Binary Search_2

@kbbhatt04

@June 23, 2023

Square Root

- Approach
    - Brute-force
        - Loop from 1 to n and check if `i * i ≤ n`
        - Time Complexity: $O(n)$
        - Space Complexity: $O(1)$
    - Optimal
        - Binary Search
        - Check if mid * mid == n
        - Else if mid * mid < n then l = mid + 1
        - Else r = mid - 1
        - return r
        - Time Complexity: $O(logn)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def floorSqrt(self, x):
        ans = 1
        for i in range(1, x):
            if i*i <= x:
                ans = i
            else:
```

```
                break
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def floorSqrt(self, x):
        l = 1
        r = x

        while l <= r:
            m = (l + r) // 2
            sqr = m * m
            if sqr == x:
                return m
            if sqr < x:
                l = m + 1
            else:
                r = m - 1
                # l is the first number whose square is >= n
                # We need to return r as it will be highest numb
        return r
```

```
// C++
// Optimal Solution
#define ll long long
class Solution{
  public:
    long long int floorSqrt(long long int x)
    {
        ll l = 1, r = x, mid, sqr;
        while (l <= r) {
            mid = l + (r - l) / 2;
            sqr = mid * mid;
```

```
            if (sqr == x) {return mid;}
            if (sqr < x) {l = mid + 1;}
            else {r = mid - 1;}
        }
                // l is the first number whose square is >= n
                // We need to return r as it will be highest nur
        return r;
    }
};
```

Find the Nth root of a number X

- Approach
  - Brute-force
    - Loop from 1 to x and run loop n times to calculate i^n
    - Time Complexity: $O(n * x)$
    - Space Complexity: $O(1)$
  - Better
    - Loop from 1 to x and use the pow method which is $O(logn)$ time complexity to calculate i^n
    - Time Complexity: $O(x * logn)$
    - Space Complexity: $O(1)$
  - Optimal
    - Binary Search
    - Check if pow(mid, n) == x
    - Else if pow(mid, n) < x then l = mid + 1
    - Else r = mid - 1
    - return -1

- Time Complexity: $O(logn * logx)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def NthRoot(self, n, m):
        for i in range(1, m+1):
            power = 1
            for j in range(n):
                power *= i
            if power == m:
                return i
            elif power > m:
                break
        return -1
```

```
# Python3
# Better Solution
class Solution:
    def NthRoot(self, n, m):
        for i in range(1, m+1):
            power = pow(i, n)
            if power == m:
                return i
            elif power > m:
                break
        return -1
```

```
# Python3
# Optimal Solution
class Solution:
    def NthRoot(self, n, m):
        l = 1
```

```python
        r = m

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

            power = pow(mid, n)
            if power == m:
                return mid
            elif power < m:
                l = mid + 1
            else:
                r = mid - 1
        return -1
```

```cpp
// C++
// Optimal Solution
class Solution{
    public:
    int NthRoot(int n, int m)
    {
        long long low=1,high=m;
        while(low<=high){
            long long mid=(low+high)/2;
            if(pow(mid,n)==m){
                return mid;
            }
            else if(pow(mid,n)>m){
                high=mid-1;
            }
            else{
                low=mid+1;
            }
        }
        return -1;
```

```
    }
};
```

Koko Eating Bananas

Return *the minimum integer* `k` *such that she can eat all the bananas within* `h` *hours*.

Example 1:

Input: piles = [3,6,7,11], h = 8
Output: 4

Example 2:

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

Example 3:

