Dynamic Programming.

Longest Increasing Subsequence. (LIS).

---

e.g.

Index.	0	1	2	3	4.
Nums	1	4	3	4	2

dp[3] = 3
⇓
1 → 3 → 4
⇓
longest = 3

dp[4] = 2
⇓
1 → 2
⇓
longest = 2.

Index.	0	1	2	3	4.	5
Nums	1	4	3	4	2	3
dp	1	2	2	3	2	?

How could we find dp[5] based on prev?

Similar to Dijkstra's shortest path algo.

We search for Num[i] < Num[5]
⇓
3

with max && dp[i].

⇒ when i = 0, i = 4

Num[0] = 1 Num[4] = 2

dp[0] = 1 dp[4] = 2

so dp[5] = 3 by Num[4]

Convert the logic into pseudo code:

Index.	0	1	2	3	4	5
Nums	1	4	3	4	2	3
dp	1	2	2	3	2	?

(j: 0–4, i: 5)

Let i to be result index. j to be prev index.

```
for (int j=0 ; j < i ; j++) {
    if (Nums[j] < Nums[i]) {
        dp[i] = Math.max(dp[i], dp[j]+1);
    }
}
```

Adding itself

↳ compare to the best result so far.

Then the logic can be combined:

```
for (int i=0 ; i < num.side() ; i++) {
    prevCode;
}
```

Then loop thro dp[i], return max dp[i].

Time Complexity is $O(n^2)$.

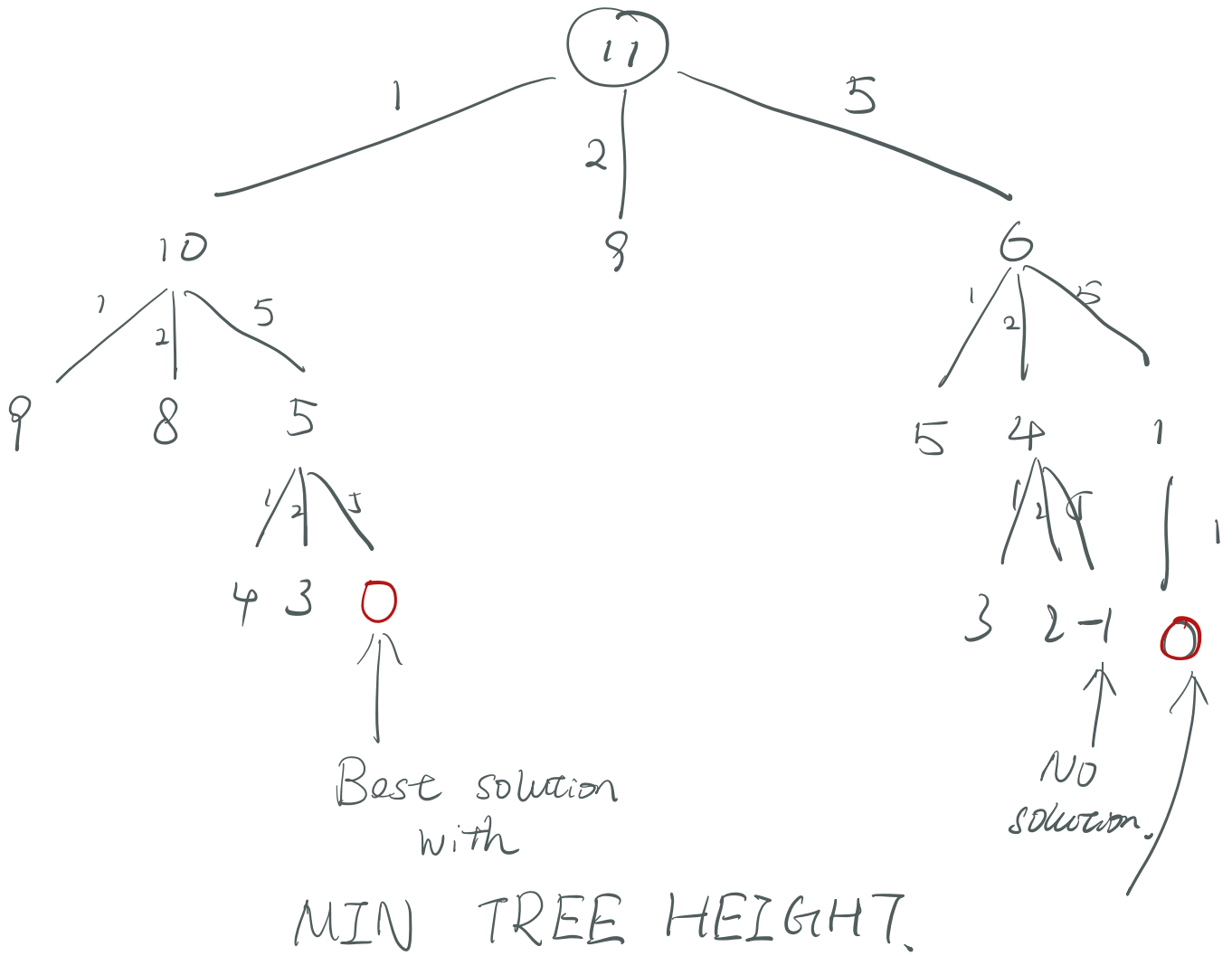
Given k numbers of coin type, they're $C_1, C_2, \cdots C_k$.

