Editorial : Binary Search Intro

---

1) Binary Search - Iterative

```
def solve(n, arr, k):
    low = 0
    high = n - 1

    while (low <= high):
        mid = (low + (high - low) // 2)
        if (arr[mid] == k):
            return 1
        elif (arr[mid] > k):
            high = mid - 1
        else:
            low = mid + 1
    return -1

def inpt():
    n, k = map(int, input().strip().split())
    arr = list(map(int, input().strip().split()))
    print(solve(n, arr, k))

inpt()
```

Explanation

1. **Parameters**
    - `n` : The number of elements in the array.
    - `arr` : The sorted array of distinct numbers.
    - `k` : The target value to search for.
2. **Initialize `low` and `high`**

    ```
    low = 0
    high = n - 1
    ```

    - `low` starts at the beginning of the array (index 0).

- `high` starts at the end (index `n - 1`).

3. **Binary Search Loop**

```python
while (low <= high):
    mid = (low + (high - low) // 2)
    ...
```

   - We continue searching as long as `low <= high`.
   - We compute `mid` as the midpoint between `low` and `high`.

4. **Compare `arr[mid]` with `k`**

```python
if arr[mid] == k:
    return 1
elif arr[mid] > k:
    high = mid - 1
else:
    low = mid + 1
```

   - If the midpoint's value equals `k`, we return `1` (indicating the value is found).
   - If `arr[mid]` is greater than `k`, we move the `high` pointer to `mid - 1`.
   - Otherwise, we move the `low` pointer to `mid + 1`.

5. **Return -1 if Not Found**
   - If we exit the loop without finding `k`, we return `-1`.

6. **`inpt()` Function**

```python
def inpt():
    n, k = map(int, input().split())
    arr = list(map(int, input().split()))
    print(solve(n, arr, k))
```

   - Reads `n` and `k`, then reads the array `arr`.
   - Prints the result of `solve(n, arr, k)`.

---

2) Machines at Work

```python
def isPossible(n, m, arr, mid):
    sm = 0
    for i in range(n):
        sm += (mid // arr[i])
    return sm >= m

def solve(n, m, arr):
    low = 1
    high = max(arr) * m

    while (low <= high):
        mid = low + ((high - low) // 2)
        if (isPossible(n, m, arr, mid)):
            high = mid - 1
        else:
            low = mid + 1

    print(high + 1)

def inpt():
    n, m = map(int, input().split())
    arr = list(map(int, input().split()))
    solve(n, m, arr)

inpt()
```

Explanation

1. **Scenario**

   - You have `n` machines, each with a fixed time to produce 1 item (times given in `arr`).
   - You need to produce `m` total items as quickly as possible.
   - The question: **What is the minimum time** to produce `m` items if all machines can work in parallel?

2. `isPossible(...)`

   ```python
   def isPossible(n, m, arr, mid):
       sm = 0
       for i in range(n):
           sm += (mid // arr[i])
       return sm >= m
   ```

- Given a candidate time `mid`, we compute how many items can be produced by all machines in `mid` units of time.
- `mid // arr[i]` is how many items machine `i` can produce in `mid` time.
- We sum all production into `sm` and check if `sm >= m`.

3. **Binary Search for Minimum Time**

```
def solve(n, m, arr):
    low = 1
    high = max(arr) * m
    ...
```

- We set `low = 1` (the minimum time can't be 0) and `high = max(arr) * m` (worst case: the slowest machine makes all items).

4. **Check Mid**

```
mid = (low + high) // 2
if isPossible(n, m, arr, mid):
    high = mid - 1
else:
    low = mid + 1
```

- If `isPossible(...)` is `True`, it means we can produce `m` items within `mid` time, so we try a smaller time (`high = mid - 1`).
- Otherwise, we need more time (`low = mid + 1`).

5. **Final Answer**

```
print(high + 1)
```

- After the loop, `high + 1` is the smallest time in which production of `m` items is possible.

