>> Intro

Binary Search is quite easy to understand conceptually. Basically, it splits the search space into two halves and only keep the half that probably has the search target and throw away the other half that would not possibly have the answer. In this manner, we reduce the search space to half the size at every step, until we find the target. Binary Search helps us reduce the search time from linear O(n) to logarithmic O(log n). **But when it comes to implementation, it's rather difficult to write a bug-free code in just a few minutes**. Some of the most common problems include:

* When to exit the loop? Should we use left < right or left <= right as the while loop condition?
* How to initialize the boundary variable left and right?
* How to update the boundary? How to choose the appropriate combination from left = mid, left = mid + 1 and right = mid, right = mid - 1?

A rather common misunderstanding of binary search is that people often think this technique could only be used in simple scenario like "Given a sorted array, find a specific value in it". As a matter of fact, it can be applied to much more complicated situations.

After a lot of practice in LeetCode, I've made a powerful binary search template and solved many Hard problems by just slightly twisting this template. I'll share the template with you guys in this post. **I don't want to just show off the code and leave. Most importantly, I want to share the logical thinking: how to apply this general template to all sorts of problems**. Hopefully, after reading this post, people wouldn't be pissed off anymore when LeetCoding, "This problem could be solved with binary search! Why didn't I think of that before!"

>> Most Generalized Binary Search

Suppose we have a **search space**. It could be an array, a range, etc. Usually it's sorted in ascending order. For most tasks, we can transform the requirement into the following generalized form:

**Minimize k , s.t. condition(k) is True**

The following code is the most generalized binary search template:

def binary\_search(array) -> int:

def condition(value) -> bool:

pass

left, right = min(search\_space), max(search\_space) # could be [0, n], [1, n] etc. Depends on problem

while left < right:

mid = left + (right - left) // 2

if condition(mid):

right = mid

else:

left = mid + 1

return left

What's really nice of this template is that, for most of the binary search problems, **we only need to modify three parts after copy-pasting this template, and never need to worry about corner cases and bugs in code any more**:

* Correctly initialize the boundary variables left and right to specify search space. Only one rule: set up the boundary to **include all possible elements**;
* Decide return value. Is it return left or return left - 1? Remember this: **after exiting the while loop, left is the minimal k​ satisfying the condition function**;
* Design the condition function. This is the most difficult and most beautiful part. Needs lots of practice.

Below I'll show you guys how to apply this powerful template to many LeetCode problems.

>> Basic Application

[278. First Bad Version [Easy]](https://leetcode.com/problems/first-bad-version/)

You are a product manager and currently leading a team to develop a new product. Since each version is developed based on the previous version, all the versions after a bad version are also bad. Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad. You are given an API bool isBadVersion(version) which will return whether version is bad.

**Example:**

Given n = 5, and version = 4 is the first bad version.

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

First, we initialize left = 1 and right = n to include all possible values. Then we notice that we don't even need to design the condition function. It's already given by the isBadVersion API. Finding the first bad version is equivalent to finding the minimal k satisfying isBadVersion(k) is True. Our template can fit in very nicely:

class Solution:

def firstBadVersion(self, n) -> int:

left, right = 1, n

while left < right:

mid = left + (right - left) // 2

if isBadVersion(mid):

right = mid

else:

left = mid + 1

return left

[69. Sqrt(x) [Easy]](https://leetcode.com/problems/sqrtx/)

Implement int sqrt(int x). Compute and return the square root of *x*, where *x* is guaranteed to be a non-negative integer. Since the return type is an integer, the decimal digits are truncated and only the integer part of the result is returned.

**Example:**

Input: 4

Output: 2

Input: 8

Output: 2

Easy one. First we need to search for minimal k satisfying condition k^2 > x, then k - 1 is the answer to the question. We can easily come up with the solution. Notice that I set right = x + 1 instead of right = x to deal with special input cases like x = 0 and x = 1.

def mySqrt(x: int) -> int:

left, right = 0, x + 1

while left < right:

mid = left + (right - left) // 2

if mid \* mid > x:

right = mid

else:

left = mid + 1

return left - 1 # `left` is the minimum k value, `k - 1` is the answer

[35. Search Insert Position [Easy]](https://leetcode.com/problems/search-insert-position/)

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You may assume no duplicates in the array.

