```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

print("VISUAL CODE IN Python")

author: Olivia Yan

Contents

Thanks

This guidebook is a collection of eighteen Leetcode problems' visual solutions. I started working on it since October, 2021, and finished it in January, 2022. I wrote the book not only for my Grade 12 Capstone project, but also to accomplish one of my 2021 goals/flags: creating something. I would like to thank Mr. Ubial for being my Capstone project mentor. I would like to thank Richmond Brighouse Library and Vancouver Central Library for providing a nice environment.

2022.01.01
Xi, Yan

MATH

Chapter One

Calculation is the basic function of a programming language. In this chapter, we are going to solve some Math questions using Python.

First, let's review Python common mathematical operations.

operator	function	example
+	addition	>>> 3 + 4 7
-	subtraction	>>> 3 - 4 -1
*	multiplication	>>> 3 * 4 12
/	division	>>> 7 / 2 3.5
%	return the remainder of a division	>>> 7 & 2 1
**	exponentiation	>>> 7 ** 2 49
//	return the largest integer of a division	>>> 7 // 2 3

Here are some common assignment operators and comparison operations. They can make our codes simpler and more readable:

operator	function	example

+=	abbreviation	>>> a = 3
		>>> a += 1
		a: 4

-=	abbreviation	>>> a = 3
		>>> a -= 1
		a: 2

==	equal to	>>> 1 == 3 False
>= or <=	greater than or equal to	>>> 1 >= 0 False
!=	not equal	>>> 10 != 4 True

1.1 Missing Number

Given an array `nums` containing `n` distinct numbers in the range `[0, n]`, return the only number in the range that is missing from the array.

Example 1

```
Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there
are 3 numbers, so all numbers
are in the range [0,3]. 2 is the
missing number in the range since
it does not appear in nums.
```

Example 2

```
Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there
are 3 numbers, so all numbers
are in the range [0,3]. 2 is the
missing number in the range since
it does not appear in nums.
```

In this question, the answer is the difference between the sum of array that has no missing number and the sum of array `nums`, which has a missing number.

To get the sum of array that has no missing number, we use the following math formula:

$$S = \frac{n\,(a + 1)}{2}$$

where,

S = sum of the consecutive integers

n = number of integers

a = first term

For example, to calculate the sum of array `[1,2,3...100]`, we use

S = 100(1+100) /2

= 5050

To calculate the sum of array `[7,8,9...100]`, we use

S = (100-6) (7 + 100) /2

= 5029

In this question, we apply the same formula to calculate the ideal sum, which has no missing number. Remember, when we do this calculation, **the first term should be 1 but not 0:**

ideal_sum = (the length of nums) (first num + last term) /2
= (len(nums)) (1 + len(nums)) / 2

Then we find the difference between the ideal array and sum of nums

result = ideal_sum - sum(nums)

See the following code

```
1    # Missing Number; mathematical solution
2    def missingNumber(self, nums: List[int]) -> int:
3        ideal_sum = len(nums) *(1+(len(nums))) // 2
4        return ideal_sum - sum(nums)
```

1.2 Reverse a Number

Given a signed 32-bit integer `x`, return `x` with its digits reversed. If reversing `x` causes the value to go outside the signed 32-bit integer range `[-231, 231 - 1]`, then return 0.
Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

Example 1

```
Input: x = -123
Output: -321
```

Example 2

```
Input: x = 120
Output: 21
```

Without transforming `x` into a string or list, we can still use mathematical operators to collect the ones digit of `x`. And in the same way, we can obtain the tens digit, hundreds digit, or even thousands digit of `x`, like the following procedure tables.

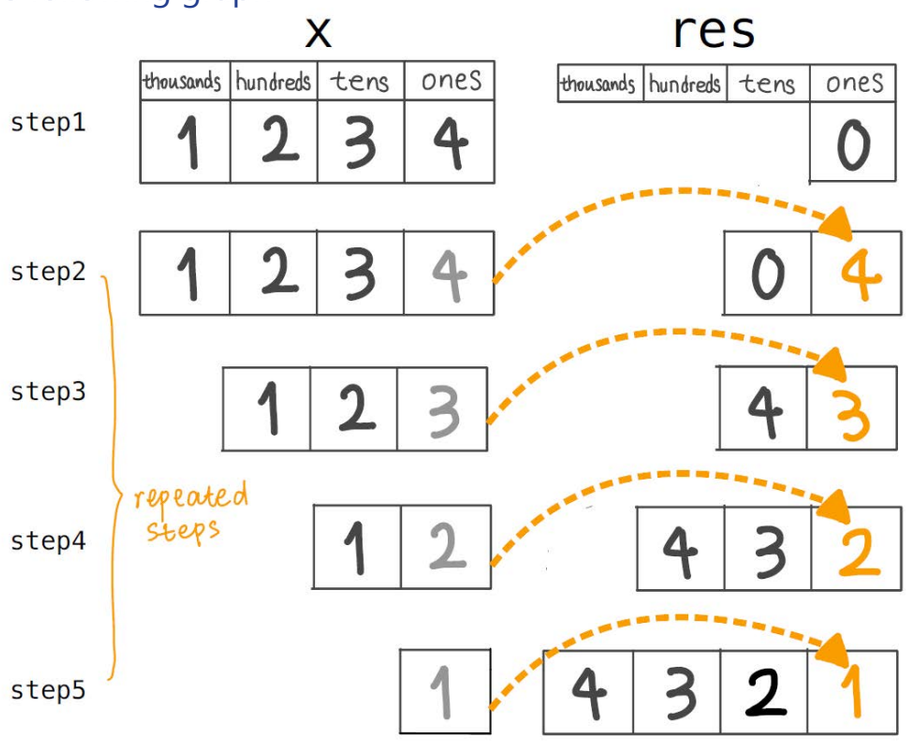
* For the table, we will use rule 1 and rule 3 to solve this question mathematically.

	code	explanation
1	>>> 123 % 10 3	123 ÷ 10 = 12 …3 `%` returns is the remainder of the division `123 ÷ 10`, which is the ones digit of 123
2	>>> 123 % 100 23	123 ÷ 100 = 1 … 23 `%` returns is the tens and ones digit of `123`
3	>>> 123 // 10 12	123 ÷ 10 = 12.3 `//` returns the largest integer of the division, which is the digits after `123`'s ones digit
4	>>> 123 // 100 1	123 ÷ 100 = 1.23 `//` here returns the digits after `123`'s tens digit

For this question, we repeatedly "pop"the last digit (ones digit) of `x` and then "push"the digit to variable `res`. In the end, the variable `res` will be the reversed `x`. We check whether `x` is negative first. If yes, we reverse the absolute value of it, and, in the end, put a negative sign onto variable `res`.

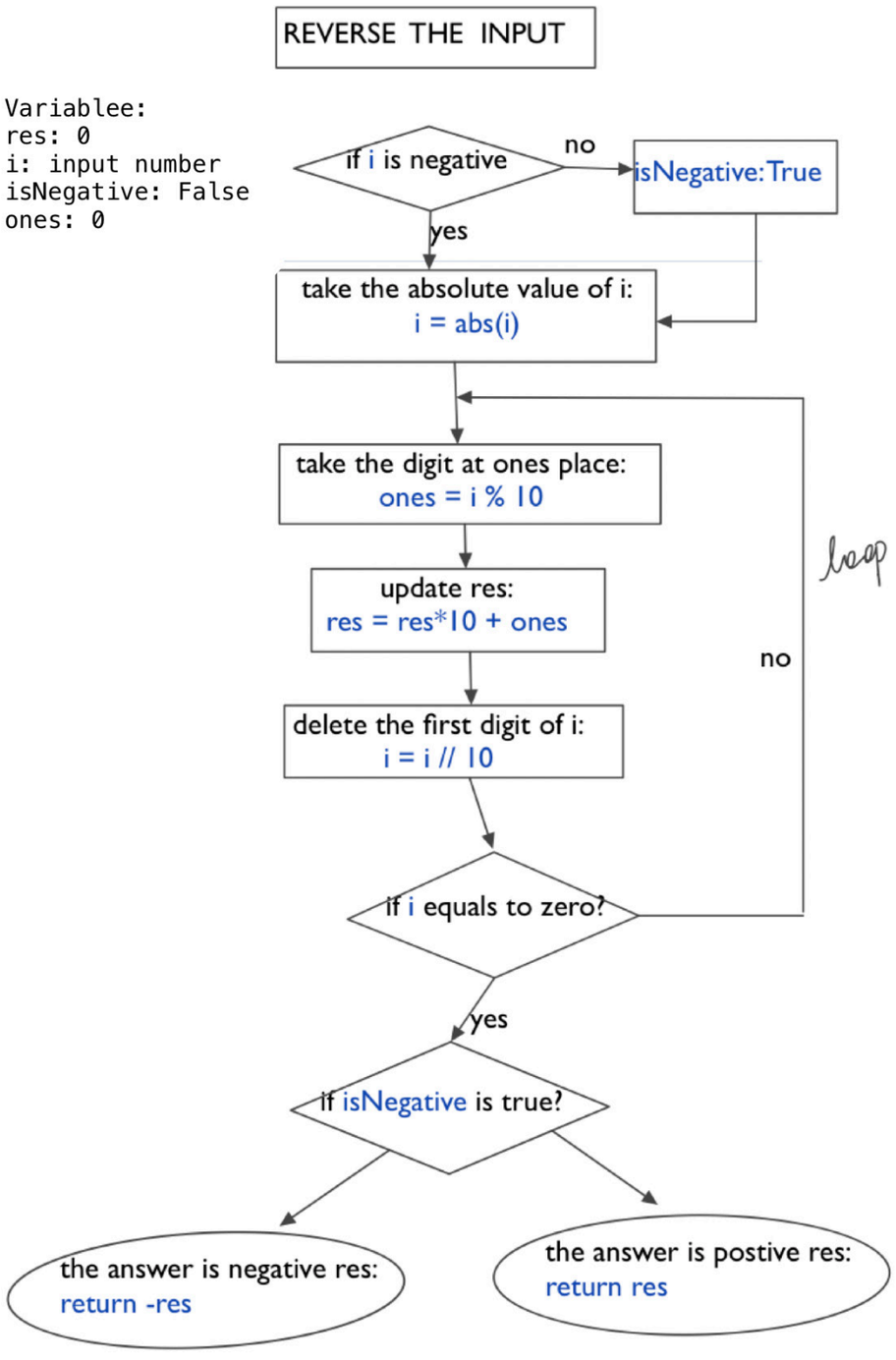
- if reversed `x` go outside the signed 32-bit integer range `[-231, 231 - 1]`, we return 0

See the following graph



See the following flowchart

The flowchart illustrates both the logic and the loop in the solution. In the graph, words in black are explanations, and words in blue are the code in Python.



See the code

```python
# Reverse A Number; mathematical solution
# by olivia y
def reverse(self, x: int) -> int:
    res = 0
    isNegative = False
    if x < 0:
        isNegative = True
        x = abs(x)
    while x > 0:
        temp = x % 10
        x = x //10
        res = res*10 + temp
    if res > (2**31-1) or res < (-2**31):
        return 0
    if isNegative:
        return -res
    return res
```

1.3 Palindrome Number

Given an integer x, return true if x is palindrome integer.
An integer is a palindrome when it reads the same backward as forward.

Example 1

```
Input: x = 121
Output: true
Explanation: 121 reads as 121
from left to right and from right
to left.
```

Example 2

```
Input: x = 10
Output: false
Explanation: Reads 01 from right
to left. Therefore it is not a
palindrome.
```

In this question, we can either reverse the whole `x` or just reverse the half of it. Although the latter sounds more efficient, as a matter of fact, the former is more efficient than the latter.

* if `x` is negative, it is definitely not palindrome, so return `False`

Approach One: Reverse the Half of input

Similar to the last question, we repeatedly"pop"the last digit of `x` and"push"the digit into a variable `res`. When we have reversed half of `x`, we know whether the half-reversed `x`-- `res`-- equals the rest of non-reversed `x`.

But how can we know when we stop reversing `x`?

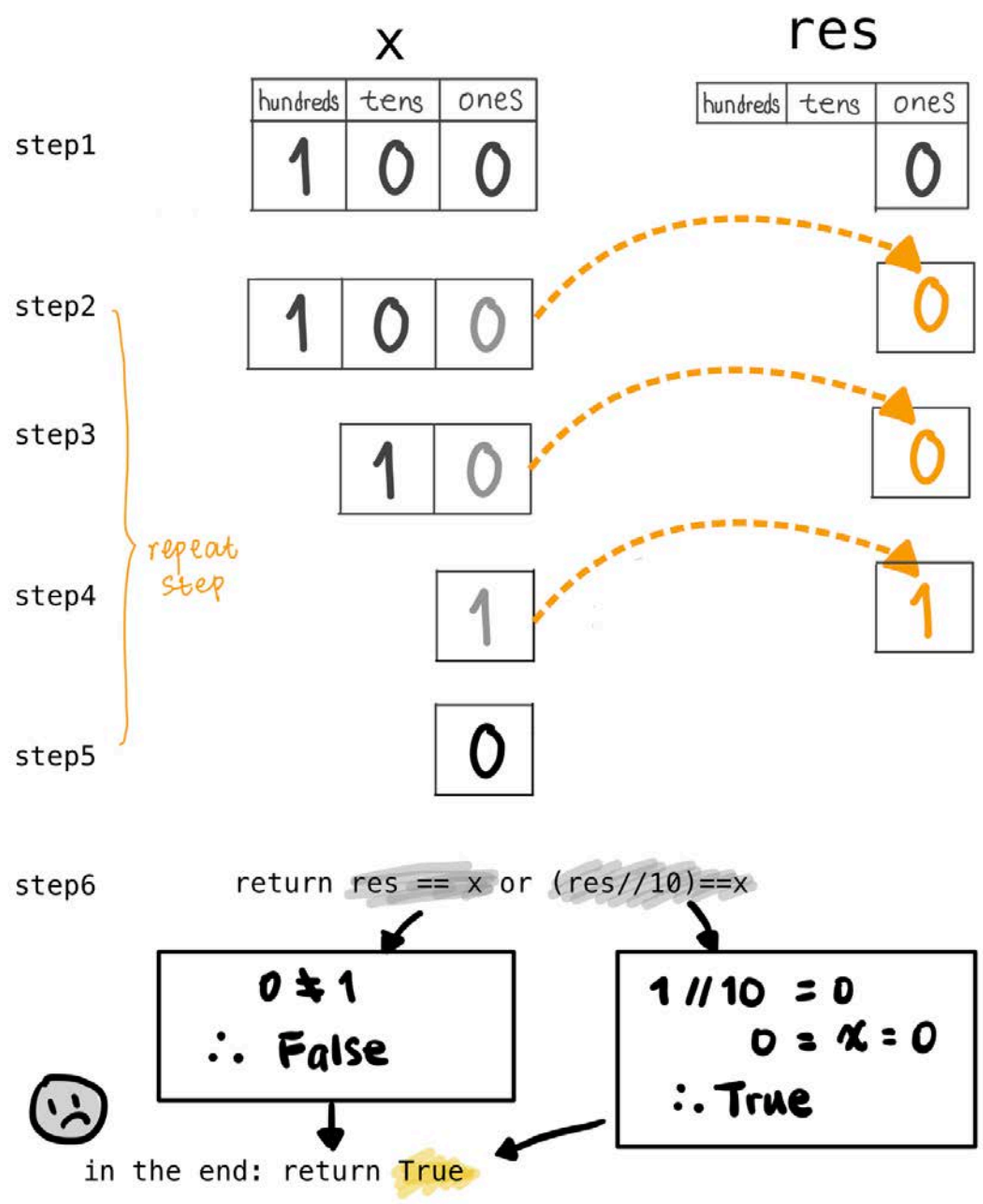
Since every time we pop and push, the value of `res` increases while the value of `x` decreases, we can tell when to stop by their values.

See the code:

```python
# Palindrome; Reverse half of x
# by olivia y
def isPalindrome(self, x: int) -> bool:
    if x < 0:
        return False

    res = 0
    while res < x:  # when we haven't reversed half of x yet
        res = res*10 + x % 10
        x = x // 10
    print(res, x)
    return res == x or (res//10) == x
```

However, there is one special case: what if the input is 100?

The following graph illustrates what will happen if the input `x` is 100, based on the above code.



Obviously, 100 is not a palindrome number while the final result is `True`. When the input is in `[110, 10, 220...]`, the answer the code returns is not what we want. To solve this issue, we need to add one more if statement

```
if x % 10 == 0 and x != 0: return False
```

See the revised code:

```python
# Palindrome; reverse the half
# by yx
def isPalindrome(self, x: int) -> bool:
    if x < 0 or (x % 10 == 0 and x != 0):
        return False
    res = 0
    while res < x:
        res = res*10 + x % 10
        x = x // 10
    print(res, x)
    return res == x or (res//10) == x
```

Approach Two: reverse the whole input

To reverse the whole input, we use the same approach as the question reverse the number.