Asking for min amount of coins to get to total. ∞

e.g. k = 3. $\begin{matrix}1\\2\\5\end{matrix}$ 零錢問題.

Asking amount = 11.

∴ 3 coins ⇒ 5+5+1 = 11.



MIN TREE HEIGHT.

$$dp(n) \begin{cases} 0, & n = 0 \\ -1, & n < 0 \quad (\text{No solution}). \\ \min\{dp(n-coin)+1 \mid coin \in coins\}, & n > 0. \end{cases}$$

We could also use table to reduce some unnecessary calculation,
similar to fib number solution.

背包问题.

- A bag can load

  ↙ OR ↘

  W Weight N object

- Now we have

object	Weight	Value
i	Wt[i]	Val[i]

---

e.g. bag can load < 3 object
4 weight.

i	W[i]	V[i]
1	2	4
2	1	2
3	3	3

Should we consider value per weight. As factor?

SO we will choose i=1, i=2 into bag, total value of 6. = Value_MAX

① Status < Capacity.
Objects.

② Choice < object go in to the bag.
Object NOT go in to the bag.

Structure for dp:

```
for statusAIndex in allStatusA {
    for statusBIndex in allStatusB {

        dp[statusAI][statusB] = Best(choiceA, choiceB.);

    }
}
```

base case { dp[0][...] = 0
dp[...][0] = 0.

In this case:

```
for i in [1 ... n]:
    for w in [1 ... W]:

        dp[i][w] = max(把 i 装进背包, 不把 i 装进背包)
                    //                  //

return dp[N][W];
```

What is dp[i][w] ?

对于前 i 个物品，当前背包容量为 w 时，

该情况下，可以装下的最大价值为 dp[i][w].

If we want object i into bag.

then maxValue must contain val[i].

Then the capcity of bag is

W - wt[i]

We want to put maxValue objects into the rest of this capacity.

Hence,

val[i] + dp[i-1][w - wt[i]]

dp[v-1][w].

e.g. Same example as above,

if I don't want the third object in bag,

so the max value in bag is when first two objects in bag.

considering.

dp[i][w]

=

Put in bag. ↙ OR ↘ NOT Put in bag.

val[i-1] + dp[i-1][w - wt[i-1]] dp[v-1][w].

i-1 because 0 is based case, where bag is empty,

so in actual case, we need to Shift 1 for weight and value.

Balanced Partition Problem. (分割等和子集).

Given set [1, 5, 11, 5]

separate into two equal size sets.

Return [1, 5, 5] & [11].

This is another type of backpack question.

the capacity of the bag is sum/2.

Edit distance problem. (編輯距離)

Q:

Change the word1 to word2 with min operation.

Operations: Enter / delete / replace an character.

e.g. word1 = "horse" word2 = "ros"

Return : 3 ⇒ horse → rorse
rorse → rose
rose → ros

A: Normally when we see comparing two strings,
we will use two pointers.

S1 r a d e
S2 a p p l e

i at d, j at l
S1[i] != S2[j]
⇓ operation.
insert e

S1 r a d l e
S2 a p p l e

i at d, j at l
S1[i] != S2[j]
⇓ operation.
insert l

S1 r a d p l e
S2 a p p l e

i at d, j at p
S1[i] != S2[j]
⇓ operation.
replace d to p

S1 r a p l e
S2 a p p l e

i at a, j at p
S1[i] != S2[j]
⇓ operation.
insert p
and
delete r.

5 times operation.