**Example:**

Input: [1,3,5,6], 5

Output: 2

Input: [1,3,5,6], 2

Output: 1

Very classic application of binary search. We are looking for the minimal k value satisfying nums[k] >= target, and we can just copy-paste our template. Notice that our solution is correct regardless of whether the input array nums has duplicates. Also notice that the input target might be larger than all elements in nums and therefore needs to placed at the end of the array. That's why we should initialize right = len(nums) instead of right = len(nums) - 1.

class Solution:

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

left, right = 0, len(nums)

while left < right:

mid = left + (right - left) // 2

if nums[mid] >= target:

right = mid

else:

left = mid + 1

return left

>> Advanced Application

The above problems are quite easy to solve, because they already give us the array to be searched. We'd know that we should use binary search to solve them at first glance. However, **more often are the situations where the search space and search target are not so readily available**. Sometimes we won't even realize that the problem should be solved with binary search -- we might just turn to dynamic programming or DFS and get stuck for a very long time.

As for the question "When can we use binary search?", my answer is that, **If we can discover some kind of monotonicity, for example, if condition(k) is True then condition(k + 1) is True, then we can consider binary search**.

[1011. Capacity To Ship Packages Within D Days [Medium]](https://leetcode.com/problems/capacity-to-ship-packages-within-d-days/)

A conveyor belt has packages that must be shipped from one port to another within D days. The i-th package on the conveyor belt has a weight of weights[i]. Each day, we load the ship with packages on the conveyor belt (in the order given by weights). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within D days.

**Example :**

Input: weights = [1,2,3,4,5,6,7,8,9,10], D = 5

Output: 15

Explanation:

A ship capacity of 15 is the minimum to ship all the packages in 5 days like this:

1st day: 1, 2, 3, 4, 5

2nd day: 6, 7

3rd day: 8

4th day: 9

5th day: 10

Note that the cargo must be shipped in the order given, so using a ship of capacity 14 and splitting the packages into parts like (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) is not allowed.

Binary search probably would not come to our mind when we first meet this problem. We might automatically treat weights as search space and then realize we've entered a dead end after wasting lots of time. In fact, we are looking for the minimal one among all feasible capacities. We dig out the monotonicity of this problem: if we can successfully ship all packages within D days with capacity m, then we can definitely ship them all with any capacity larger than m. Now we can design a condition function, let's call it feasible, given an input capacity, it returns whether it's possible to ship all packages within D days. This can run in a greedy way: if there's still room for the current package, we put this package onto the conveyor belt, otherwise we wait for the next day to place this package. If the total days needed exceeds D, we return False, otherwise we return True.

Next, we need to initialize our boundary correctly. Obviously, capacity should be at least max(weights), otherwise the conveyor belt couldn't ship the heaviest package. On the other hand, capacity need not be more than sum(weights), because then we can ship all packages in just one day.

Now we've got all we need to apply our binary search template:

def shipWithinDays(weights: List[int], D: int) -> int:

def feasible(capacity) -> bool:

days = 1

total = 0

for weight in weights:

total += weight

if total > capacity: # too heavy, wait for the next day

total = weight

days += 1

if days > D: # cannot ship within D days

return False

return True

left, right = max(weights), sum(weights)

while left < right:

mid = left + (right - left) // 2

if feasible(mid):

right = mid

else:

left = mid + 1

return left

[410. Split Array Largest Sum [Hard]](https://leetcode.com/problems/split-array-largest-sum/)

Given an array which consists of non-negative integers and an integer *m*, you can split the array into *m* non-empty continuous subarrays. Write an algorithm to minimize the largest sum among these *m* subarrays.

**Example:**

Input:

nums = [7,2,5,10,8]

m = 2

Output:

18

Explanation:

There are four ways to split nums into two subarrays. The best way is to split it into [7,2,5] and [10,8], where the largest sum among the two subarrays is only 18.

If you take a close look, you would probably see how similar this problem is with LC 1011 above. Similarly, we can design a feasible function: given an input threshold, then decide if we can split the array into several subarrays such that every subarray-sum is less than or equal to threshold. In this way, we discover the monotonicity of the problem: if feasible(m) is True, then all inputs larger than m can satisfy feasible function. You can see that the solution code is exactly the same as LC 1011.

def splitArray(nums: List[int], m: int) -> int:

def feasible(threshold) -> bool:

count = 1

total = 0

for num in nums:

total += num

if total > threshold:

total = num

count += 1

if count > m:

return False

return True

left, right = max(nums), sum(nums)

while left < right:

mid = left + (right - left) // 2

if feasible(mid):

right = mid

else:

left = mid + 1

return left

But we probably would have doubts: It's true that left returned by our solution is the minimal value satisfying feasible, but how can we know that we can split the original array to **actually get this subarray-sum**? For example, let's say nums = [7,2,5,10,8] and m = 2. We have 4 different ways to split the array to get 4 different largest subarray-sum correspondingly: 25:[[7], [2,5,10,8]], 23:[[7,2], [5,10,8]], 18:[[7,2,5], [10,8]], 24:[[7,2,5,10], [8]]. Only 4 values. But our search space [max(nums), sum(nums)] = [10, 32] has much more that just 4 values. That is, no matter how we split the input array, we cannot get most of the values in our search space.