```python
# Palindrome; reverse the whole x
# by olivia y
def isPalindrome(self, x: int) -> bool:
    if x <0:
        return False
    res = 0
    original_x = x
    while x > 0 :
        res = res*10 + x%10
        x = x//10
    return res == original_x

```

* Since we compare `res` with original `x` in the end, and `x` changes every time we pop its last digit, we need to store the `original x` first.

Approach 3: use string and slice

Reverse `x` and return the boolean value of `reversed_x == x`. In Python, we use `[::-1]` to reverse a string, see the following example.

```
>>> a = “Hello World”
>>> a[::-1]
‘dlroW olleH’
```

*slice creates a new string. It does not change the string itself.

See the code:

```python
# Palindrome; string & slice
# by olivia y
def isPalindrome(self, x: int) -> bool:
    return str(x) == str(x)[::-1]
```

1.4 Ugly Number

An ugly number is a positive integer whose prime factors are limited to 2, 3, and 5.
Given an integer `n`, return true if `n` is an ugly number.

Example 1

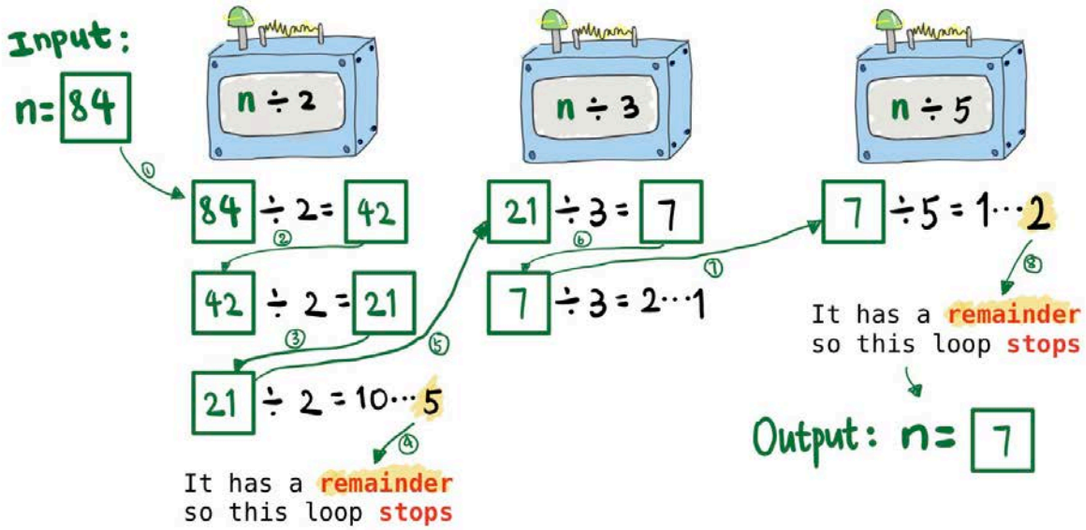
```
Input: n = 6
Output: true
Explanation: 6 = 2 × 3
```

Example 2

```
Input: n = 14
Output: false
Explanation: 14 is not ugly since
 it includes the prime factor 7.
```

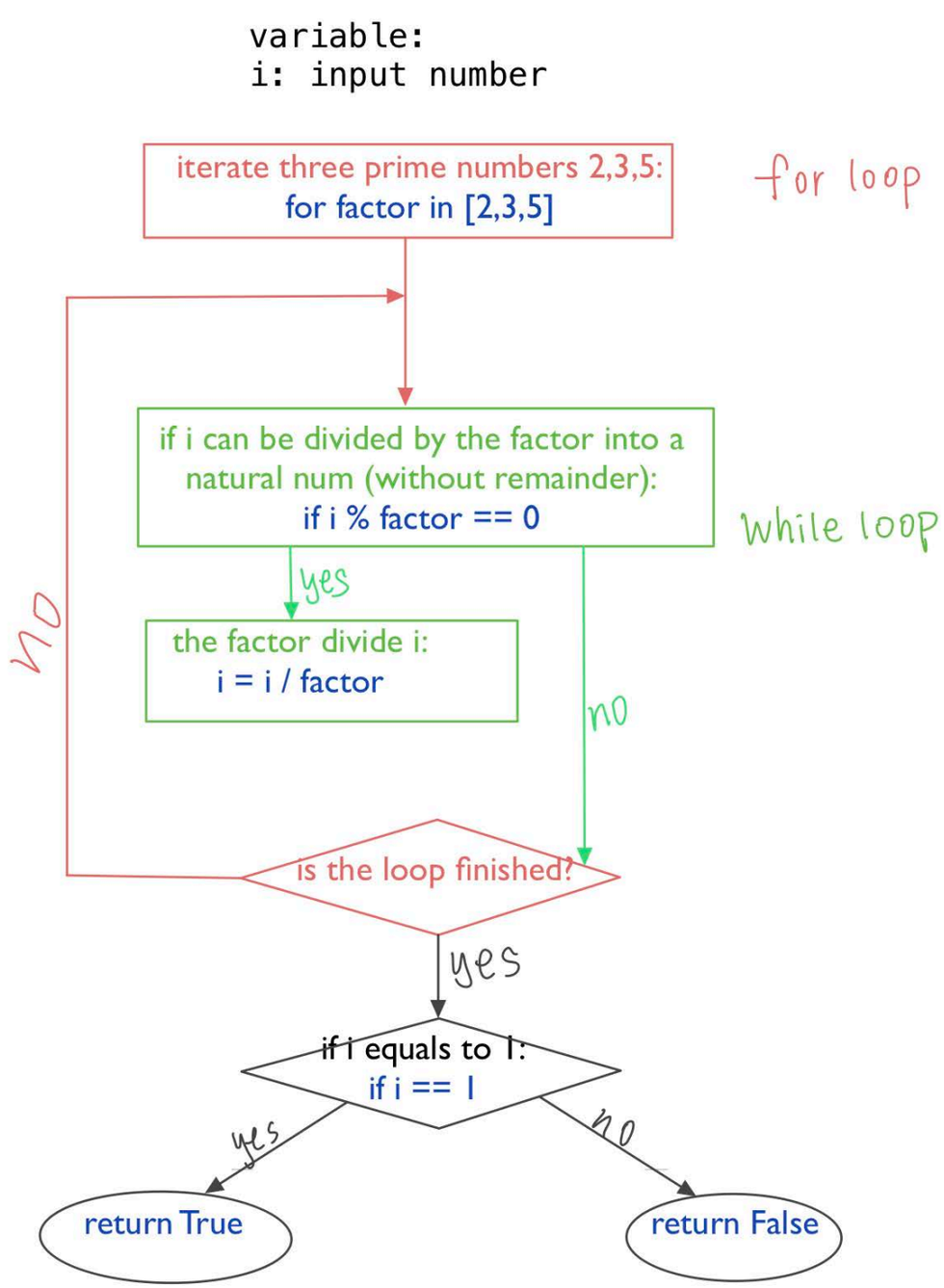
If n can be divided by 2 with no remainder, we repeatedly divide n by 2 until there is a remainder. Then, if n can be divided by 3 with no remainder, we repeatedly divide n by 3 until there is a remainder. Similarly, if n can be divided by 5 with no remainder, we repeatedly divide n by 5 until there is a remainder. In the end, if n equals to 1, it is an ugly number, vice versa.

See the visualized code:



The output is 7. Since 7 does not equal 1, 86 is not an ugly number.

See the flowchart:



See the code:

```
1    # Ugly Number
2    # by olivia yan
3    def isUgly(self, n: int) -> bool:
4        if n == 0:
5            return False
6        for divider in [2,3,5]:
7            while n % divider == 0:
8                n = n // divider
9        return n == 1
```

*If the input is 0, the while loop from line 7 to 8 will run forever. Since 0 is a special case in this question, we need to add an if statement before it runs into the while loop. So from line 4 to 5, we add an if statement

1.5 Happy Number

Write an algorithm to determine if a number `n` is happy.
A happy number is a number defined by the following process:

- Starting with any positive integer, replace the number by the sum of the squares of its digits.
- Repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1.
- Those numbers for which this process ends in 1 are happy.

Return true if `n` is a happy number, and false if not.

Example 1

Input: n = 19
Output: true
Explanation:
$1^2 + 9^2 = 82$
$8^2 + 2^2 = 68$
$6^2 + 8^2 = 100$
$1^2 + 0^2 + 0^2 = 1$

Example 2

Input: n = 2
Output: false
Explanation:
$2^2 = \mathbf{4}$
$4^2 = 16$
$1^2 + 6^2 = 37$
$3^2 + 7^2 = 58$
$5^2 + 8^2 = 89$
$8^2 + 9^2 = 145$
$1^2 + 4^2 + 5^2 = 42$
$4^2 + 2^2 = 20$
$2^2 + 0^2 = \mathbf{4}$
...

Repeatedly replace `n` by the sum of the squares of its digits until `n` equals 1. To avoid that `n` is in a cycle, we create a `visited` list that stores every visited number. If we encounter a visited number, we are in a cycle, which means there is no other way to get 1 and `n` is not a happy number.

There are two approaches to get the sum of the squares of `n`'s digits

Approach One: Mathematically

we pop the last digit of `n` and square the digit until `n` is 0

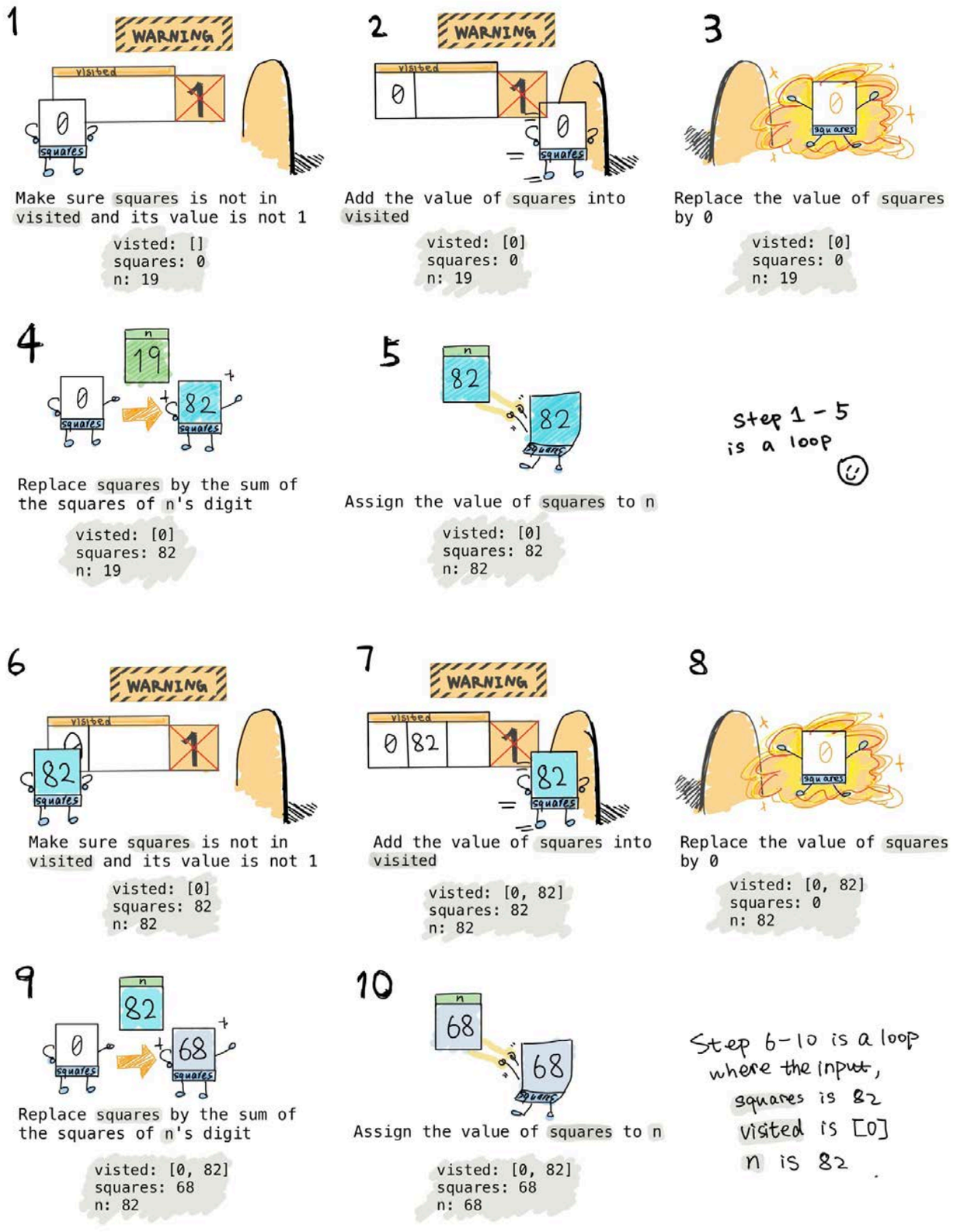
```
>>> n, squares = 123, 0
>>> while n > 0:
    squares += (n % 10) ** 2
    n = n // 10
>>> print (n)
0
>>> print(squares)
14
```

Approach Two: String

We convert `n` into a string. Get each digit by its index.

```
>>> n, squares = 123, 0
>>> n = str(123)
>>> for digit in range (len(n)):
    squares += (int(n[digit]) **
2)
>>> print(n)
‘123’
>>> print(squares)
14
```

The graph on the next page is a visualized procedure of the solution to the question. In the graph, step 4 and step 9 is to get the sum of the squares of `n`’s digits, which we can use the two above ways to achieve.



The loop continues until the variable `squares` is in `visited` or equals 1.

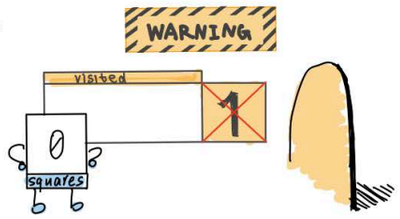
See the code:

```
3       visited = []
4       sqaures = 0
5
6       while sqaures != 1 and sqaures not in visited:
7           visited.append(sqaures)
8           n = str(n)
9           sqaures = 0
10          for digit in range (len(n)):
11              sqaures += int(n[digit]) ** 2
12          n = sqaures
13
14      return n == 1
15
```

For creating a `visited` list, it is important to know where to put the `visited` list. If it is put in a wrong position, the code will run in a wrong way. Also, when items are put in `visited` are important too. Let's have a look at a wrong example:

For the above graph, if we assign 0 to `squares` after step five, the next loop will stop. The following graph will happen then.

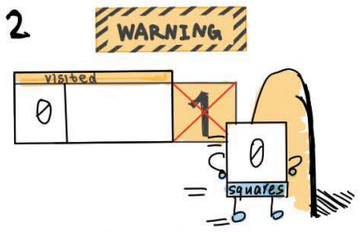
1



Make sure `squares` is not in `visited` and its value is not 1

```
visted: []
squares: 0
n: 19
```

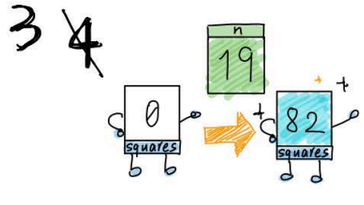
2



Add the value of `squares` into `visited`

```
visted: [0]
squares: 0
n: 19
```

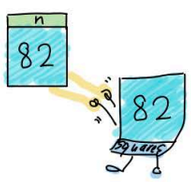
3 ~~4~~



Replace `squares` by the sum of the squares of `n`'s digit

```
visted: [0]
squares: 82
n: 19
```

~~4~~ 4



Assign the value of `squares` to `n`

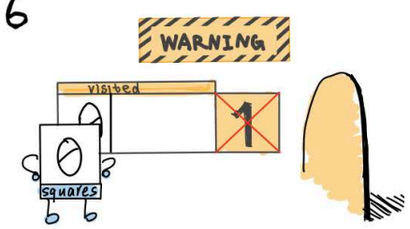
```
visted: [0]
squares: 82
n: 82
```

~~5~~ 5

Replace the value of `squares` by `0`

```
visted: [0]
squares: 0
n: 82
```

6



Make sure `squares` is not in `visited` and its value is not 1

