.5 Git reset

There are moments you want to roll over your commit HEAD back to a specific commit. git reset could come in handy when you want to revert your changes.

reset the current HEAD back to n history commits.

```
$ git reset --hard [HEAD~n]
```

reset the current HEAD to a commit

```
$ git reset [commit]
```

9.6 Git workflow

In 9.1, we showed a classical git branching model that used in industry. This section is used from A successful Git branching model

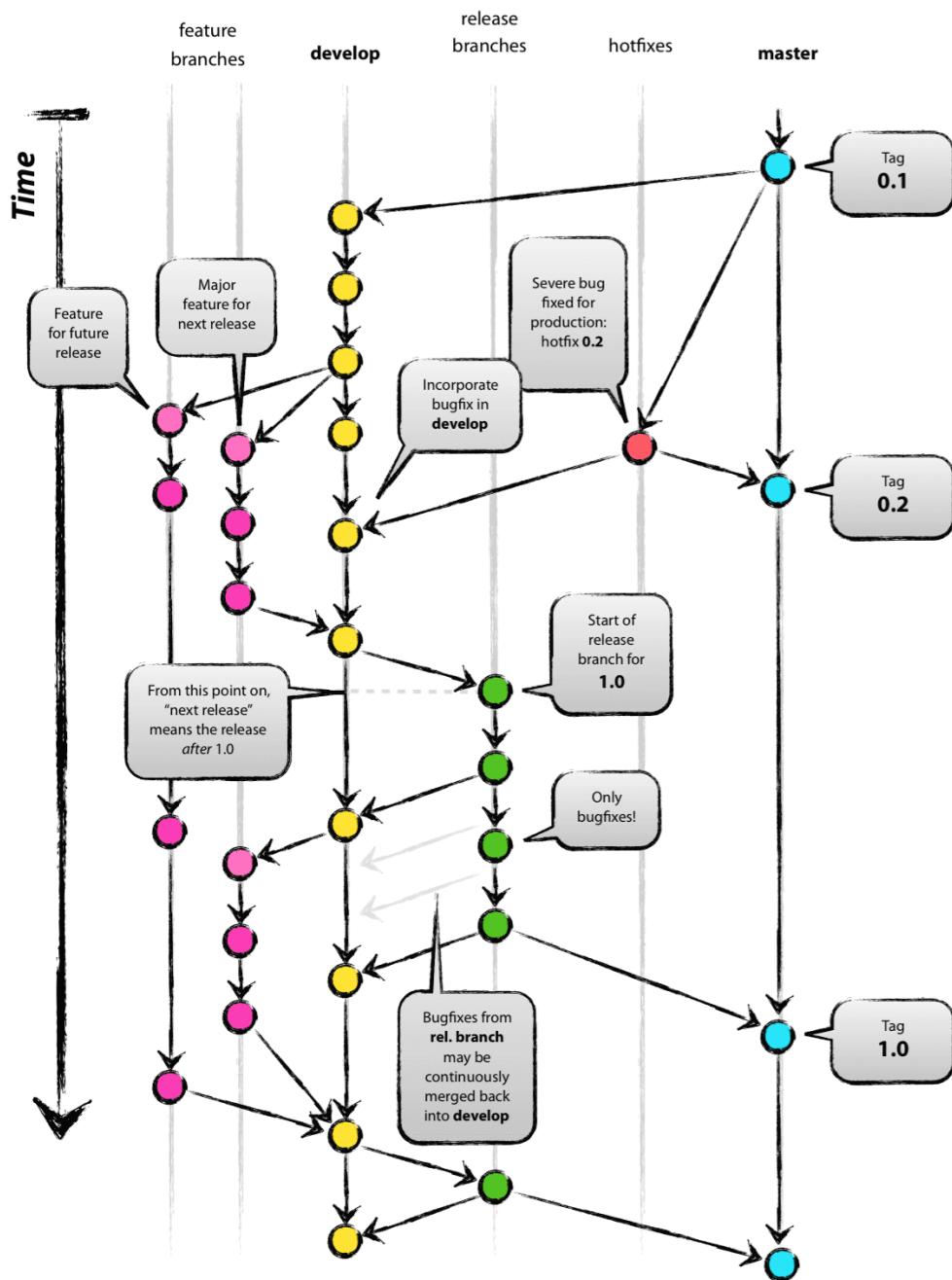


Figure 9.1: Git branching model

Different features will be implemented in different feature branches. After a specific time sprint, corresponding features that have been implemented should be merged into **develop** and then this new features will be created as a new release in the **release branches**. After the releasing test, we could merge this into **master** and tagging in **master** branch.

If we found serious bug in master branch, then hotfix could be used to fix bugs. After the bug fixed, merge it back to master create a new tag. Also the fix part should be merged into develop branch.

Branch Category:

- Master Branch: Same as the code in production and has tagged, each tag could be stably released.
- Hotfix Branch: Used for fixing bug.
- Release Branch: Used for testing before putting into production (master) branch.
- Develop Branch: Used for merging all the feature created from feature branch.
- Feature Branch: Used for feature implementation.

Flow: First create and checkout a feature branch based on develop.

```
$ git checkout -b myfeature develop
```

9.6.1 Git Flow Step

The Finished features may be merged into the develop branch to definitely add them to the upcoming release:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.