Let's say k is the minimal value satisfying feasible function. We can prove the correctness of our solution with **proof by contradiction**. Assume that no subarray's sum is equal to k, that is, every subarray sum is less than k. The variable total inside feasible function keeps track of the total weights of current load. If our assumption is correct, then total would always be less than k. As a result, feasible(k - 1) must be True, because total would at most be equal to k - 1 and would never trigger the if-clause if total > threshold, **therefore feasible(k - 1) must have the same output as feasible(k), which is True**. But we already know that k is the minimal value satisfying feasible function, **so feasible(k - 1) has to be False, which is a contradiction**. So our assumption is incorrect. Now we've proved that our algorithm is correct.

[875. Koko Eating Bananas [Medium]](https://leetcode.com/problems/koko-eating-bananas/)

Koko loves to eat bananas. There are N piles of bananas, the i-th pile has piles[i] bananas. The guards have gone and will come back in H hours. Koko can decide her bananas-per-hour eating speed of K. Each hour, she chooses some pile of bananas, and eats K bananas from that pile. If the pile has less than K bananas, she eats all of them instead, and won't eat any more bananas during this hour.

Koko likes to eat slowly, but still wants to finish eating all the bananas before the guards come back. **Return the minimum integer K such that she can eat all the bananas within H hours**.

**Example :**

Input: piles = [3,6,7,11], H = 8

Output: 4

Input: piles = [30,11,23,4,20], H = 5

Output: 30

Input: piles = [30,11,23,4,20], H = 6

Output: 23

Very similar to LC 1011 and LC 410 mentioned above. Let's design a feasible function, given an input speed, determine whether Koko can finish all bananas within H hours with hourly eating speed speed. Obviously, the lower bound of the search space is 1, and upper bound is max(piles), because Koko can only choose one pile of bananas to eat every hour.

def minEatingSpeed(piles: List[int], H: int) -> int:

def feasible(speed) -> bool:

# return sum(math.ceil(pile / speed) for pile in piles) <= H # slower

return sum((pile - 1) // speed + 1 for pile in piles) <= H # faster

left, right = 1, max(piles)

while left < right:

mid = left + (right - left) // 2

if feasible(mid):

right = mid

else:

left = mid + 1

return left

[1482. Minimum Number of Days to Make m Bouquets [Medium]](https://leetcode.com/problems/minimum-number-of-days-to-make-m-bouquets/)

Given an integer array bloomDay, an integer m and an integer k. We need to make m bouquets. To make a bouquet, you need to use k **adjacent flowers** from the garden. The garden consists of n flowers, the ith flower will bloom in the bloomDay[i] and then can be used in **exactly one** bouquet. Return *the minimum number of days* you need to wait to be able to make m bouquets from the garden. If it is impossible to make m bouquets return **-1**.

**Examples:**

Input: bloomDay = [1,10,3,10,2], m = 3, k = 1

Output: 3

Explanation: Let's see what happened in the first three days. x means flower bloomed and \_ means flower didn't bloom in the garden.

We need 3 bouquets each should contain 1 flower.

After day 1: [x, \_, \_, \_, \_] // we can only make one bouquet.

After day 2: [x, \_, \_, \_, x] // we can only make two bouquets.

After day 3: [x, \_, x, \_, x] // we can make 3 bouquets. The answer is 3.

Input: bloomDay = [1,10,3,10,2], m = 3, k = 2

Output: -1

Explanation: We need 3 bouquets each has 2 flowers, that means we need 6 flowers. We only have 5 flowers so it is impossible to get the needed bouquets and we return -1.

Now that we've solved three advanced problems above, this one should be pretty easy to do. The monotonicity of this problem is very clear: if we can make m bouquets after waiting for d days, then we can definitely finish that as well if we wait for more than d days.

def minDays(bloomDay: List[int], m: int, k: int) -> int:

def feasible(days) -> bool:

bonquets, flowers = 0, 0

for bloom in bloomDay:

if bloom > days:

flowers = 0

else:

bonquets += (flowers + 1) // k

flowers = (flowers + 1) % k

return bonquets >= m

if len(bloomDay) < m \* k:

return -1

left, right = 1, max(bloomDay)

while left < right:

mid = left + (right - left) // 2

if feasible(mid):

right = mid

else:

left = mid + 1

return left

[668. Kth Smallest Number in Multiplication Table [Hard]](https://leetcode.com/problems/kth-smallest-number-in-multiplication-table/description/)

Nearly everyone have used the [Multiplication Table](https://en.wikipedia.org/wiki/Multiplication_table). But could you find out the k-th smallest number quickly from the multiplication table? Given the height m and the length n of a m \* n Multiplication Table, and a positive integer k, you need to return the k-th smallest number in this table.