```
visted: [0]
squares: 0
n: 82
```

At step 6, `squares` is 0. 0 is in `visited` so the loop will stop and the final result will be wrong.

BINARY SEARCH

Chapter Two

One day, you borrowed 1000 books from the school library. When you left the library, the library alarm went off, which meant you registered one book wrong. So you put each book under the alarm, checking which book had a problem. At this time, a man walked to you with a disdainful smile. He divided these 1000 books into two piles. When he put the first pile under the alarm, it went off. Then, he divided the first pile into two smaller piles … Finally, after testing log1000 times, the man found the book that you forgot to register.

In the story, if you check the 1000 books one by one, it takes a lot of time. However, the man incessantly divided the books into a half and tested it. This method is called binary search, which is more efficient.

Binary search is a search method and a divide-and-conquer algorithm that can be used in a sorted list. To implement binary search, we take a sorted list and start looking for an element from the middle of the list. Each time, we divide the list into a half and check which half the element was in. There are two rules of binary search:

1. Shrink the search space every iteration
2. Cannot exclude any potential answers during each shrinking

2.1 Find an Exact Number

Given a sorted array of `n` integers and a target value, determine if the target exists in the array using the binary search algorithm. If the target exists in the array, print the index of it. If not, return None.

Example 1

```
Input: n = [1,4,6,10], target = 6
Output: 2
Explanation: n[2] == 6 so returns
2
```

Example 2

```
Input: n = [1,10,20,41], target =
50
Output: None
Explanation: in list n, none of
its elements is 50
```

We can go through each element one by one in the given list, but if the list has millions of numbers, it will take more than one hour to find out the result.

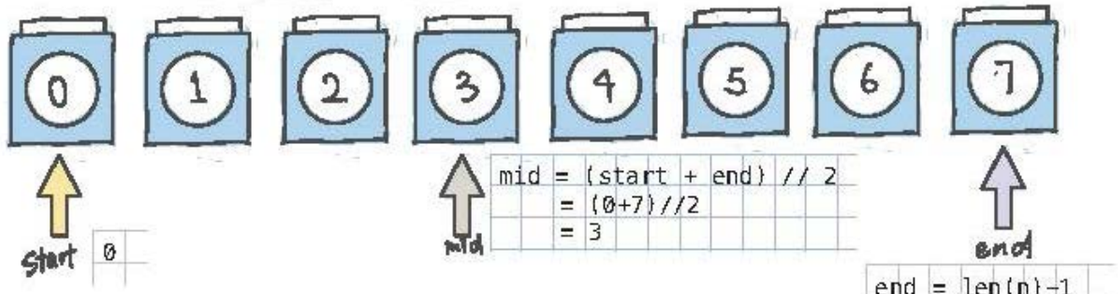
We should create three nodes, which respectively represent `n`'s starting point, its middle point, and its ending point. When the target is greater than the middle node, the target may be in the section between the middle node and the ending node, so we continuously search the target in that section, and vice versa. When the middle node points at the target, return `True`. However, if the starting node and ending node are adjacent to each other and the target is still not found, the target is not in the list.

* when we use this approach, the original list `n` will not be altered.

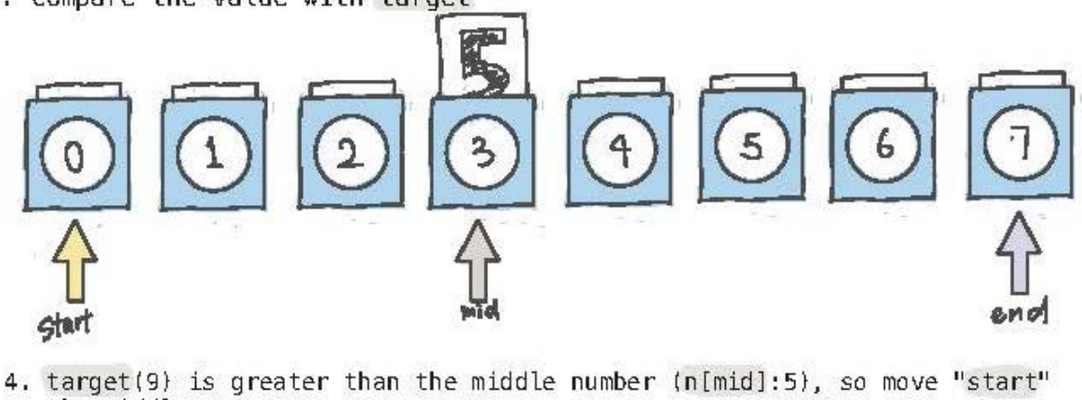
The graph on the next page illustrates how the code will work. The number on each blue rectangle is its index in n not its value.

input: n = [1,3,4,5,7,8,9,10], target = 9

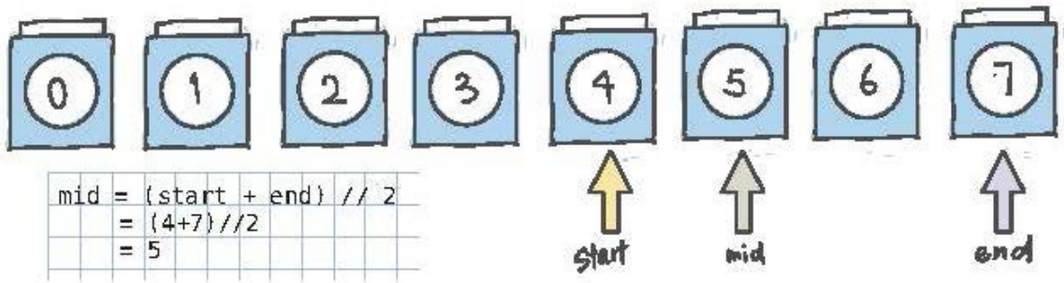
1. initialize start, mid, and end



2. get the value that mid points at
3. compare the value with target



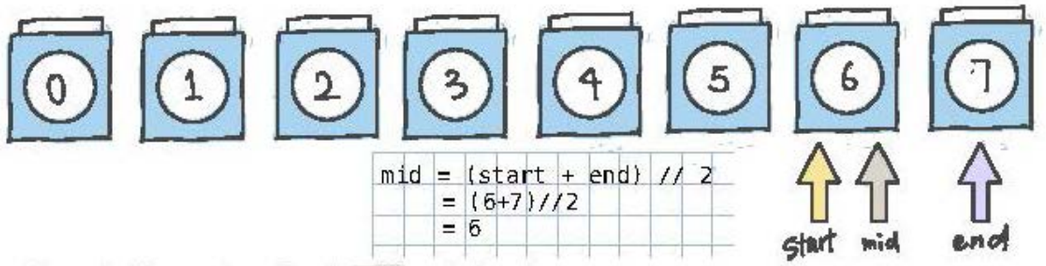
4. target(9) is greater than the middle number (n[mid]:5), so move "start" to the middle:
*For the next run, there is no need to include current "mid" (3)



5. get the value that mid points at
6. compare the value with target

7. target(9) is greater than the middle number (n[mid]:8), so move "start" to the middle

* Still, for the next run, there is no need to include cuurent "mid" (5)



8. get the value that mid points at
9. compare the value with target

10. target(9) equals to the middle number (n[mid]:9), so you find where target is. Its index in n is 6

See the code:

```
1   # binary search - find the exact value
2   # oy
3   n = [1,3,4,5,7,8,9,10]
4   target = 9
5   def binary(start, end):
6       mid = (start + end) // 2
7       if n[mid] == target:
8           return True
9       if start == mid:
10          return False
11
12      if n[mid] > target:
13          return binary(start, mid)
14      elif n[mid] < target:
15          return binary(mid+1, end)
16  print(binary(0, len(n)-1))
```

On line 15, should the new start be mid or mid+1? Or either is fine?

2.2 Find an Uncertain Number

Given a sorted array of `n` integers and a target value, determine the index of the smallest possible number that is greater than the target in the array using the binary search algorithm. If the smallest possible number exists, print the index of it. If not, return `None`.

Example 1

input: n = [2,4,6,8], target = 5
output: 2
explanation: n[2] is 6. In the array, it is the smallest number that is greater than 5

Example 2

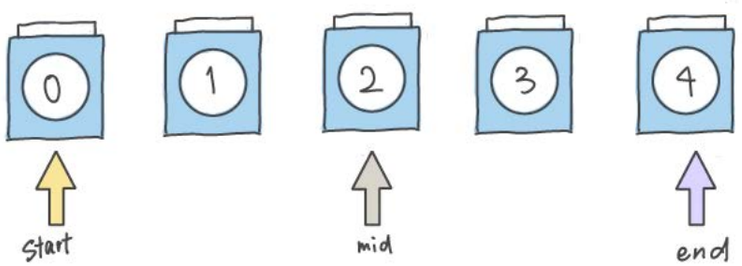
input: n = [11,12,13], target = 10
output: 0
explanation: n[0] is 11. In the array, it is the smallest number that is greater than 10

in this problem, we need to find an uncertain number. It is not a specific number like the last problem. For the last problem, if the target number is `10`, we find the number that exactly has the value of `10` in the given list. However, for this problem, if the target number is `10` in this problem, the answer could be any number that is greater than `10`. The answer completely depends on what numbers the given list has. Thus, the base case cannot be `if n[mid] == target`. Instead, it is `if start == end`. When the two nodes, `start` and `end`, point towards the same spot, it is the answer.

Following graphs illustrate the solution :

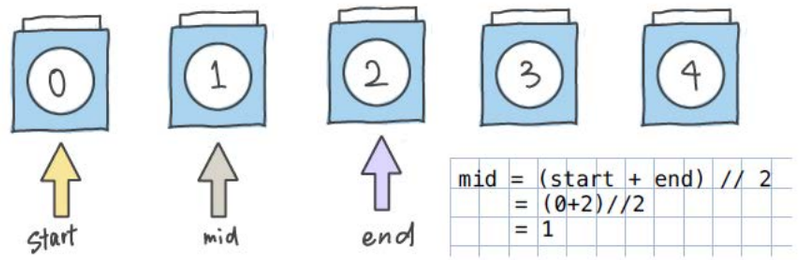
```
input: n = [0,3,5,7,11], target = 4
FIND THE SMALLEST INTEGER IN THE GIVEN LIST THAT IS
GREATER THAN target
```

1. initialize `start`, `mid`, and `end`



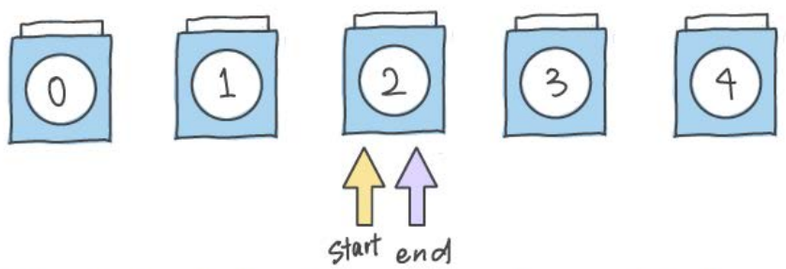
2. get the value that `mid` points at
3. compare the value with `target`

4. `target(4)` is smaller than the middle number (`n[mid]:5`), so move `end` to the middle
   * when the middle number is greater than `target`, you must include current "mid" (2) because it is possible that the middle number is the answer

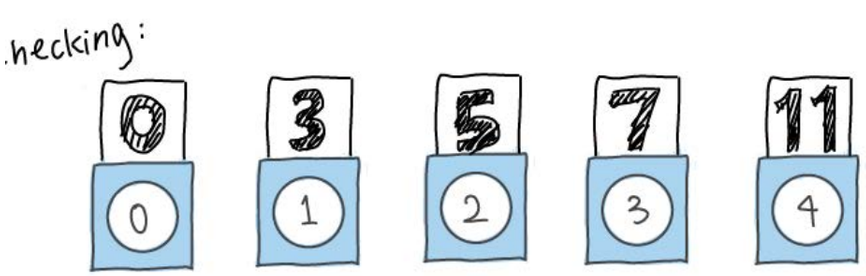


5. get the value that `mid` points at
6. compare the value with `target`

7. `target(4)` is greater than the middle number (`n[mid]:3`), so move `start` to the middle
   * when the middle number is smaller than `target`, you don't need to include the middle number for the next run because the middle number is impossible to be the answer



8. Now `start` and `end` point at the same box, so 2 is the answer, the index of the smallest integer that's greater than 4.



`n[2]` = 5 and 5 is the smallest number that is greater than 4 in this list

See the code:

```
1   # binary search - find a vague value
2   # oy
3   n = [2,3,5,7,11]
4   target = 8
5
6   if target > n[-1]:
7       print(None)
8
9   def binary(start, end):
10      mid = (start + end) // 2
11      if start == end:
12          return start
13      if target >= n[mid]:
14          # no need to keep mid
15          return binary(mid+1, end)
16      elif target < n[mid]:
17          # mid is a possible answer
18          return binary(start, mid)
19  print(binary(0, len(n)-1))
```

From line 6 to 7, we add an if statement for one special case: if the target number is greater than the greatest number in the given array, return `None`.

When writing the code, we always keep the rules of binary search in mind : "Cannot exclude any potential answers during each shrinking". For this problem, if `n[mid]` is greater than the target number, we need to include `n[mid]` for the next run, because `n[mid]` can be a potential answer. Similarly, if `n[mid]` is smaller than the target number, we do not include `n[mid]` for the next run, because, for this problem, a number that is smaller than the target number can never be a potential answer.

2.3 Panacea: Leave Two Potential Answers

Based on different requirements of different answers, we need to slice the input differently. Like the last two problems, even though they both are binary search problems, we slice the input n differently. Is there a general formula that we can use for every binary search problem? The answer is YES.

We slice an input `n` each time until two last potential answers are left. Then we select the answer based on the requirement of a problem.

> Given a sorted array of `n` integers and a target value, determine the index of the number in the array that is closest to the target using the binary search algorithm.

Example 1

Input: `n = [2,4,7,8], target = 5`
Output: `1`
Explanation: `n[1]` has the value of 4. In n, two numbers are adjacent to the target number; one is 4, and the other is 7. The difference between 4 and 5 is 1, while it between 7 and 5 is 2, so the index of 4 is the answer.

Example 2

Input: `n = [11,12,13], target = 10`
Output: `0`
Explanation: `n[0]` has the value of 11. In n, two numbers are adjacent to the target number; one is 11, and the other is 12. The difference between 11 and 10 is 1, while it between 12 and 10 is 2, so the index of 11 is the answer.

In the given array, which two elements are potential answers? They are:

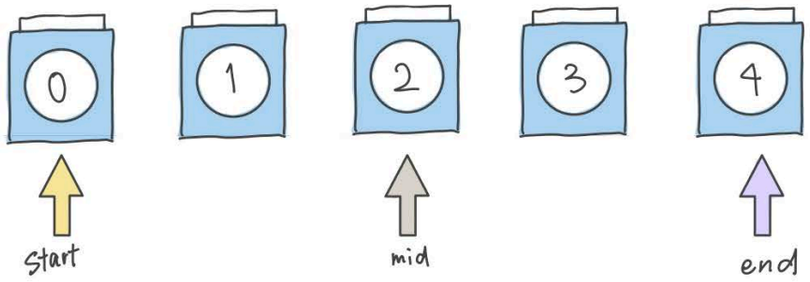
1. the greatest number that is smaller than `target` in the given array
2. the smallest number that is greater than `target` in the given array

Thus, we continuously slice the array and shorten the searching area, until two potential anwers are left.

Illustrated graph:

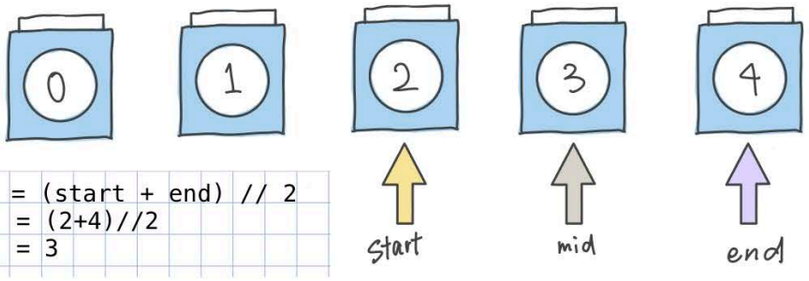
```
input: n = [0,3,4,7,11], target = 6
```

1. initialize `start`, `mid`, and `end`



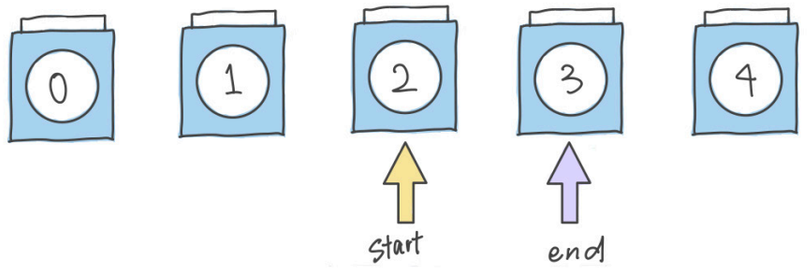
2. get the value that `mid` points at
3. compare the value with `target`

4. `target`(6) is greater than the middle number (`n[mid]`:4). We move `end` to the middle, because we are looking the smallest number that is greater than `target` then.
    * when the middle number is smaller than `target`, we must include current "`mid`" (2) for the next run, because `n[mid]` here is probably the greatest number that is smaller than `target` in the array.

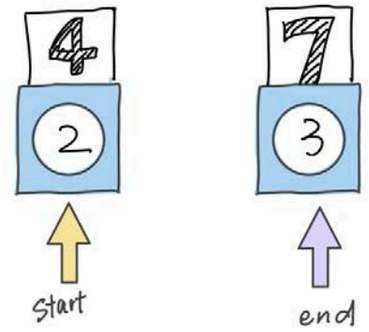


5. get the value that `mid` points at
6. compare the value with `target`

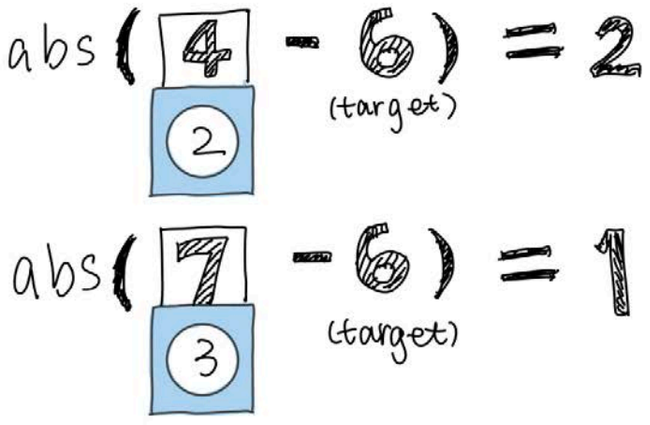
7. `target`(6) is smaller than the middle number (`n[mid]`:7). We move `end` to the middle now.
    * when the middle number is greater than `target`, we must include current "`mid`" (3) for the next run, because `n[mid]` here is probably the smallest number that is greater than `target` in the array.



8. Now `start` and `end` point towards two adjacent boxes, so we return `start` and `end`,the indice of two potential answers.



9. compare the difference between each potential answer and `target`. Return the potential answer that is closer to `target`



See the code:


```python
# binary search - find the closest int(panacea)
# oy
n = [2,3,5,8,11]
target = 6

def binary(start, end):
    mid = (start + end) // 2
    if start == end -1:
        return (start, end)
    if n[mid] > target:
        # n[mid] can be a potential answer
        return binary(start, mid)
    elif n[mid] < target:
        # n[mid] can be a potential answer
        return binary(mid, end)

first, second  = binary(0, len(n)-1)
if abs(n[first]-target) > abs(n[second] -target):
    print("The index of the closest integer is", second)
elif abs(n[first]-target) < abs(n[second] -target):
    print("The index of the closest integer is", first)
else:
    print("There are two closest integers:", first, second)

```

DYNAMIC PROGRAMMING

Chapter Three

Dynamic Programming is an algorithm technique for solving a problem by breaking it down to simpler subproblems and calculate their optimal solutions. This technique can only be used onto problems whose optimal solution depends upon the optimal solution to each subproblem.

Also, for dynamic programming, it is important that we save the answer of each subproblem. This step is called memoization. After solving one subproblem, we store its answer / data, so that, when another recursion requires the answer to this subproblem, we are able to directly use the stored value, instead of calculating the subproblem one more time. Memoization saved abundant time.

3.1 Fibonacci Sequence

Given an integer `n`, return `n`th number in Fibonaaci sequence.
Fibonacci sequence is a set of numbers that starts with a zero, followed by a one, and proceeds based on the rule that each number (called a Fibonacci number) is equal to the sum of the preceding two numbers.

Example 1

Input: n = 5
Output: 5
Explanation: Fibonacci Sequence is [0,1,1,2,3,5] and index 5 (the index starts with 0) of the sequence is 5

Example 2

Input: n = 10
Output: 34
Explanation: Fibonacci Sequence is [0,1,1,2,3,5,8,13,21,34] and index 10 of the sequence is 3

Approach One: recursion

we define a function with an argument `x`. The function will return the `x`th fibonacci number.

See the code:

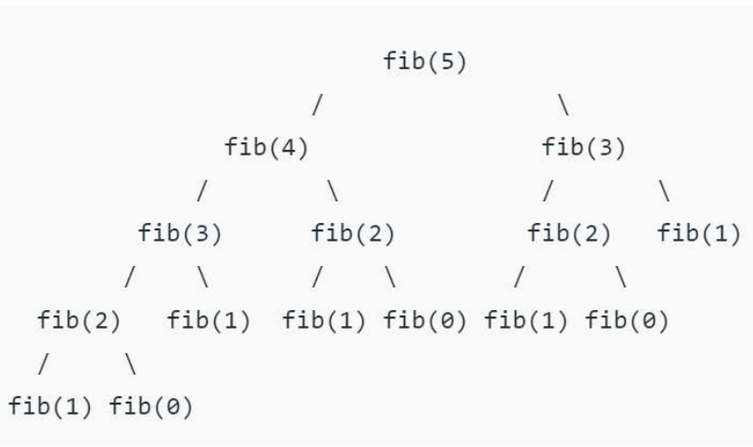
```
1   # fibonacci number; recursion
2   # yx
3   n = 5
4   def fib(n):
5       if n == 0:
6           return 0
7       if n == 1:
8           return 1
9       return fib(n-1) + fib(n-2)
10  print(fib(n))
```

However, this method is very inefficient and has many repetitive steps. See the two graphs below.

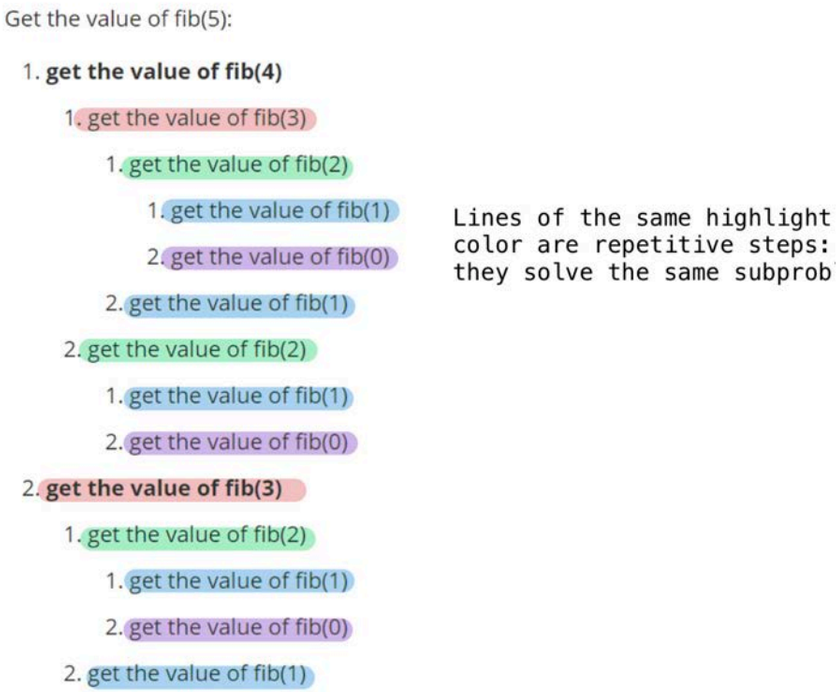
To get `fib(5)`, the code need to find the value of `fib(4)` and `fib(3)`. Then, to find `fib(4)`, the code needs to find the value of `fib(3)` and `fib(2)`. To find `fib(3)`, ...

See the graph:

Original tree for recursion



See the flowchart:



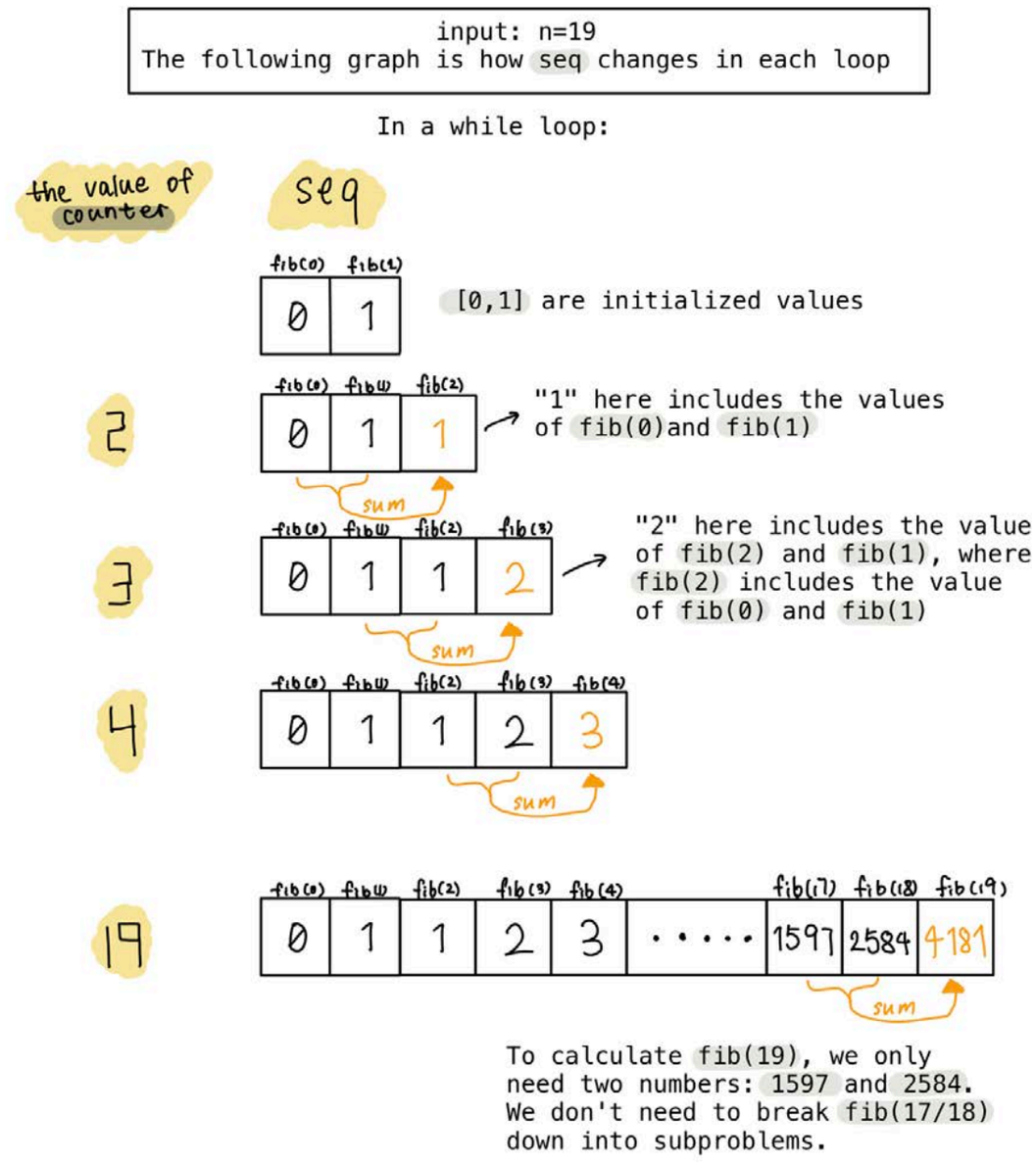
Lines of the same highlight color are repetitive steps: they solve the same subprob

The time complexity increases exponentially. When the input `n` is greater than `20`, the code runs very very very slow.

Approach Two: Dynamic Programming

all the fibonacci numbers except the preceding two do not matter. We only care the values of preceding two fibonacci numbers, and we don't care how these numbers are approached. So we create a list, and store the answers to each subproblem into the list. When any recurision requires the answer to a subproblem, we are able to directly use the stored value, instead of calculating the subproblem one more time.

See the graph:



See the code:

```python
# Fibonacci Number: Store values
# Dynamic Programming

# create a list to store all the fib numbers
seq = [0,1]

def fib(target):
    # Since the list seq contains two digits, counter starts at 2
    counter = 2
    while counter < target:
        num = seq[-1] + seq[-2]
        seq.append(num)
        counter += 1
    return seq[-1]

print(fib(10)) # ->  34
print(seq)  # -> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

The time complexity here is linear. Using dynamic programming, we can even find the 200,000th fibonacci number in one second.

Approach Two (optimized): Dynamic Programming

We can make a small change on the code above, optimizing it a little bit. Since we only care the preceding two fibonacci numbers, we only need to store three numbers:

1. `(n-2)`th fibonacci numbers
2. `(n-1)`th fibonacci numbers
3. `n`th fibonacci numbers

As a matter of fact, the optimized code is not faster than the original dynamic-programming code. What it really improved is the memory usage.

See the code:

```python
# fibonacci number; dp optimized
# yx
n = 200000

def fib(n):
    fib_num1 = 0
    fib_num2 = 1
    fib_num3 = 0 # the initialized value of fib_num3 can be anything
    counter = 1
    while counter < n:
        fib_num3 = fib_num1 + fib_num2
        fib_num1 = fib_num2
        fib_num2 = fib_num3
        counter += 1
    return fib_num3
print(fib(n))
```

3.2 Climbing Stairs

You are climbing a staircase. It takes `n` steps to reach the top.
Each time you can either climb `1` or `2` steps. In how many **distinct** ways can you climb to the top?

Example 1

```
Input: n = 2
Output: 2
Explanation: There are two ways
to climb to the top.
    1. 1 step + 1 step
    2. 2 steps
```

Example 2

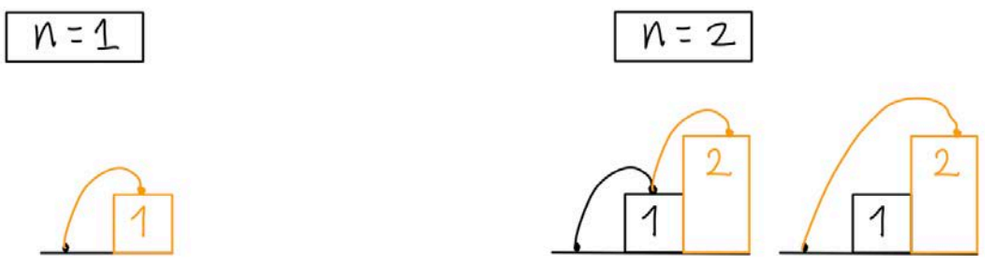
```
Input: n = 3
Output: 3
Explanation: There are three ways
to climb to the top.
    1. 1 step + 1 step + 1 step
    2. 1 step + 2 steps
    3. 2 steps + 1 step
```

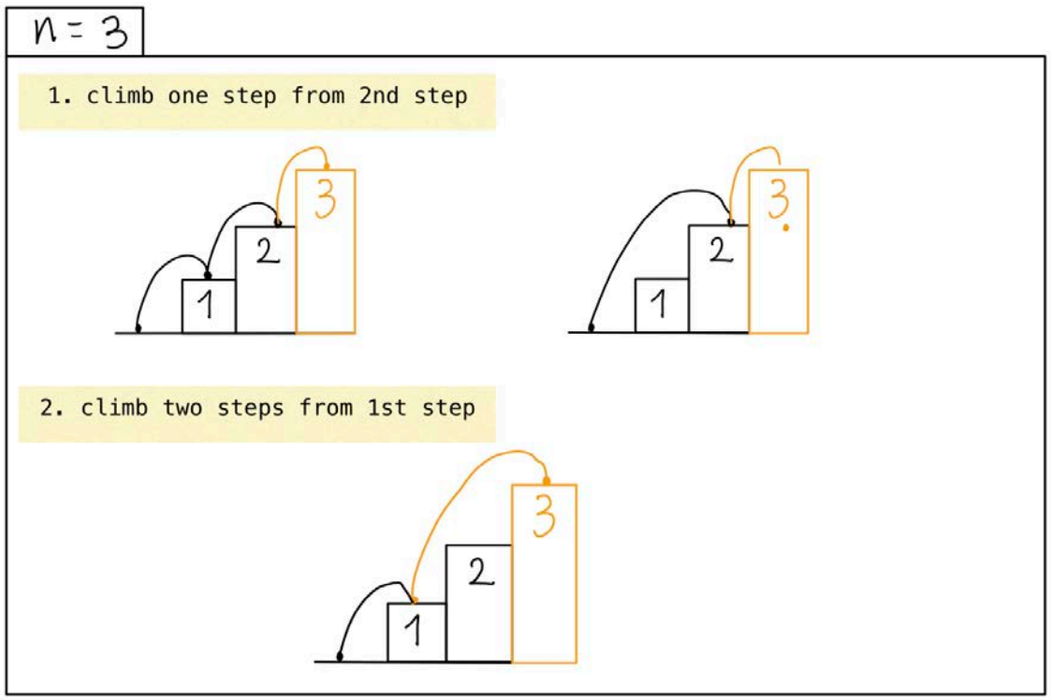
Climbing stairs is the same as the fibonacci sequence. The total distinct ways to nth step is the sum of the distinct ways to `(n-1)`th step and `(n-2)`th step. Because when you want to climb to `n`th step from other step, you only have two possible steps:

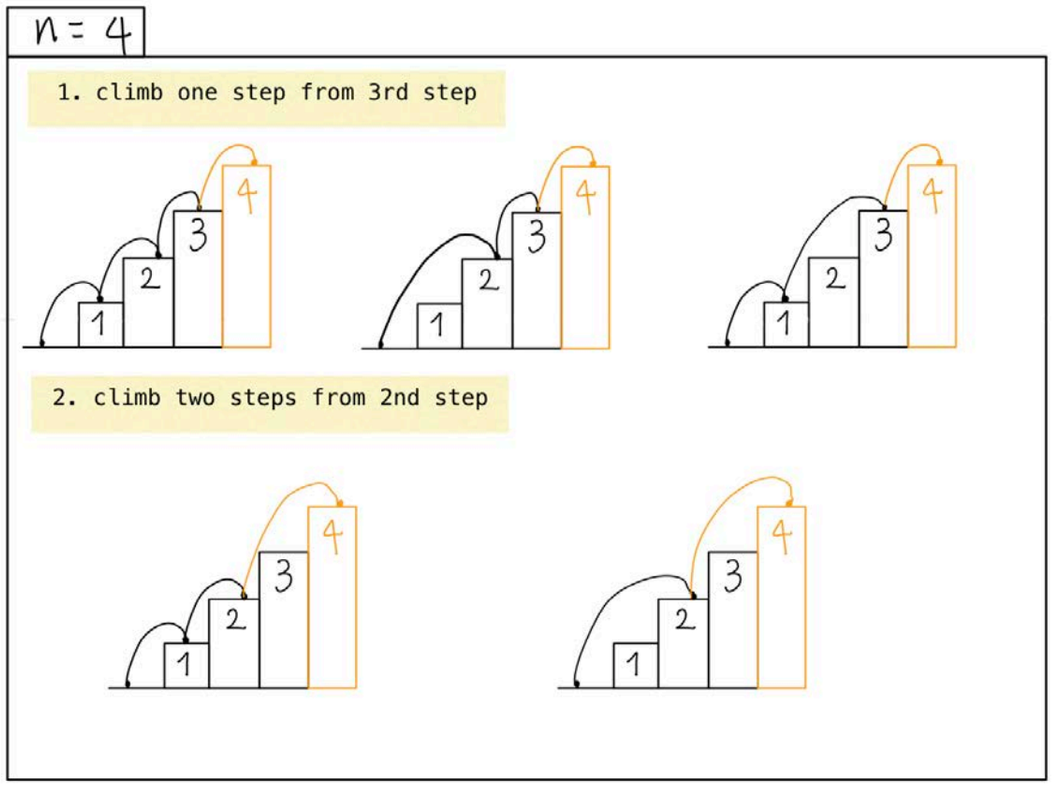
1. you climb one step from `(n-1)` th step
2. you climb two steps from `(n-2)`th step

* `n =1` and `n=2` are special cases because `n=1` does not have `(n-2)`th and `(n-1)` th step, and `n=2` does not have `(n-2)`th step.

See the graph:







See the code:

```python
# climb stair
# yx
n = 10
def climbStairs(n) -> int:
    res = [0,1,2]
    if n<3:
        return res[n]
    for i in range(3, n+1):
        res.append(res[-1] + res[-2])
    return res[-1]
print(climbStairs(10))
```

3.3 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 systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Example 1

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
```

Example 2

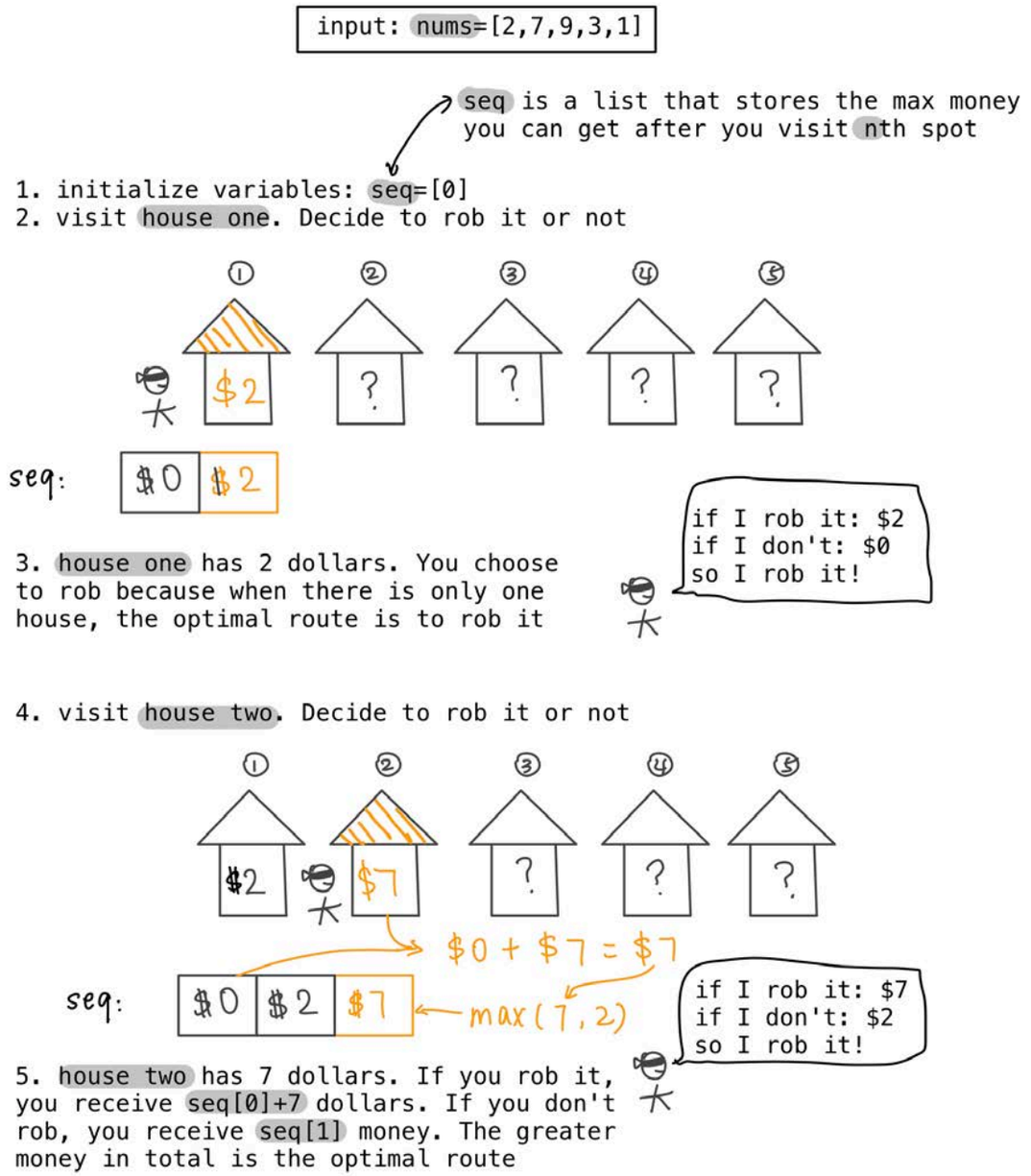
```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.
```

When we visit each house, we need to make a decision: **rob the house or not**. Each decision will affect our final result. We make the decision **based on the maximum money we can gain in total on this spot**. If we can receive more money in total by robbing the current house, we rob. If we cannot. we do not rob.

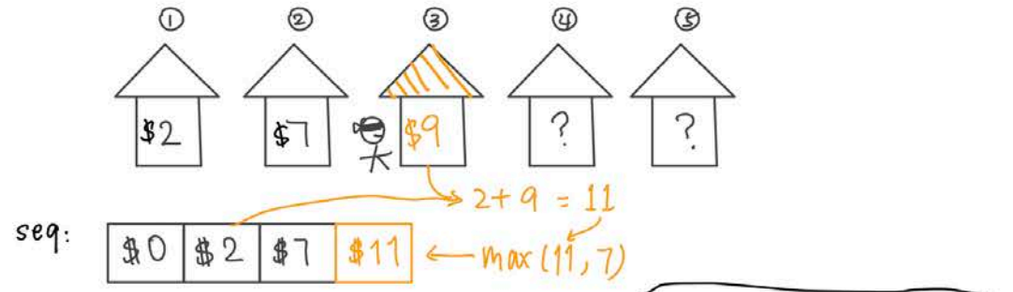
We **shouldn't consider** either the max money by robbing the next house or the previous optimal rob route. We only need to know whether we rob the current house or not.

*** the interval of two robbed houses can be either one or two, but can never be zero or three.**

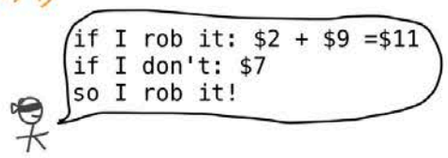
See the graph:

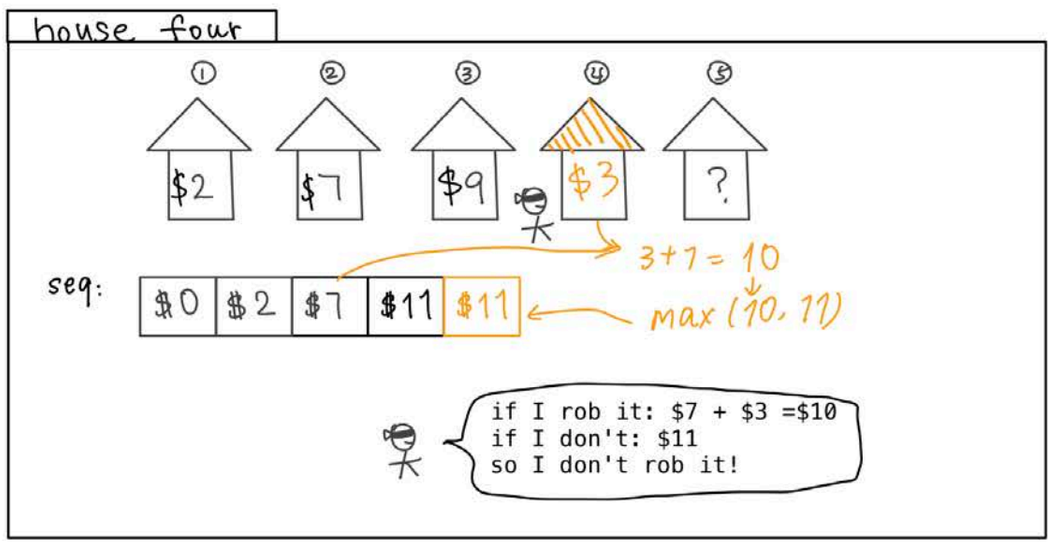


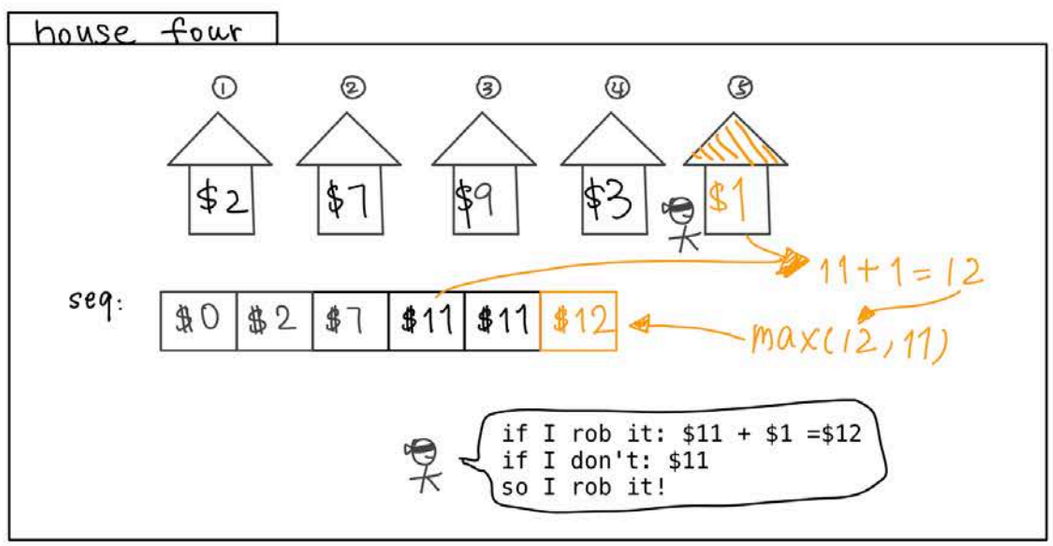
6. visit house three. Decide to rob it or not



7. house three has 9 dollars. If you rob it, you get seq[1]+9 dollars. If you don't, you get seq[2] dollars. Which decision you will make?







See the code:

```python
# house robber; dp
# oy
nums = [2,7,9,3,1]

def rob(nums)->int:
    if len(nums) ==1:
        return nums[0]

    seq = [0, nums[0]]
    for curr_house in range(2, len(nums)+1):
        seq.append(max(seq[curr_house-1], seq[curr_house-2]+nums[curr_house-1]))
    print(seq) # -> [0, 2, 7, 11, 11, 12]
    return seq[-1]
print(rob(nums)) # -> 12
```

This problem is a dynamic-programming problem because **the max money rubbed at each house includes the optimal route from its previous house**. Besides, the max money rubbed at each house is stored in a list: the step is called memoization.

3.4 Jump Game

You are given an integer array `nums`. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.
Return `true` if you can reach the last index, or false otherwise.

Constraints:

- 1 <= `nums`.length <= 104
- 0 <= `nums[i]` <= 105

Example 1

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from
index 0 to 1, then 3 steps to the
last index.
```

Example 2

```
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always ar-
rive at index 3 no matter what.
Its maximum jump length is 0,
which makes it impossible to
reach the last index.
```

Approach One: recursion

List all the possible jumping combinations and return `True` when one of the combinations reaches `nums`' the last index.

See the pseudocode:

```
SET findPath to False

DEFINE jump(current_index)
  IF findPath is True:
     STOP any other recursions

  IF current_index is greater than the last index of nums:
     CHANGE findPath to True
     STOP this recursion
  ELSE
     FOR step_length in range (1, the steps current_index can have +1)
        jump(current_index + step_length)
jump(0)
print(findPath)
```


See the code:

```python
# Jump Game. Brute Force
nums = [2,2,1,0,4] # input

findPath = False

def jump(curr):
  global findPath
  if findPath:
    return

  if curr >= len(nums)-1:
    findPath = True
    return
  for step in range(1, nums[curr]+1):
    jump(curr + step)

jump(0)
print(findPath)
```

This method is very very very slow and it exceeds the time limit.

Approach Two: Dynamic Programming

On each position, we make a decision: **jump or not**, based on **whether jumping at this moment can reach the maximum steps**. We don't care whether jumping now can reach the last index. To calculate the maximum steps, we use the same approach as the house robber problem. :

1. If we choose to jump at `n`th spot, we can jump `(n+num[n])`th spot
2. If we choose not to jump at `n`th spot, we can reach `(n-1)`th maximum steps.

In the end, if the maximum step is greater than or equal to the index of `nums`'s the last digit, we can jump to the last digit, and vice versa.