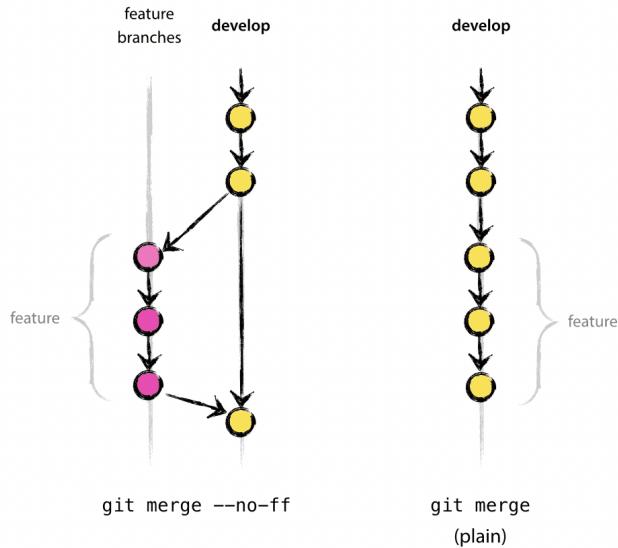


Figure 9.2: Git no-ff comparison

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of develop is ready for the “next release” and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

bump-version.sh is a fictional shell script that changes some files in the working copy to reflect the new version. (This can of course be a manual change—the point being that some files change.) Then, the bumped version number is committed.

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into master (since every commit on master is a new release by definition, remember). Next, that commit on

master must be **tagged** for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into develop, so that future releases also contain these bug fixes.

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2
```

To keep the changes made in the release branch, we need to merge those back into develop, though. In Git:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```

Now we are really done and the release branch may be removed, since we don't need it anymore:

```
$ git branch -d release-1.2
```

Hotfix branches May branch off from master and Must merge back into develop and master Branch naming convention hotfix-*.

Hotfix branches are created from the master branch. For example, say version 1.2 is the current production release running live and causing troubles due to a severe bug. But changes on develop are yet unstable. We may then branch off a hotfix branch and start fixing the problem:

```
$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
$ ./bump-version.sh 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

Then, fix the bug and commit the fix in one or more separate commits.

When finished, the bugfix needs to be merged back into master, but also needs to be merged back into develop, in order to safeguard that the bugfix is included in the next release as well. This is completely similar to how release branches are finished.

First, update master and tag the release.

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2.1
```

Now update it in develop branch

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
```

The one exception to the rule here is that, when a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of develop

Finally, remove the temporary branch:

```
$ git branch -d hotfix-1.2.1
```


Chapter 10

SQL

Chapter 11

DevOps

11.0.1 Basic Intro

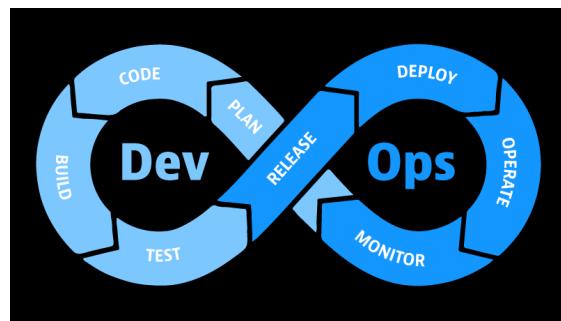


Figure 11.1: DevOps Cycle

- **Plan:** This is an opportunity as a DevOps Engineer for you to get involved with that and learn what kinds of things are going to be coming your way that you need to be involved with also to work with your customer before they go down a bad path.
- **Develop(Code):** Developing your feature, merge code. (Where Continuous Integration appear, avoid merge hell!)
- **Build:** Kick off the first of our automation processes because we're going to take their code and we're going to build it depending on what language they're using it may be transpiling it or compiling it or it might be creating a docker image from that code either way we're going to go through that process using our CI CD pipeline.
- **Testing:** Once we've built it we're going to run some tests on it now the development team usually writes the test you may have some input in what tests get written but we need to run those tests and the testing is a way for us to try and

minimise introducing problems out into production, it **doesn't guarantee** that but we want to get as close to a guarantee as we can that were one not introducing new bugs and two not breaking things that used to work

- **Release:** You know the code may just live in the GitHub repo or the git repository or wherever it lives but it may be the process of taking your compiled code or the docker image that you've built and putting it into a registry or a repository where it's accessible by your production servers for the deployment process.
- **Deploy:** which is the thing that we do next because deployment is like the end game of this whole thing because deployments when we put the code into production and it's not until we do that that our business actually realizes the value from all the time effort and hard work that you and the software engineering team have put into this product up to this point.

Continuous Delivery: Continuous Delivery = Plan ↴ Code ↴ Build ↴ Test

Continuous Integration: This is effectively the outcome of the Continuous Delivery phases above plus the outcome of the Release phase. This is the case for both failure and success but this is fed back into continuous delivery or moved to Continuous Deployment.

Continuous Integration = Plan ↴ Code ↴ Build ↴ Test ↴ Release

Continuous Deployment: If you have a successful release from your continuous integration then move to Continuous Deployment which brings in the following phases

CI Release is Success = Continuous Deployment = Deploy ↴ Operate ↴ Monitor