**Example :**

Input: m = 3, n = 3, k = 5

Output: 3

Explanation:

The Multiplication Table:

1 2 3

2 4 6

3 6 9

The 5-th smallest number is 3 (1, 2, 2, 3, 3).

For Kth-Smallest problems like this, what comes to our mind first is Heap. Usually we can maintain a Min-Heap and just pop the top of the Heap for k times. However, that doesn't work out in this problem. We don't have every single number in the entire Multiplication Table, instead, we only have the height and the length of the table. If we are to apply Heap method, we need to explicitly calculate these m \* n values and save them to a heap. The time complexity and space complexity of this process are both O(mn), which is quite inefficient. This is when binary search comes in. Remember we say that designing condition function is the most difficult part? In order to find the k-th smallest value in the table, we can design an enough function, given an input num, determine whether there're at least k values less than or equal to num. **The minimal num satisfying enough function is the answer we're looking for**. Recall that the key to binary search is discovering monotonicity. In this problem, if num satisfies enough, then of course any value larger than num can satisfy. This monotonicity is the fundament of our binary search algorithm.

Let's consider search space. Obviously the lower bound should be 1, and the upper bound should be the largest value in the Multiplication Table, which is m \* n, then we have search space [1, m \* n]. The overwhelming advantage of binary search solution to heap solution is that it doesn't need to explicitly calculate all numbers in that table, all it needs is just picking up one value out of the search space and apply enough function to this value, to determine should we keep the left half or the right half of the search space. In this way, binary search solution only requires constant space complexity, much better than heap solution.

Next let's consider how to implement enough function. It can be observed that every row in the Multiplication Table is just multiples of its index. For example, all numbers in 3rd row [3,6,9,12,15...] are multiples of 3. Therefore, we can just go row by row to count the total number of entries less than or equal to input num. Following is the complete solution.

def findKthNumber(m: int, n: int, k: int) -> int:

def enough(num) -> bool:

count = 0

for val in range(1, m + 1): # count row by row

add = min(num // val, n)

if add == 0: # early exit

break

count += add

return count >= k

left, right = 1, n \* m

while left < right:

mid = left + (right - left) // 2

if enough(mid):

right = mid

else:

left = mid + 1

return left

In LC 410 above, we have doubt "Is the result from binary search actually a subarray sum?". Here we have a similar doubt: "**Is the result from binary search actually in the Multiplication Table?**". The answer is yes, and we also can apply proof by contradiction. Denote num as the minimal input that satisfies enough function. Let's assume that num is not in the table, which means that num is not divisible by any val in [1, m], that is, num % val > 0. Therefore, changing the input from num to num - 1 doesn't have any effect on the expression add = min(num // val, n). So enough(num - 1) would also return True, same as enough(num). But we already know num is the minimal input satisfying enough function, so enough(num - 1) has to be False. Contradiction! The opposite of our original assumption is true: num is actually in the table.

[719. Find K-th Smallest Pair Distance [Hard]](https://leetcode.com/problems/find-k-th-smallest-pair-distance/)

Given an integer array, return the k-th smallest **distance** among all the pairs. The distance of a pair (A, B) is defined as the absolute difference between A and B.

**Example :**

Input:

nums = [1,3,1]

k = 1

Output: 0

Explanation:

Following are all the pairs. The 1st smallest distance pair is (1,1), and its distance is 0.

(1,3) -> 2

(1,1) -> 0

(3,1) -> 2

Very similar to LC 668 above, both are about finding Kth-Smallest. Just like LC 668, We can design an enough function, given an input distance, determine whether there're at least k pairs whose distances are less than or equal to distance. We can sort the input array and use two pointers (fast pointer and slow pointer, pointed at a pair) to scan it. Both pointers go from leftmost end. If the current pair pointed at has a distance less than or equal to distance, all pairs between these pointers are valid (since the array is already sorted), we move forward the fast pointer. Otherwise, we move forward the slow pointer. By the time both pointers reach the rightmost end, we finish our scan and see if total counts exceed k. Here is the implementation:

def enough(distance) -> bool: # two pointers

count, i, j = 0, 0, 0

while i < n or j < n:

while j < n and nums[j] - nums[i] <= distance: # move fast pointer

j += 1

count += j - i - 1 # count pairs

i += 1 # move slow pointer

return count >= k

Obviously, our search space should be [0, max(nums) - min(nums)]. Now we are ready to copy-paste our template:

def smallestDistancePair(nums: List[int], k: int) -> int:

nums.sort()

n = len(nums)

left, right = 0, nums[-1] - nums[0]

while left < right:

mid = left + (right - left) // 2

if enough(mid):

right = mid

else:

left = mid + 1

return left

[1201. Ugly Number III [Medium]](https://leetcode.com/problems/ugly-number-iii/)

Write a program to find the n-th ugly number. Ugly numbers are **positive integers** which are divisible by a **or** b **or** c.