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

- Approach
  - Brute-force
    - Minimum time will be taken when koko eats `max(piles)` bananas each hour and total hours taken will be `len(piles)` hours and maximum time will be taken when koko eats 1 banana per hour
    - Thus for every number in range `1` to `max(piles)`, we calculate hours taken and if it is < h, then return it
    - Time Complexity: $O(n * max(piles))$
    - Space Complexity: $O(1)$
  - Optimal
    - Binary Search between 1 and max(piles)
    - Time Complexity: $O(n * log(max(piles)))$
    - Space Complexity: $O(1)$

```python
# Python3
# Brute-force Solution
from math import ceil
class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        for i in range(1, max(piles) + 1):
            hrs = 0
            for j in range(len(piles)):
                hrs += ceil(piles[j] / i)
            if hrs <= h:
                return i
        return -1
```

```python
# Python3
# Optimal Solution
from math import ceil
class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        l = 1
        r = max(piles)

        while l <= r:
            mid = (l + r) // 2
            s = 0
            for i in piles:
                s += ceil(i / mid)
            if s <= h:
                r = mid - 1
            else:
                l = mid + 1
        return l
```

```cpp
// C++
// Optimal Solution
```

```
#include <bits/stdc++.h>
class Solution {
public:
    int find_max(vector<int>& piles) {
        int maxi = piles[0];
        for (int i = 1; i < piles.size(); i++) {
            maxi = max(maxi, piles[i]);
        }
        return maxi;
    }

    int minEatingSpeed(vector<int>& piles, int h) {
        int l = 1, r = find_max(piles);
        while (l <= r) {
            int mid = l + (r - l) / 2;
            long long hrs = 0;
            for (int i = 0; i < piles.size(); i++) {
                hrs += ceil(double(piles[i]) / double(mid));
            }
            if (hrs <= h) {
                r = mid - 1;
            }
            else {
                l = mid + 1;
            }
        }
        return l;
    }
};
```

Minimum Number of Days to Make m Bouquets

You want 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 `i th` 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`.

- Approach
  - Brute-force
    - We know that minimum it could require `min(bloomDay)` days as if m = 1 and k = 1 then min(bloomDay) will be the answer and max it would take up to `max(bloomDay)` days as if m == len(bloomDay)
    - Thus we check for every number in the range `min(bloomDay)` and `max(bloomDay)`
    - If a number satisfies return that number
    - Time Complexity: $O(n * max(bloomDay))$
    - Space Complexity: $O(1)$
  - Optimal
    - Binary Search between `min(bloomDay)` and `max(bloomDay)`
    - Time Complexity: $O(n * log(max(bloomDay)))$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def minDays(self, bloomDay: List[int], m: int, k: int) -> in
        if len(bloomDay) < m * k:   return -1

        n = len(bloomDay)
        start = min(bloomDay)
        end = max(bloomDay)

        for i in range(start, end + 1):
            adj, bouq = 0, 0
```

```
            for flower in bloomDay:
                if flower <= i:
                    adj += 1
                else:
                    adj = 0
                if adj >= k:
                    bouq += 1
                    adj = 0
                    if bouq == m:
                        return i
        return -1
```

```
# Python3
# Optimal Solution
class Solution:
    def minDays(self, bloomDay: List[int], m: int, k: int) -> in
        if len(bloomDay) < m * k:   return -1

        n = len(bloomDay)
        start = min(bloomDay)
        end = max(bloomDay)

        while start <= end:
            mid = (start + end) // 2
            flow, bouq = 0, 0
            for i in range(n):
                if bloomDay[i] <= mid:
                    flow += 1
                else:
                    flow = 0
                if flow >= k:
                    flow = 0
                    bouq += 1
                    if bouq == m:   break
            if bouq >= m:
```

```
                end = mid - 1
            else:
                start = mid + 1
        return start
```

```
// C++
// Optimal Solution
class Solution {
public:
    int minDays(vector<int>& bloomDay, int m, int k) {
        if (bloomDay.size() < (long)m * (long)k) {return -1;}

        int n = bloomDay.size(), l = bloomDay[0], r = bloomDay[0
        for (int i = 0; i < n; i++) {
            if (bloomDay[i] < l) {l = bloomDay[i];}
            else if (bloomDay[i] > r) {r = bloomDay[i];}
        }

        while (l <= r) {
            int mid = l + (r - l) / 2;
            int adj = 0, bq = 0;
            for (auto flower: bloomDay) {
                if (flower <= mid) {
                    adj++;
                }
                else {
                    adj = 0;
                }
                if (adj == k) {
                    bq++;
                    adj = 0;
                    if (bq == m) {
                        break;
                    }
                }
```