*we create a list to store the max steps we can reach on each spot (in a linear order)

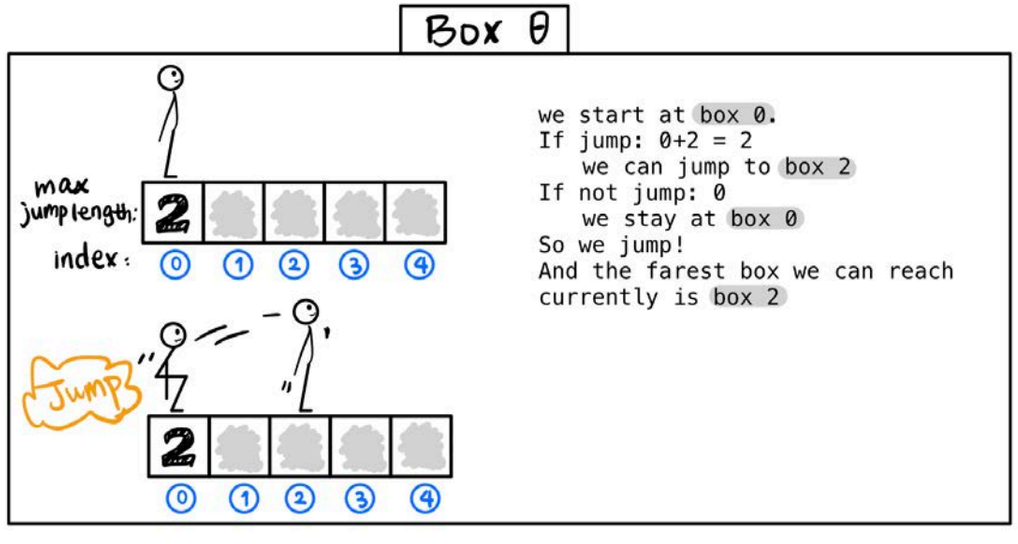
*it is easy for us to make some common mistakes when writing the base case of the problem

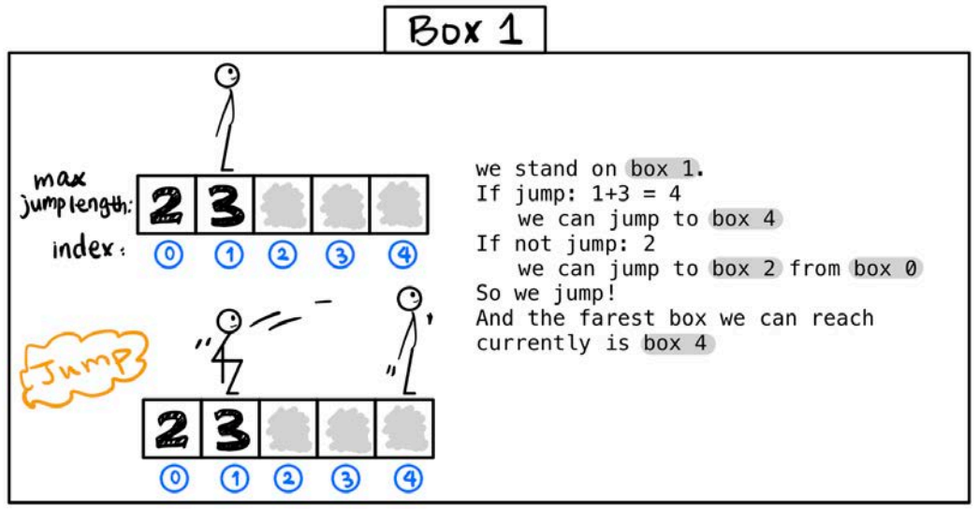
1. Should we return `False` when the current jump length is `0`?
   - In this example: `nums = [3,2,1,0,4]`. When we iterate the elements of `nums`, we find `nums[3] == 0`. If we return `False` because the current jump length is `0`, our answer is correct.
   - However, in another example: `nums = [3,0,1]`. If we return `False` because `nums[1] == 0`, our answer is incorrect. Because `num[0]` can jump beyond nums' last digit. Thus, we cannot use `if nums[i] == 0: return False` as a base case.
2. Should we use if the last maximum jump step is greater than or equal to the index of `nums`'s the last digit, return `True`. If not, return `False`?

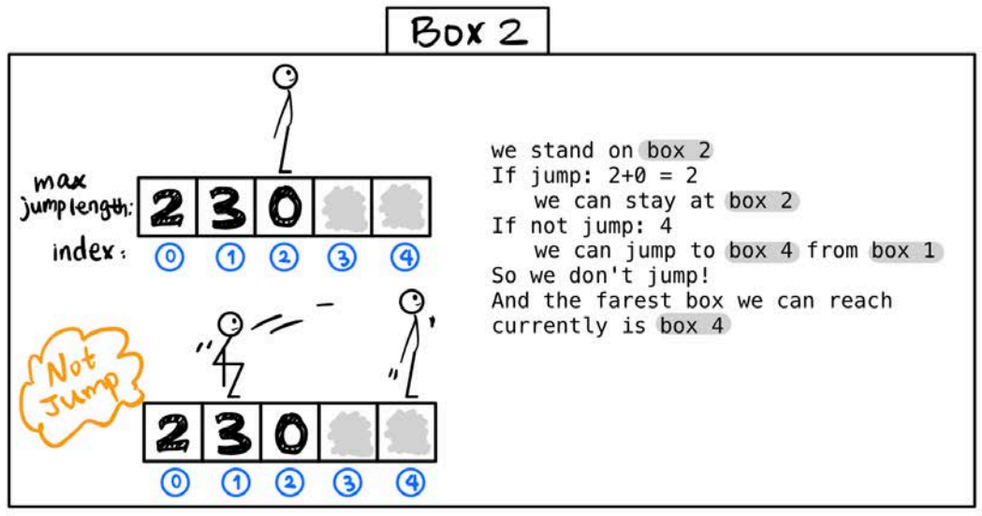
Let's see the illustrated graph first. And after that, we will talk about the base case.

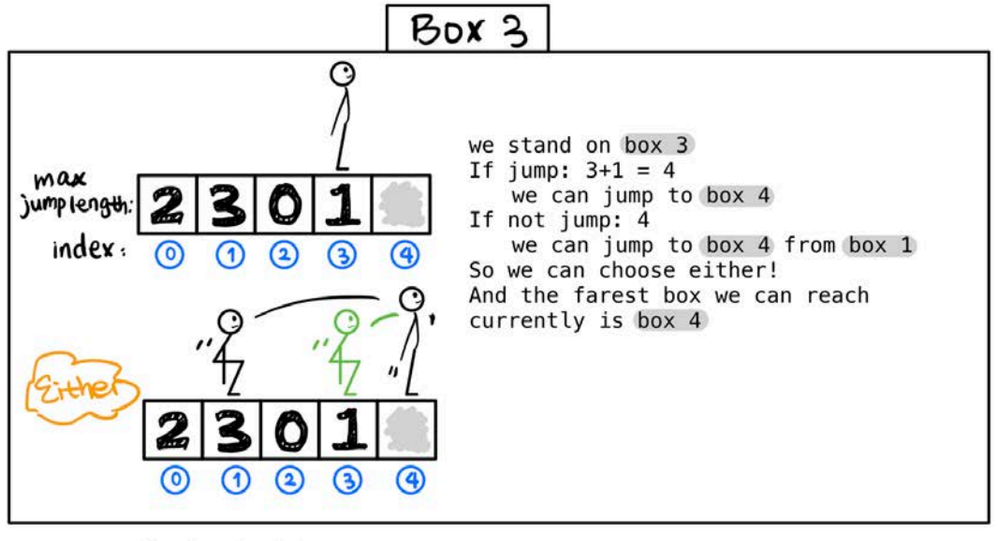
See the graph:

`Input: nums = [2,3,0,1,6]`









Now, let's check!
On `box 3`, The farest box we can reach is `box 4`, so return `True`
The value of `box 4` does not matter!

See the code:

```
# jump game; dp wiz wrong base case
# oy
nums = [2,3,0,1,6]
def canJump(nums) -> bool:
    if len(nums) ==1: # if nums only has one element
        return True
    max_steps = [nums[0]]
    for each_house in range(1, len(nums)-1):
        max_steps.append(max(nums[each_house]+each_house, max_steps[-1]))
    print(max_steps) # -> [2,4,4,4]
    return max_steps[-1] >= len(nums)-1
print(canJump(nums)) #-> True
```

What if `nums = [0,1,2,3]`? We cannot jump to any box from `box 0` because its jump length is 0. However, if we put the input `[0,1,2,3]` into the code above, it returns `True`. See the highlight part from the following code:

```
# jump game; dp wiz wrong base case
# oy
nums = [0,1,2,3]
def canJump(nums) -> bool:
    if len(nums) ==1: # if nums only has one element
        return True
    max_steps = [nums[0]]
    for each_house in range(1, len(nums)-1):
        max_steps.append(max(nums[each_house]+each_house, max_steps[-1]))
    print(max_steps) # -> [0,2,4]
    return max_steps[-1] >= len(nums)-1
print(canJump(nums)) #-> True
```

We must stop the function and return `False` when we stand on `box 0` and realize that we cannot jump to any other box. But, as we mentioned at the beginning, we cannot just say `` `if the current jump length is 0: return False` ``. Instead, our base case should be

`if the current jump length is 0 and we cannot jump beyond the current box from previous boxes.`

Translate it into Python:

`if nums[each_house] == 0 and each >= max_steps[-1]`

See the revised/correct code:

```
# jump game; dp
# oy
nums = [0,1,2,3]
def canJump(nums) -> bool:
    if len(nums) ==1: # if nums only has one element
        return True
    max_steps = [0] # cannot directly start with the first box
    for each_house in range(len(nums)-1):
        if nums[each_house] == 0 and each_house >= max_steps[-1]:
            print(max_steps) #-> [0]
            return False
        max_steps.append(max(nums[each_house]+each_house, max_steps[-1]))
    print(max_steps)
    return max_steps[-1] >= len(nums)-1
print(canJump(nums)) #-> False
```

See the pseudocode:

```
INPUT nums: a list of integers
IF only one element in nums:
    return True
IF nums[0] is 0:
    return False
SET max_lengths to []
ADD nums[0] into max_lengths

FOR each in range (1 to len(nums)-1):
    IF nums[each] is 0 and nums[each-1] cannot jump to any index
beyond nums[each]:
        return
    ELSE:
        max(nums[each] + each, max_lengths[-1])
        ADD the greater value into max_lengths
END for loop
PRINT if the max_lengths[-1] can reach the last index of nums
```

See the flowchart:

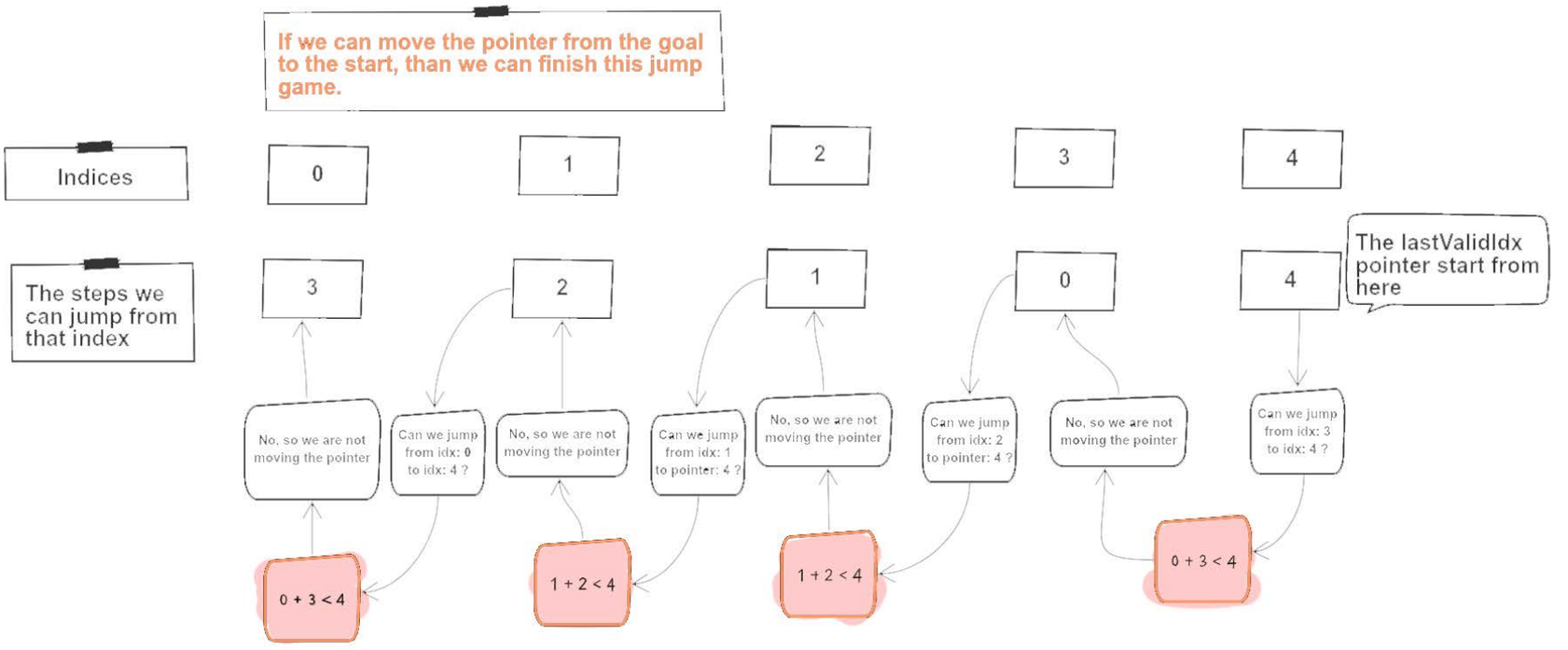


photo from https://medium.com/@kaido0121/leetcode-patterns-dynamic-programming-79e8acc6cf4e

Approach Three: Backtracking

start from the last index of `nums`, and go from the right of `nums` to the left of it. From right to left, we look for the `box Y` that can jump to or beyond the last box. After we find `box Y`, we continuously move from the right of `nums[:Y]` to the left of it, and we look for any other box that can jump to or beyond `box Y`. We return `True` until we start looking for any box that can jump to or beyond the starting box - `nums[0]`. Return `False` when we iterate all the boxes before `box Y` and find none of them can jump to or beyond `box Y`.

See the pseudocode:

```
INPUT nums: a list of integers
IF only one element in nums:
    return True
DEFINE backtracking (end):
    IF end can be reached by nums[0]:
        return True
    ITERATE each in all the element from nums[end] to nums[0];
count backwards
        IF each can reach end:
            return backtracking(each)
    END for loop
    return False
return backtracking(len(nums)-1)
```

See the code:

```python
# Jump Game; Backtracking

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

    if len(nums) ==1:
        return True

    def backtracking(end):
        if nums[0] >= end:
            return True
        for each in range(end-1, -1, -1):
            if nums[each] + each >= end:
                return backtracking(each)
        return False

    return backtracking(len(nums)-1)

```

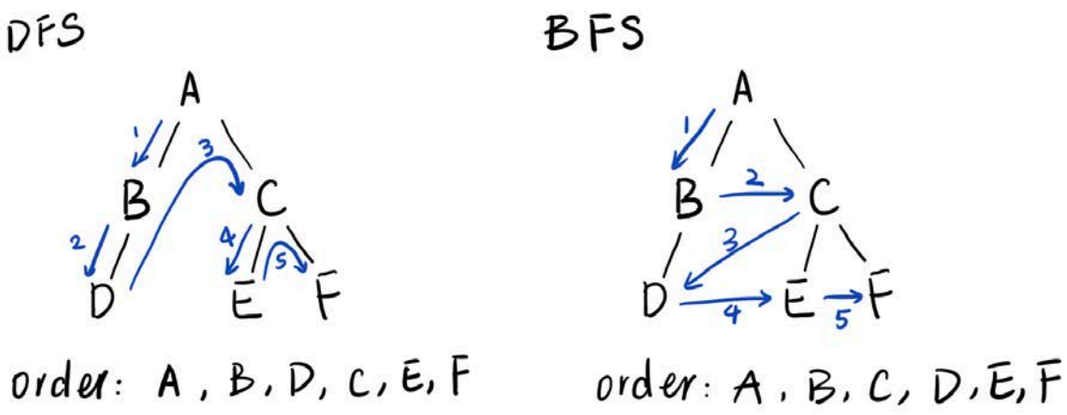
DEPTH FIRST SEARCH

Chapter Four

Depth-first search (DFS) is an algorithm technique for traversing or searching tree data structure. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. Each time the algorithm explores, a child node is found, the branch becomes longer. Then the recursion starts at the new child node; the child node becomes a parent node. When one child node returns None, the algorithm goes back to its parent node and continuously explores any other child nodes. The whole search stops when every element has been iterated.

DFS can also be used to check a cycle path.

On the contrary, BFS (Breadth-First-Search) is used to find the shortest path.



4.1 Letter Combination

Given a string containing digits from `2-9` inclusive, return all possible letter combinations that the number could represent. Return the answer **in any order**.
A mapping of digit to letters (just like on the telephone buttons) is given below (left pic). Note that `1` does not map to any letters.

Constraints:

* 0 <= `digits`.length <= 4
* `digits[i]` is a digit in the `range ['2', '9']`.