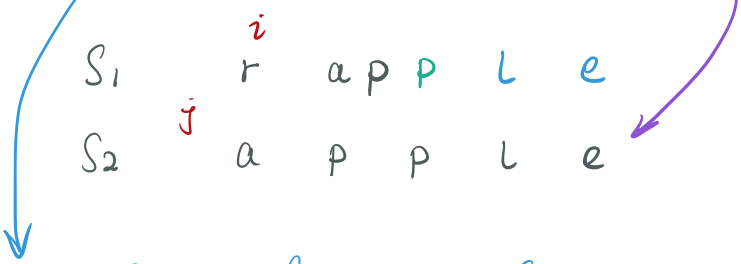
Status < the position of i j.

Choice < skip, insert, delete, replace

definition : dp[i][j] will return min edit distance for $S_1[0..i]$ & $S_2[0..j]$

base case : $i == -1$ return $j+1$

$j == -1$ return $i+1$

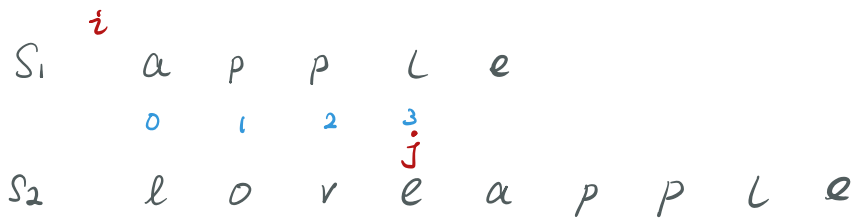


if j reaches -1, which means comparison finished.

all characters at and at the left of i must be deleted.

the operations to delete these are $i+1$, because index 0 is a number.

if S_1 is finished, but S_2.



We need to have $4 = (j+1)$ operations to insert these characters.

```python
def minDistance(s1, s2) -> int:
    # 定义：dp(i, j) 返回 s1[0..i] 和 s2[0..j] 的最小编辑距离
    def dp(i, j):
        # base case
        if i == -1: return j + 1
        if j == -1: return i + 1

        if s1[i] == s2[j]:
            return dp(i - 1, j - 1)  # 啥都不做 ✓
        else:
            return min(
                dp(i, j - 1) + 1,     # 插入
                dp(i - 1, j) + 1,     # 删除
                dp(i - 1, j - 1) + 1 # 替换
            )

    # i, j 初始化指向最后一个索引
    return dp(len(s1) - 1, len(s2) - 1)
```

similar to insertion as explained below.

0 1 2 3 4
apple size = 5.
that's why -1.

Insertion is inserting another character next to current index.

```
        i
S1  r a d l e        S1[i] != S2[j]
          j                  ↓ operation.
S2  a p p l e          insert l
```

so index i cannot more since not matched yet,

j more forward, resulting dp(i, j-1) + 1

adding one operation

However similar to all other dp question,

which will repeatly calculate some value. (重叠子问题).

There are two ways of solving < memo / dp table.

Memo (dictionary):

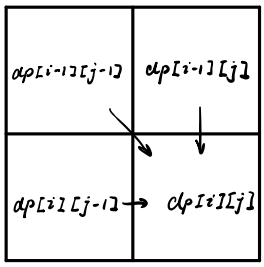
Simply store the info in memo, then check memo each time.

```python
def minDistance(s1, s2) -> int:

    memo = dict() # 备忘录
    def dp(i, j):
        if (i, j) in memo:
            return memo[(i, j)]
        ...

        if s1[i] == s2[j]:
            memo[(i, j)] = ...
        else:
            memo[(i, j)] = ...
        return memo[(i, j)]

    return dp(len(s1) - 1, len(s2) - 1)
```

dp table:

S1 \ S2	""	a	p	p	l	e
""	0	1	2	3	4	5
r	1	1 ①	2 ②	3	4	5
a	2	1	2	3	4	5
d	3	2	2	3	4	5

PINK indicates base case.



Based on the relationship, we can calculate in this order.

依赖关系.

Eventually find result.

```java
int minDistance(String s1, String s2) {
    int m = s1.length(), n = s2.length();
    int[][] dp = new int[m + 1][n + 1];
    // base case
    for (int i = 1; i <= m; i++)
        dp[i][0] = i;
    for (int j = 1; j <= n; j++)
        dp[0][j] = j;
    // 自底向上求解
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (s1.charAt(i-1) == s2.charAt(j-1))
                dp[i][j] = dp[i - 1][j - 1];
            else
                dp[i][j] = min(
                    dp[i - 1][j] + 1,
                    dp[i][j - 1] + 1,
                    dp[i-1][j-1] + 1
                );
    // 储存着整个 s1 和 s2 的最小编辑距离
    return dp[m][n];
}

int min(int a, int b, int c) {
    return Math.min(a, Math.min(b, c));
}
```

prev was 自顶向下

String

Marching

Robin - Karp Algorithm.