```
                }
                if (bq >= m) {r = mid - 1;}
                else {l = mid + 1;}
            }
            return l;
        }
};
```

Find the Smallest Divisor Given a Threshold

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

Each result of the division is rounded to the nearest integer greater than or equal to that element. (For example: `7/3 = 3` and `10/2 = 5`).

- Approach
    - Brute-force
        - Answer lies between `1` and `max(nums)` as dividing by `1` will give max sum and dividing by `max(nums)` will give minimum sum
        - For every integer in range 1 to `max(nums)` check if the sum after division's result is ≤ threshold
        - Time Complexity: $O(n * max(nums))$
        - Space Complexity: $O(1)$
    - Optimal
        - Binary Search answer between `1` and `max(nums)`
        - Time Complexity: $O(n * log(max(nums)))$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
from math import ceil
class Solution:
    def smallestDivisor(self, nums: List[int], threshold: int) -
        rng = max(nums)
        for i in range(1, rng+1):
            if sum(ceil(j / i) for j in nums) <= threshold:
                return i
        return -1
```

```
# Python3
# Optimal Solution
from math import ceil
class Solution:
    def smallestDivisor(self, nums: List[int], threshold: int) -
        l = 1
        r = max(nums)

        while l <= r:
            mid = (l + r) // 2
            if sum(ceil(j / mid) for j in nums) <= threshold:
                r = mid - 1
            else:
                l = mid + 1
        return l
```

```
// C++
// Optimal Solution
class Solution {
public:
    int smallestDivisor(vector<int>& nums, int threshold) {
        int l = 1, r = nums[0];
        for (int i = 0; i < nums.size(); i++) {
            r = max(r, nums[i]);
        }
```

```
        while (l <= r) {
            int mid = l + (r - l) / 2;
            long long int sum = 0;
            for (int i = 0; i < nums.size(); i++) {
                                // typecast both ints to double
                sum += ceil((double)nums[i]/(double)mid);
            }
            if (sum <= threshold) {
                r = mid - 1;
            }
            else {
                l = mid + 1;
            }
        }
        return l;
    }
};
```

Capacity To Ship Packages Within D Days

A conveyor belt has packages that must be shipped from one port to another within `days` 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 `days` days.

- Approach
    - Brute-force
        - Minimum capacity should be `max(weights)` so as to accommodate largest package onto the ship
        - Maximum capacity would be `sum(weights)` so as to ship all packages in one day

        - Thus we find answer in the range `max(weights)` to `sum(weights)`
        - For every integer in the range we calculate days required and match with the given `days`
        - Time Complexity: $O(n * sum(weights))$
        - Space Complexity: $O(1)$
    - Optimal
        - Binary Search answer between the range `max(weights)` to `sum(weights)`
        - Time Complexity: $O(n * log(sum(weights)))$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> i
        for i in range(max(weights), sum(weights)+1):
            d = 1
            temp_weight = 0
            for j in weights:
                temp_weight += j
                if temp_weight > i:
                    temp_weight = j
                    d += 1
            if d <= days:
                return i
        return -1
```

```
# Python3
# Optimal Solution
class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> i
        l = max(weights)
        r = sum(weights)
```

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

            d = 1
            temp_weight = 0
            for j in weights:
                temp_weight += j
                if temp_weight > mid:
                    temp_weight = j
                    d += 1

            if d <= days:
                r = mid - 1
            else:
                l = mid + 1
        return l
```

```
// C++
// Optimal Solution
class Solution {
public:
    bool feasible(vector<int>& weights, int c, int days) {
        int daysNeeded = 1, currentLoad = 0;
        for (int weight : weights) {
            currentLoad += weight;
            if (currentLoad > c) {
                daysNeeded++;
                currentLoad = weight;
            }
        }
        return daysNeeded <= days;
    }

    int shipWithinDays(vector<int>& weights, int days) {
```

```
        int totalLoad = 0, maxLoad = 0;
        for (int weight : weights) {
            totalLoad += weight;
            maxLoad = max(maxLoad, weight);
        }

        int l = maxLoad, r = totalLoad;

        while (l <= r) {
            int mid = (l + r) / 2;
            if (feasible(weights, mid, days)) {
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return l;
    }
};
```