Example 1

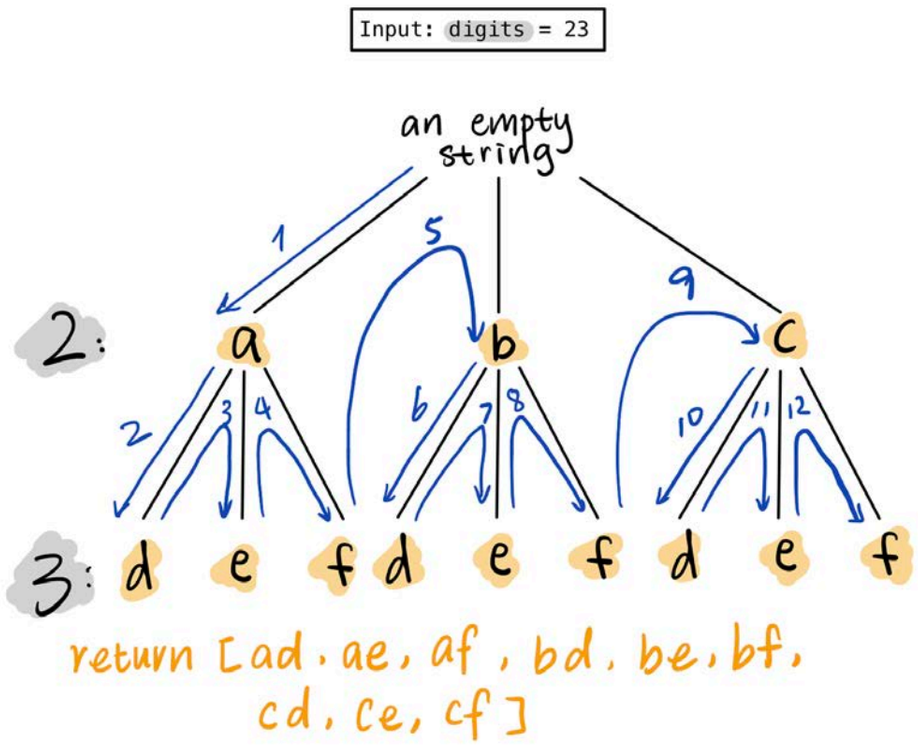
```
Input: digits = "23"
Output: ["ad","ae","af","b-
d","be","bf","cd","ce","cf"]
```

Example 2

```
Input: digits = "2"
Output: ["a","b","c"]
```

In DFS, we find each possible letter combination one by one and store it into a list res. In the end, we return res. We use recursions to implement DFS in this problem.

The graph illustrates the order of finding each letter combination



See the code:

```python
# letter combination; dfs
# oy
digits = "23"
# create a dict to store phone numbers and their corresponding characters
dic = {1:[], 2:['a','b','c'], 3:['d','e','f'], 4:['g','h','i'], 5:['j','k','l'], 6:['m','n','o'], 7:['p','q','r','s'], 8:['t','u','v'], 9:['w','x','y','z']}

res = [] # create a list res to store all the possible combinations
if digits == "": # return None if the input has no number
    print([])

def dfs(comb, index):
    if len(comb) == len(digits): # a complete combination
        res.append(comb)
        return
    for one_digit in dic[int(digits[index])]: # recursion
        dfs(comb + one_digit, index +1)
dfs("", 0)
print(res)
```


4.2 Permutations

Given an array `nums` of **distinct** integers, return all the possible permutations. You can return the answer **in any order**.

Constraints:

* 1 <= `nums.length` <= 6
* -10 <= `nums[i]` <= 10
* All the integers of nums **are unique**.

Example 1

```
Input: nums = [1,2,3]
Output: [[1,2,3],
     [1,3,2],
     [2,1,3],
     [2,3,1],
     [3,1,2],
     [3,2,1]]
```

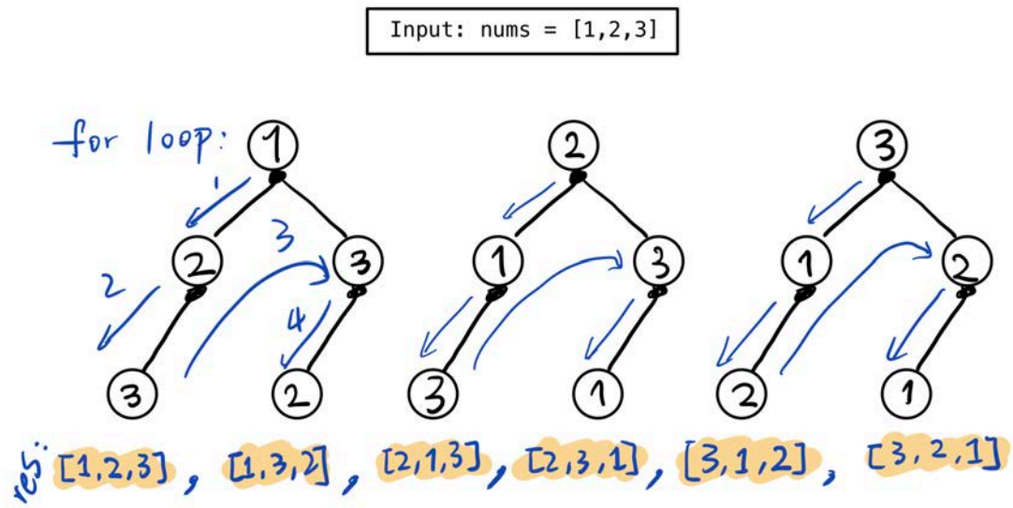
Example 2

```
Input: nums = [0,1]
Output:[[0,1],
     [1,0]]
```

Example 3

```
Input: nums = [1]
Output: [[1]]
```

The graph below illustrates the order of finding each permutation:



See the code:

```
1   # permutation; dfs
2   # yx
3   nums = [1,2,3] # List[int]
4   res = []
5   def dfs(comb, non_visited):
6       if len(non_visited) == 0:
7           res.append(comb)
8           return
9       for child in non_visited:
10          next_non_visited = non_visited.copy()
11          next_non_visited.remove(child)
12          dfs(comb+[child], next_non_visited) # `append` returns none
13  dfs([], nums)
14  print(res)
```

In the code, `non_visited` is a list that tells what child nodes the current `comb` has. Take `nums = [1,2,3]` as an example; if the current `comb` is "`1`", its child nodes are in `non_visited`: [2,3]. Thus, for the next step, the current `comb` can either `2` or `3`. For the next run, the explored child node has been deleted from `non_visited`. When there is no element left in `non_visited`, we append the `comb` into `res`.

Besides, from line 10 to 11, we make a copy of `non_visited` and change the copy, so that the changes won't affect any other recursions.

4.3 Permutation II

Given a collection of numbers, `nums`, that might **contain duplicates**, return all possible **unique** permutations **in any order.**
Constraints:

1 <= `nums.length` <= 8
-10 <= `nums[i]` <= 10

Example 1

```
Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there
are 3 numbers, so all numbers
are in the range [0,3]. 2 is the
missing number in the range since
it does not appear in nums.
```

Example 2

```
Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there
are 3 numbers, so all numbers
are in the range [0,3]. 2 is the
missing number in the range since
it does not appear in nums.
```

In this problem, numbers are not unique. If we use the code of last problem, the output has repetitive combinations:

```
Input: nums = [1,1,2]
Output: [[1,1,2], [1,2,1], [2,1,1], [1,1,2], [1,2,1], [2,1,1]]
```

To avoid repetitions, we have several approaches

Approach One: Eliminate Redundant Combinations in the End

We still find out all the possible combinations. But, before the code appends `comb` into `res`, we add an if statement: if the current comb has appended into res before, we don't append it again.

`if comb in res: return`

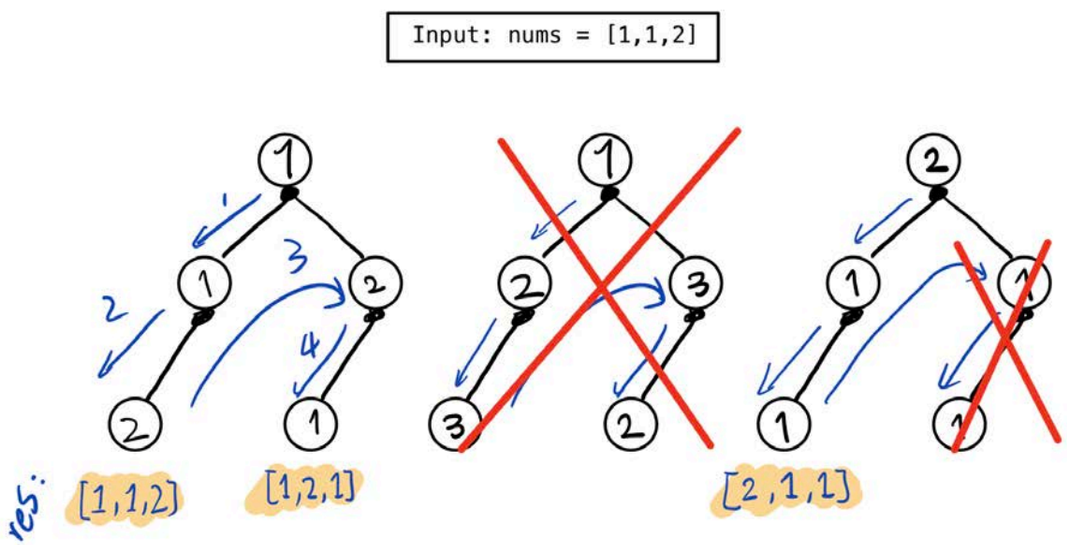
See the code (line 7 and 8):

```
1   # permutation ii; wrong
2   # yx
3   nums = [1,1,2]
4   res = []
5   def dfs(comb, non_visited):
6       if len(non_visited) == 0:
7           if comb in res:
8               return
9           res.append(comb)
10          return
11      for child in non_visited:
12          next_non_visited = non_visited.copy()
13          next_non_visited.remove(child)
14          dfs(comb+[child], next_non_visited)
15  dfs([], nums)
16  print(res)
```

However, this approach is inefficient. Can't we eliminate redundant combinations at first?

Approach Two: Eliminate Redundant Calculation At First

The following graph illustrates which part is redundant and should be eliminated at first



We create a dictionary, whose key is each unique number in nums, and whose value is the number of occurrences of the key.

```
>>> nums = [7,7,8,9,9]
dic = {7: 2, 8:1, 9:2}
```

Then we only make different combinations based on the keys of dic, which are all unique numbers.

See the code

```
1   # permutation II
2   # oy
3   nums = [1,1,2]
4   res = [] # store all the combinations
5
6   # create a dictionary that records the occurance of each int
7   dic = {}
8   for each in nums:
9       if each not in dic:
10          dic[each] = 1 # assign 1 to a new key
11      else:
12          dic[each] += 1
13
14
15  def dfs (comb, visited: dict):
16      if len(comb) == len(nums):
17          res.append(comb)
18          return
19
20      for i in visited: # iterate its key
21          if visited[i] ==0: # i has been used
22              continue
23          else:
24              next_visited = visited.copy()
25              next_visited[i] -= 1
26              dfs(comb+[i], next_visited)
27  dfs([], dic)
28  print( res )
```

4.4 Combination Sum

Given an array of **distinct** integers `candidates` and a target integer `target`, return *a list of **all unique combinations*** of `candidates` *where the chosen numbers sum to target.* You may return the combinations **in any order**.

The **same** number may be chosen from `candidates` an **unlimited number of times.** Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is **guaranteed** that the number of unique combinations that sum up to target is less than 150 combinations for the given input.
*candidates is not a sorted list

Example 1

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used
multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

Example 2

```
Input: candidates = [2], target = 1
Output: []
```

Example 3

```
Input: candidates = [2,3,5], target = 8
Output: [[2,2,2,2],[2,3,3],[3,5]]
```


1. In DFS, we explore each candidate and continue exploring the child nodes of each candidate. Since numbers in the given array can be used multiple times in this problem, the child nodes must include their parent node.
    i. for example, `candidates = [2,3,4], target = 5`. When the parent node is `2`, it can choose one of `[2,3,4]` for the next run. The parent node `2`'s child nodes are `[2,3,4]` but not `[3,4]`

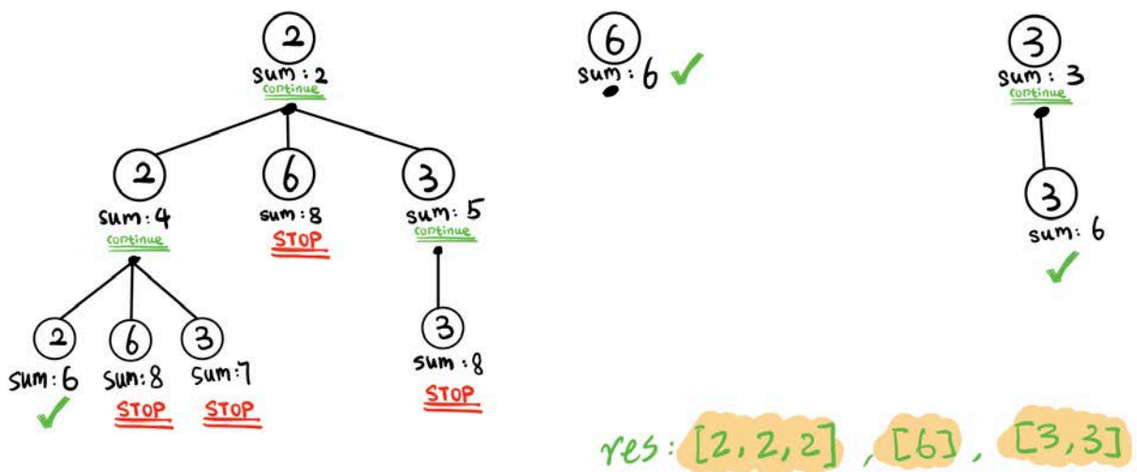
2. To avoid redundant arrays, we need to slice out unnecessary parts. The number that has been used before is unnecessary.
    i. For example, `candidates = [2,3,4], target = 5`. When we have iterated all the possible combinations start at `2` and continue to find all the combinations start at `3`, the child nodes don't need to include the previous starting number —-`2`. Thus, the child nodes of `3` should be `[3,4]` but not `[2,3,4]`.

See the pseudocode:

```
candidates = a list of integers
INIT a list results to store combinations
INIT a function dfs(result, target, last_index), where result:[], target:
target number, last_index: 0
    IF the sum of result is equal to the target:
        APPEND the result into results
        STOP
    IF the sum of result is greater than target:
        STOP
    IF the sum of result is smaller than target:
        FOR each element in candidates [last_index:]:
            dfs(result+[this element], target, the index of this element)
```

See the graph:

```
Input: candidates = [2,6,3] target = 6

when sum < target: continue
when sum == target: ✓
when sum > target: STOP
```



See the code:

```python
# combination sum
# yx
candidates = [2,3,6,7]
target = 7
res = [] # a list to store combinations
def dfs(comb, target, last_index):
    summ = sum(comb)
    if summ == target:
        res.append(comb)
        return
    elif summ < target:
        for each in range(last_index, len(candidates)):
            dfs(comb+[candidates[each]], target, each)

dfs([], target, 0)
print(res)
```


4.5 Combination Sum II

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all **unique** combinations in `candidates` where the candidate numbers sum to `target`.
Each number in `candidates` may only be used once in the combination.
Note: The solution set must not contain **duplicate** combinations.

Example 1

```
Input: candidates =
[10,1,2,7,6,1,5], target = 8
Output: [[1,1,6],
         [1,2,5],
         [1,7],
         [2,6]]
```

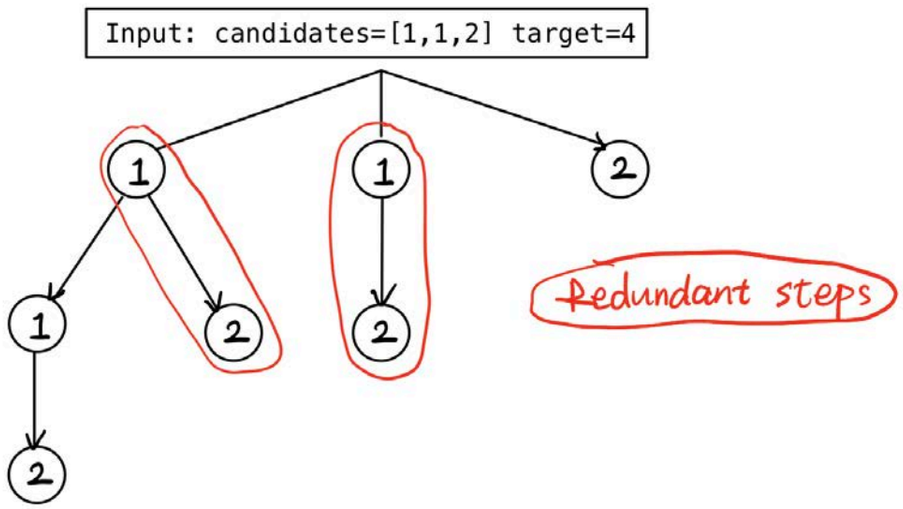
Example 2

```
Input: candidates = [2,5,2,1,2],
target = 5
Output: [[1,2,2],
         [5]]
```

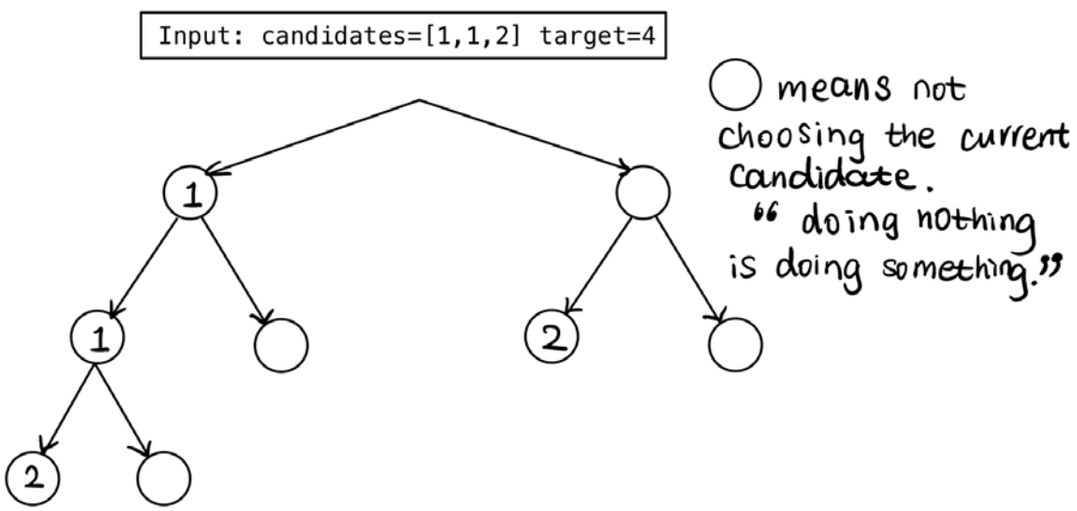
There are two differences between this problem and the earlier problem:

1. In this problem, each number in the input is **NOT unique**. The implication of this difference is that we need some mechanism to avoid generating duplicate combinations.
2. In this problem, each number can be **used only once**. The implication of this difference is that once a number is chosen as a candidate in the combination, it will not appear again as a candidate later.