**Example :**

Input: n = 3, a = 2, b = 3, c = 5

Output: 4

Explanation: The ugly numbers are 2, 3, 4, 5, 6, 8, 9, 10... The 3rd is 4.

Input: n = 4, a = 2, b = 3, c = 4

Output: 6

Explanation: The ugly numbers are 2, 3, 4, 6, 8, 9, 10, 12... The 4th is 6.

Nothing special. Still finding the Kth-Smallest. We need to design an enough function, given an input num, determine whether there are at least n ugly numbers less than or equal to num. Since a might be a multiple of b or c, or the other way round, we need the help of greatest common divisor to avoid counting duplicate numbers.

def nthUglyNumber(n: int, a: int, b: int, c: int) -> int:

def enough(num) -> bool:

total = mid//a + mid//b + mid//c - mid//ab - mid//ac - mid//bc + mid//abc

return total >= n

ab = a \* b // math.gcd(a, b)

ac = a \* c // math.gcd(a, c)

bc = b \* c // math.gcd(b, c)

abc = a \* bc // math.gcd(a, bc)

left, right = 1, 10 \*\* 10

while left < right:

mid = left + (right - left) // 2

if enough(mid):

right = mid

else:

left = mid + 1

return left

[1283. Find the Smallest Divisor Given a Threshold [Medium]](https://leetcode.com/problems/find-the-smallest-divisor-given-a-threshold/)

Given an array of integers nums and an integer threshold, we will choose a positive integer divisor and divide all the array by it and sum the result of the division. Find the **smallest** divisor such that the result mentioned above is less than or equal to threshold.

Each result of division is rounded to the nearest integer greater than or equal to that element. (For example: 7/3 = 3 and 10/2 = 5). It is guaranteed that there will be an answer.

**Example :**

Input: nums = [1,2,5,9], threshold = 6

Output: 5

Explanation: We can get a sum to 17 (1+2+5+9) if the divisor is 1.

If the divisor is 4 we can get a sum to 7 (1+1+2+3) and if the divisor is 5 the sum will be 5 (1+1+1+2).

After so many problems introduced above, this one should be a piece of cake. We don't even need to bother to design a condition function, because the problem has already told us explicitly what condition we need to satisfy.

def smallestDivisor(nums: List[int], threshold: int) -> int:

def condition(divisor) -> bool:

return sum((num - 1) // divisor + 1 for num in nums) <= threshold

left, right = 1, max(nums)

while left < right:

mid = left + (right - left) // 2

if condition(mid):

right = mid

else:

left = mid + 1

return left

End

Wow, thank you so much for making it to the end! Really appreciate that. As you can see from the python codes above, they all look very similar to each other. That's because I copy-pasted my own template all the time. No exception. This is the strong proof of my template's powerfulness and adaptability. I believe everyone can acquire this binary search template to solve many problems. All we need is just more practice to build up our ability to discover the monotonicity of the problem and to design a beautiful condition function.

Hope this helps.

**Reference**

* [[C++ / Fast / Very clear explanation / Clean Code] Solution with Greedy Algorithm and Binary Search](https://leetcode.com/problems/split-array-largest-sum/discuss/89819/C%2B%2B-Fast-Very-clear-explanation-Clean-Code-Solution-with-Greedy-Algorithm-and-Binary-Search)
* [Approach the problem using the "trial and error" algorithm](https://leetcode.com/problems/find-k-th-smallest-pair-distance/discuss/109082/Approach-the-problem-using-the-%22trial-and-error%22-algorithm)
* [Binary Search 101 The-Ultimate-Binary-Search-Handbook - LeetCode](https://leetcode.com/problems/binary-search/discuss/423162/Binary-Search-101-The-Ultimate-Binary-Search-Handbook)
* [ugly-number-iii Binary Search with picture & Binary Search Template - LeetCode](https://leetcode.com/problems/ugly-number-iii/discuss/387539/cpp-Binary-Search-with-picture-and-Binary-Search-Template)