Kth Missing Positive Number

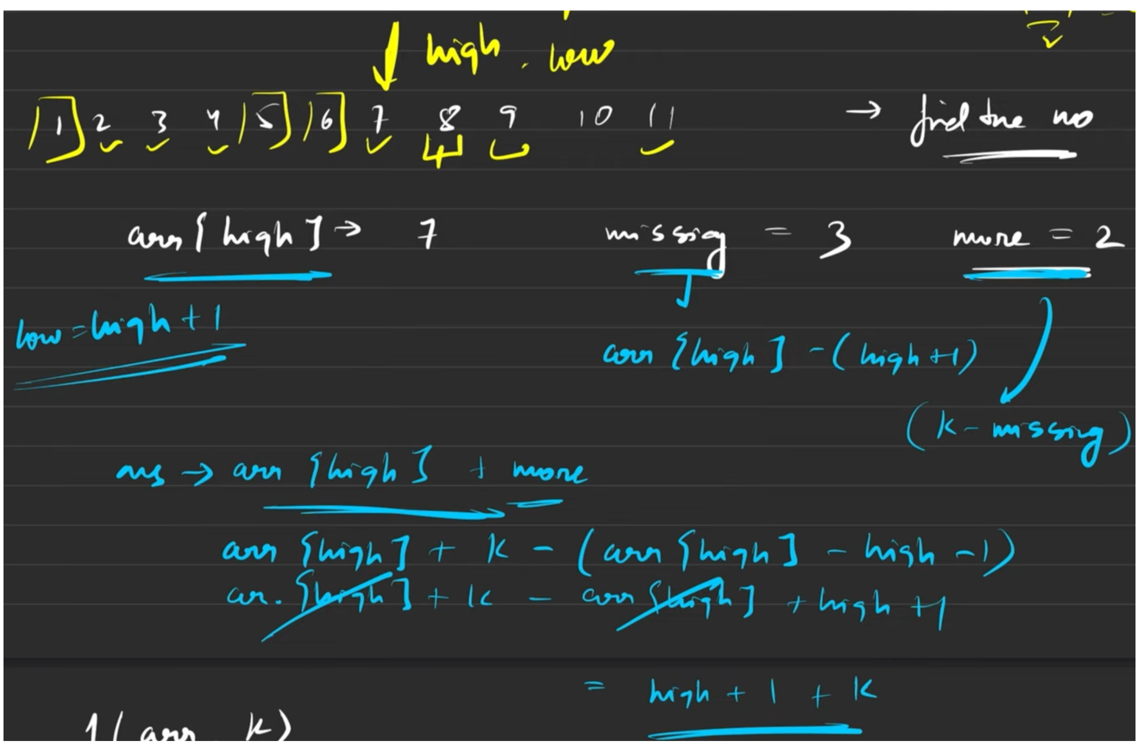
Input: arr = [2,3,4,7,11], k = 5
Output: 9
Explanation: The missing positive integers are [1,5,6,8,9,10,12,13,...]. The 5th missing positive integer is 9.

- Approach
  - Brute-force
    - Consider `arr = [5, 7, 9, 11]` and `k = 4`
    - Here `k < arr[0]` so we know that `kth` missing number is `k` itself
    - Consider `arr = [5, 7, 9, 11]` and `k = 6`
    - Ideally there should have been `1` at the `0th` index but this place is occupied by a `number ≤ 6` i.e. arr[0] = 5 so we need to increment k by 1

- Thus we iterate over array and increment k by 1 till `arr[i] ≤ k`, then return k
- Time Complexity: $O(n)$
- Space Complexity: $O(1)$

◦ Optimal

- Consider `arr = [2, 3, 4, 7, 11]` and `k = 5`
- Ideally arr should be `[1, 2, 3, 4, 5]` but as 7 is 3 places before, we know that there are 3 missing numbers before 7 and likewise 6 missing numbers before 11 because 11 - 5 = 6
- Missing numbers at an index = `arr[index] - (index + 1)`
- Binary Search the maximum index where missing numbers at the location is ≤ k
- Binary search will end when low > high so what we need to return is `high + k + 1`