If we put the input `[1,1,2]` into the code of last problem, it returns duplicate results, as the following graph.



To avoid redundant steps, we use backtracking here. We iterate through the sorted input array, via backtracking to build the combinations. In one recursion, we only use a repeated number once, which means we need to skip any other numbers of the same value in one recursion. See the following graph:



See the code:

```
1   # combination sum 2
2   # code from NeetCode on Youtube
3   candidates = [1,1,1,2]
4   target = 3
5   res = [] # a list to store combinations
6   candidates.sort() # sort the given list first
7   def backtracking(comb, pos, target):
8       if target == 0:
9           res.append(comb)
10          return
11      elif target < 0:
12          return
13      prev_child = -1
14      for child in range(pos, len(candidates)):
15          if candidates[child] == prev_child:
16              continue
17          backtracking(comb+[candidates[child]], child+1, target-candidates[child])
18          prev_child = candidates[child]
19  backtracking([], 0, target)
20  print(res) #-> [[1,1,1],[1,2]]
```

On line 13, since all the elements in `candidates` are positive integers, we assign a negative value to `prev_child`, so that the if statement on line 15 and 16 will never execute on the first iteration.

Watch ***https://www.youtube.com/watch?v=rSA3t6BDDwg*** for more detailed explanations.

OTHERS

Chapter Five

This chapter has two problems. I did not categorize them, because they don't fit any of the previous four chapters I wrote. Moreover, neither of my limited knowledge nor energy are able to encourage me to write more chapters for these two problems. I just want to show their illustrated graphs. And they are fun.

They are easier than all the previous problems. :D

5.1 Two Sums

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they can add up to `target`.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer **in any order.**

Example 1

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2

```
Input: nums = [3,2,4], target =
6
Output: [1,2]
```

Example 3

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

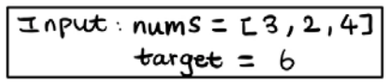
Let's imagine each element in `nums` is a card that has a value on it and an index.

Approach One: Brute Force

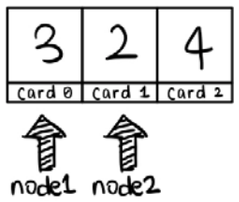
Create two nodes: `node1` points towards `card one`, `node2` iterates all the cards after `card one`. Every time `node2` points at a different card, the code calculates the sum of `node1` and `node2`. After `node2` finishes iterating all the cards that follow `card one`, `node1` moves to `card two` and `node2` continuously iterates all the cards following `card two`. The recursion stops when the sum of `node1` and `node2` equals to `target`.

Time complexity: $O(n^2)$

See the graph:

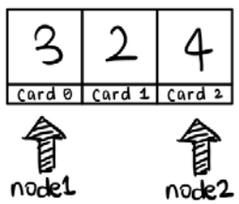


1. Initialize `node1` and `node2`



2. calculate the sum.
the sum does not equal to `target`,
so `node2` continuously moves

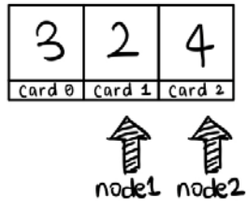
Sum = 3 + 2
= 5
5 ≠ 6



3. calculate the sum.
the sum does not equal to `target`,
so `node2` continuously moves

sum = 3 + 4
= 7
7 ≠ 6

4. `node2` has reached the last element of `nums`, which means its iteration has ended.
Let's move `node1`



5. calculate the sum.
the sum equals to `target`, return.

sum = 2 + 4
= 6
6 = 6

return node1 and node2

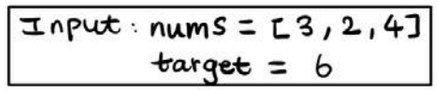
See the code:

```python
# two sums; brute force
nums = [2,7,11,15]
target = 9

for node1 in range(len(nums)-1):
    for node2 in range(node1+1, len(nums)):
        if nums[node1] + nums[node2] == target:
            print(node1, node2)

```

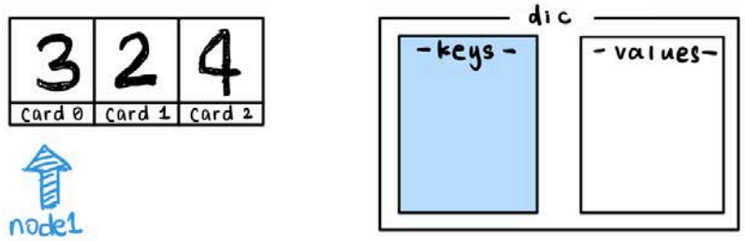
Approach Two: One-pass Hash Table

we create a dictionary `dic` and then iterate the given array. On each iteration, we check if the complement of the current number has been visited before in `dic`. If yes, we return the indices of the current number and its complement. If not, we append the value and index of the current number into `dic`. Since this problem guarantees that each input has exactly one solution, we don't need to have a base case.

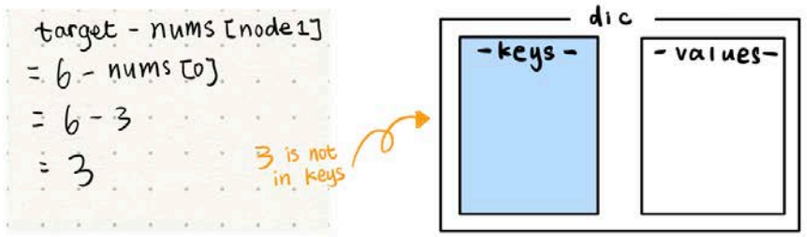
See the graph:



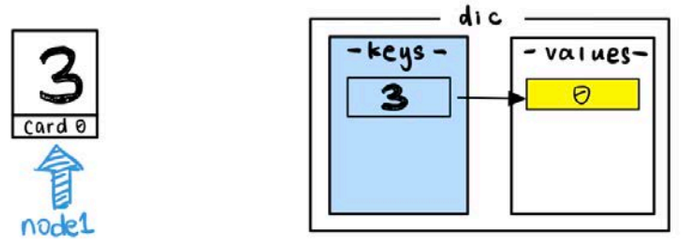
1. Initialize `node1` = 0; `dic` = {}
2. `node1` iterates `nums`



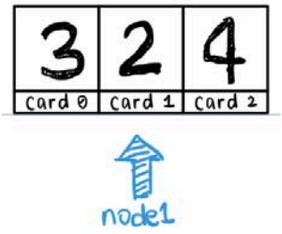
3. `node1` points towards `card0`
4. the value of `card0` is 3. Check if (target-3) in `dic`



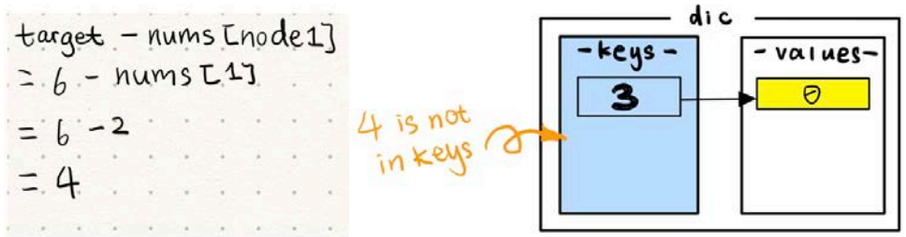
5. store the value of `card0` (3) and index of the card (0)



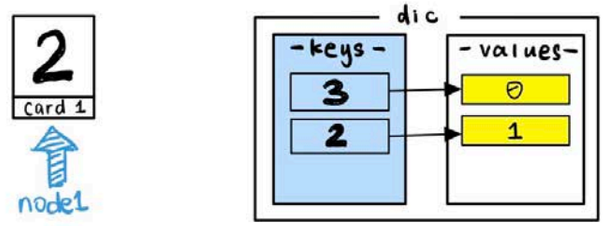
6. `node1` moves to the next card: `card1`



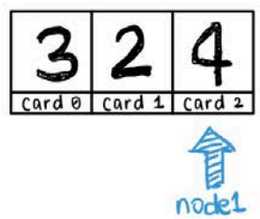
7. the value of `card1` is 2. Check if (target-2) in `dic`



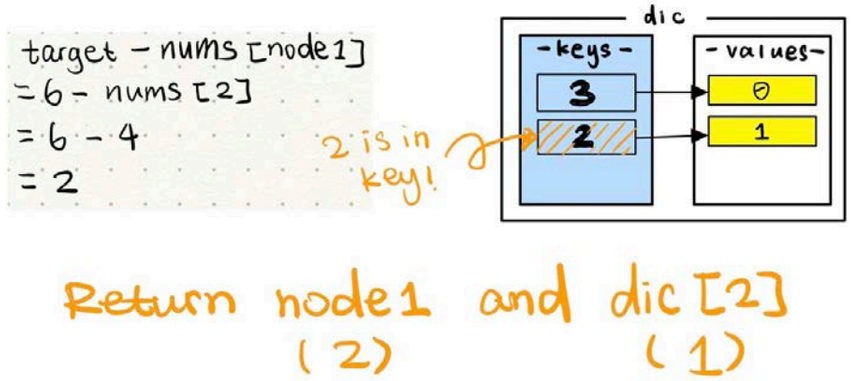
8. store the value of `card1` (2) and index of the card (1)



9. `node1` moves to the next card: `card2`



10. the value of `card2` is 4. Check if `(target-4)` in `dic`



See the code:

```
# two sums;
nums = [2,7,11,15]
target = 9
dic = {}
for each in range(len(nums)):
    if (target-nums[each]) in dic:
        print([dic[target-nums[each]], each])
        break
    dic[nums[each]] = each
```


5.2 Valid Parentheses

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by **the same type of** brackets.
2. Open brackets must be closed **in the correct order.**

Example 1

```
Input: s = "()"
Output: true
```

Example 2

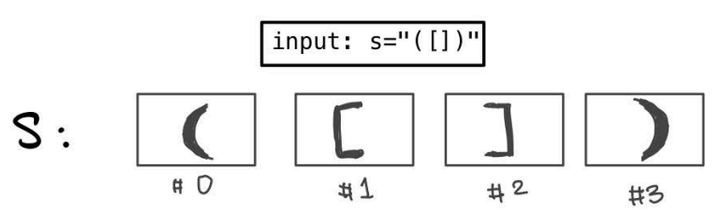
```
Input: s = "()[]{}"
Output: true
```

Example 3

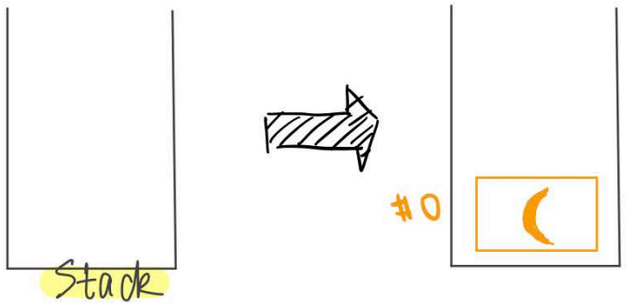
```
Input: s = "(]"
Output: false
```

For this problem, we use the data structure stack. A stack (sometimes called a "push-down stack") is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the "top." The end opposite the top is known as the "base."

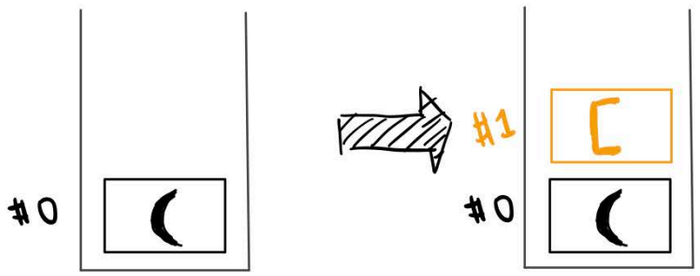
See the graph:



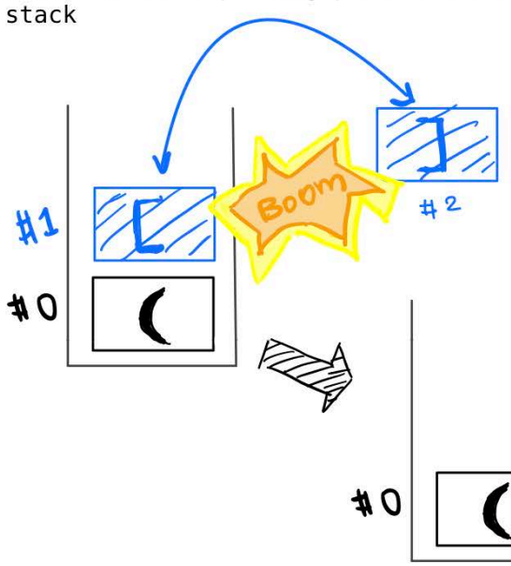
1. create a stack.
2. the first element of `s` is a left-side parenthese, so push it into the stack



3. the second element of `s` is a left-side parenthese, so push it into the stack

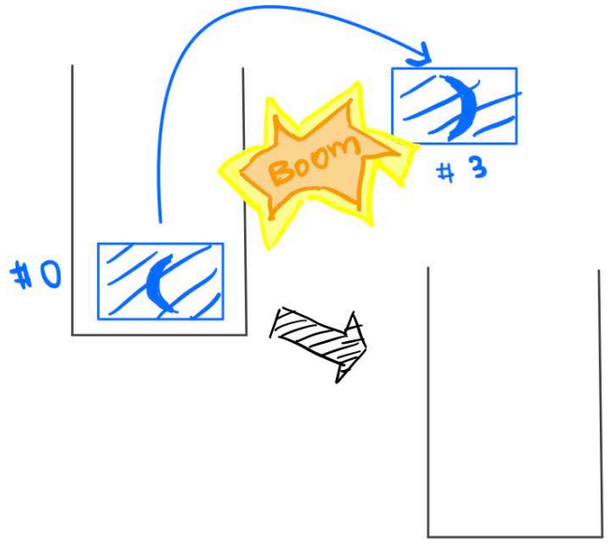


4. the third element of `s` is a right-side parenthese, so check if it is the corresponding parenthese of the top parenthese of the stack



5. the third element is the corresponding parenthese, so pop the top parenthese of the stack

6. the fourth element of `s` is a right-side parenthese, so check if it is the corresponding parenthese of the top parenthese of the stack



7. all the elments of `s` have been iterated. Now check if the stack is empty. If yes, all the parentheses of s are valid. If not, not all the parentheses of s are valid.

Return True

See the code:

```python
# valid parentheses
s = '([])'
# create a dictionary and an emplty list (stack)
dic = {'[':']', '{':'}','(':')'}
stack = []
def isValid(s) -> bool:
    for each in range(len(s)):
        if s[each] in dic: # if the curr element is right-sided
            stack.append(s[each])
        else:
            if len(stack)>0 and s[each] == dic[stack[-1]]:
                stack.pop()
            else:
                return False
    return len(stack) == 0
print(isValid(s))
```

Thank you