Originally, check of P in S.

S = " a a a a a b"

P = " a a a b"

We need to check P with S_1 ~ S_4,
then S_2 ~ S_5, S_3 ~ S_6 etc
until they matched.

Time complexity: $O(n \times m)$ where n = lens(S), m = lens(p).

But we can check the hashing value.

S = " a a a a a b"

P = " a a a b"

h(aaaa) = h_t ← A number

h(aaab) = h_p ← Another number.

Using sliding window in S to see any $h_t == h_p$.
if hashing number matched,
we compare each character inside hashing window S with P.

Rolling hash:

if ascii value table is following:

a	b	c	d	e	.....
1	2	3	4	5	

S = a b c d e f g h i ...

P = c a e f.

$h_{pattern} = (3 + 4 + 5 + 6) \cdot 18$

$h_{abcd} = (1 + 2 + 3 + 4) = 10$

THEN: ↙ reuse

$h_{bcde} = 10 - a + e$

$= 10 - 1 + 5$

$= 14$

SO constant search time (2) regardless pattern length.

Time Complexity : $O(m+n)$

IF

no collision (e.g. same fare hashing value).

$O(m \times n)$ in worst case if all collided.

solution. ↙

S = a b c d e f g h i ...

P = c a e f.

Add a multiplier: 10^{n-1} where n is the character position from right.

$h_{abcd} = h(1, 2, 3, 4)$.

$= 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

$= 1234$

So continue rolling method for bcde

$h_{bcde} = h_{abcd} - h_a + h_e$

$= \left(\quad - 1 \times 10^3 \right) \times 10 + 5$

↑ shift left.

$= 2345$

However if length is 100, 10^{100} will cause overflow.

↳ Involve appropriate mod operation.

e.g. $h_{abcd} = h(1, 2, 3, 4)$.

$= (1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0) \bmod 113$

↑ A number decided by yourself.

So the number will within range of $[0, 113]$

But $10^{99} \bmod 113 = ?$

Since, $(A \times B) \bmod C = ((A \bmod C) \times (B \bmod C)) \bmod C$.

↳ $10^2 \bmod 113 = ((10 \bmod 113) \times 10 \bmod 113)) \bmod 113$.

↳ for ( ) {

$n = (n \times 10) \% 113$

}

KMP Substring Search.

Originally, check of P in S.

S = " a a a a a b "

P = " a a a b "

We need to check P with S_1 ~ S_4,
then S_2 ~ S_5, S_3 ~ S_6 etc
until they matched.

Time complexity: $O(n \times m)$ where n = lens(S), m = lens(p).

However, in KMP,
we take advantage of
successful character comparson.
(avoiding unnecessary comparison).

S = a d s g w a d s x d s g w a d s g z

P = d s g w a d s g z

S: a d s g w a d s x d s g w a d s g z

P: d s g w a d s g z

not match

In brute force, we need to start at S[2] and compare P again.

But now

we are looking for,

a suffix which is also a prefix,

S: a d s g w a d s x d s g w a d s g z

P: d s g w a d s g z

then no need to compare the known part,

compare the prefix of P with suffix of S.

Prefix - Suffix table.

Compare i j. advance j if no match

i j

	d	s	g	w	a	d	s	g	z
	0	1	2	3	4	5	6	7	8
Table	0	0	0	0	0				

Table: where to start matching in P after a mismatch at i+1.

advance both i and j.

	d	s	g	w	a	d	s	g	z
	0	1	2	3	4	5	6	7	8
Table	0	0	0	0	0	i+1	2		

1

if this is a mismatch.

we don't need to move i back to 0,

we move i to 2,

because S told me!

Combined:

S: a d s g w a d s x d s g w a d s g z

P: d s g w a d s g z

0 1 2 3 4 5 6 7

check last match item in table

	d	s	g	w	a	d	s	g	z
	0	1	2	3	4	5	6	7	8
Table	0	0	0	0	0	1	2	3	0

go to index 2.

S: a d s g w a d s x d s g w a d s g z

P: d s g w a d s g z

we are going from x in S,

g in P.

But table[g] = 0. Back to d.

compare d with x. → not match

↓

advance S.

↓

Time Analysis:

n = lens(S)
m = lens(p).

Building table: $O(m)$

Matching traversal: $O(n)$.

Overall: $O(m+n)$.

Intractability.

	P (within polynomial time)	NP
Easy to solve	✓	X
Easy to verify.	✓	✓
Example.	prime number	拼图 jigsaw.

Cook Theorem: Every NP problem is polynomially reducible to the SAT problem.

SAT ∈ NP-Complete.