- Time Complexity: $O(logn)$

- Space Complexity: $O(1)$

```python
# Python3
# Brute-force Solution
class Solution:
    def findKthPositive(self, a: List[int], k: int) -> int:
        for i in range(len(a)):
            if a[i] <= k:
                k += 1
            else:
                break
        return k
```

```python
# Python3
# Optimal Solution
class Solution:
    def findKthPositive(self, a: List[int], k: int) -> int:
        if k < a[0]:    return k
        l = 0
        r = len(a) - 1

        while l <= r:
            mid = (l + r) // 2
            missing = a[mid] - (mid + 1)

            if missing < k:
                l = mid + 1
            else:
                r = mid - 1

        return r + k + 1
```

```cpp
// C++
// Optimal Solution
```

```
class Solution {
public:
    int findKthPositive(vector<int>& arr, int k) {
        if (k < arr[0]) {return k;}
        int l = 0, r = arr.size() - 1;

        while (l <= r) {
            int mid = l + (r - l) / 2;
            int missing = arr[mid] - (mid + 1);

            if (missing < k) {
                l = mid + 1;
            }
            else {
                r = mid - 1;
            }
        }
        return r + k + 1;
    }
};
```

```
# Python3
# Given is increasing sequence not necessarily starting from 1
# Need to return -1 if all elements present
def KthMissingElement(arr, n, k):
    if n == arr[n-1] - arr[0] + 1:
        return -1
    l = 0
    r = n - 1

    while l <= r:
        mid = (l + r) // 2
        missing = arr[mid] - (mid + arr[0])
        if missing < k:
            l = mid + 1
```

```
        else:
            r = mid - 1
    if l >= n or r < 0:    return -1
    return r + k + arr[0]
```

Aggressive Cows

You are given an array consisting of n integers which denote the position of a stall. You are also given an integer k which denotes the number of aggressive cows. You are given the task of assigning stalls to k cows such that the minimum distance between any two of them is the maximum possible.

Input:
n=5
k=3
stalls = [1 2 4 8 9]
Output:
3
Explanation:
The first cow can be placed at stalls[0],
the second cow can be placed at stalls[2] and
the third cow can be placed at stalls[3].
The minimum distance between cows, in this case, is 3,
which also is the largest among all possible ways.

- Approach
    - Brute-force
        - We need to sort the array first
        - The maximum distance between two cows could be largest - smallest element in the array and the minimum distance could be 1
        - So we linear search for the answer between this range
        - To check if the cows can fit or not, iterate over the stalls array and place a cow in a stall[i] if `stall[i] - last_cow_stall ≥ dist` and check if all cows are placed or not

- Time Complexity: $O(nlogn + n * maxDistance)$
- Space Complexity: $O(1)$

◦ Optimal

- We need to sort the array first
- The maximum distance between two cows could be largest - smallest element in the array and the minimum distance could be 1
- So we binary search for the answer between this range
- To check if the cows can fit or not, iterate over the stalls array and place a cow in a stall[i] if `stall[i] - last_cow_stall ≥ dist` and check if all cows are placed or not
- Time Complexity: $O(nlogn + n * log(maxDistance))$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def solve(self,n,m,arr):
        def feasible(n, m, arr, mid):
            cows = 1
            last_cow = arr[0]
            for i in range(1, n):
                if arr[i] >= last_cow + mid:
                    cows += 1
                    last_cow = arr[i]
                    if cows == m:
                        return True
            return False

        arr.sort()
        l = 1
        r = arr[-1] - arr[0]
```

```
        for i in range(r, l - 1, -1):
            if feasible(n, m, arr, i):
                return i
        return -1
```

```
# Python3
# Optimal Solution
class Solution:
    def solve(self,n,m,arr):
        def feasible(n, m, arr, mid):
            cows = 1
            last_cow = arr[0]
            for i in range(1, n):
                if arr[i] >= last_cow + mid:
                    cows += 1
                    last_cow = arr[i]
                    if cows == m:
                        return True
            return False

        arr.sort()
        l = 1
        r = arr[-1] - arr[0]

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

            if feasible(n, m, arr, mid):
                l = mid + 1
            else:
                r = mid - 1
        return r
```

```
// C++
// Optimal Solution
bool isPossible(int a[], int n, int cows, int minDist) {
      int cntCows = 1;
      int lastPlacedCow = a[0];
      for (int i = 1; i < n; i++) {
        if (a[i] - lastPlacedCow >= minDist) {
          cntCows++;
          lastPlacedCow = a[i];
        }
      }
      if (cntCows >= cows) return true;
      return false;
    }

int main() {
      int n = 5, cows = 3;
      int a[]={1,2,8,4,9};
      sort(a, a + n);

      int low = 1, high = a[n - 1] - a[0];

      while (low <= high) {
        int mid = (low + high) >> 1;

        if (isPossible(a, n, cows, mid)) {
          low = mid + 1;
        } else {
          high = mid - 1;
        }
      }
      cout << "The largest minimum distance is " << high << endl
```

```
        return 0;
    }
```

Template

- Approach
    - Brute-force
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$
    - Better
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$
    - Optimal
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```