---

3) Square Root of an Integer

```
t = int(input())
for _ in range(t):
    n = int(input())
```

```python
    if n == 0:
        print(0)
        continue
    if n == 1:
        print(1)
        continue

    low, high = 1, n
    res = 0

    while low <= high:
        mid = low + (high - low) // 2
        mid_squared = mid * mid

        if mid_squared == n:
            res = mid
            break
        elif mid_squared < n:
            res = mid
            low = mid + 1
        else:
            high = mid - 1

    print(res)
```

Explanation

1. **Multiple Test Cases**
   - We read `t`, the number of test cases, then iterate.
2. **Base Cases**

   ```python
   if n == 0: print(0)
   if n == 1: print(1)
   ```

   - Quickly handle `n=0` and `n=1`.
3. **Binary Search for the Floor of the Square Root**

   ```python
   low, high = 1, n
   res = 0
   while low <= high:
       mid = (low + high) // 2
   ```

```
    mid_squared = mid * mid
    ...
```

- We check `mid_squared` compared to `n`.
- If `mid_squared == n`, we found the exact square root, store in `res` and break.
- If `mid_squared < n`, store `mid` in `res` (a potential floor) and move `low` up.
- If `mid_squared > n`, move `high` down.

4. **Print the Result**
   - `res` holds the floor of the square root (or the exact root if perfect square).

---

4) Wood Cutter

```python
def collectedWood(n, m, arr, mid):
    sm = 0
    for i in range(n):
        if (arr[i] > mid):
            sm += (arr[i] - mid)
    return sm >= m

def solve(n, m, arr):
    low = min(arr)
    high = max(arr)
    while (low <= high):
        mid = low + (high - low) // 2
        if (collectedWood(n, m, arr, mid)):
            low = mid + 1
        else:
            high = mid - 1
    return low - 1

def inpt():
    n, m = map(int, input().split())
    arr = list(map(int, input().split()))
    print(solve(n, m, arr))

inpt()
```

Explanation

1. **Scenario**

- We have `n` trees, each with a height given in `arr`.
- We want at least `m` units of wood by cutting the trees. We can set a machine height `mid`; any part of a tree above `mid` is cut off and collected.

2. **`collectedWood(...)`**

```
def collectedWood(n, m, arr, mid):
    sm = 0
    for i in range(n):
        if arr[i] > mid:
            sm += (arr[i] - mid)
    return sm >= m
```

- For each tree taller than `mid`, we collect `(arr[i] - mid)` wood.
- We check if the total collected `sm` is at least `m`.

3. **Binary Search**

```
low = min(arr)
high = max(arr)
while (low <= high):
    mid = ...
    if collectedWood(..., mid):
        low = mid + 1
    else:
        high = mid - 1
return low - 1
```

- We search for the **maximum** `mid` that still allows collecting at least `m` wood.
- If `collectedWood` is True, it means we can try a taller `mid` (cut less wood, so we go `low = mid + 1`).
- If False, we need a smaller `mid` (`high = mid - 1`).

4. **Final Result**

- After the loop, `low - 1` is the highest possible cut height that yields at least `m` wood.

---

5) Restaurants during pandemic

```
def isPossible(n, c, arr, mid):
    person = 1
```

```python
    curr = arr[0]
    for i in range(1, n):
        if (arr[i] - curr) >= mid:
            person += 1
            curr = arr[i]
            if (person >= c):
                break
    return (person >= c)

def solve(n, c, arr):
    low = 0
    high = arr[n - 1] - arr[0]

    while (low <= high):
        mid = low + ((high - low) // 2)
        if (isPossible(n, c, arr, mid)):
            low = mid + 1
        else:
            high = mid - 1
    return low - 1

def inpt():
    t = int(input())
    for _ in range(t):
        n, c = map(int, input().split())
        arr = list(map(int, input().split()))
        arr.sort()
        print(solve(n, c, arr))

inpt()
```

Explanation

1. **Scenario**
    - We have `n` seats (positions in `arr`) and `c` customers.
    - We want to **maximize the minimum distance** between any two customers.
2. **`isPossible(...)`**

    ```python
    def isPossible(n, c, arr, mid):
        person = 1
        curr = arr[0]
        for i in range(1, n):
            if (arr[i] - curr) >= mid:
                person += 1
                curr = arr[i]
    ```

```python
            if (person >= c):
                break
    return (person >= c)
```

- We place the first customer at `arr[0]`.
- We then try to place additional customers such that each is at least `mid` units away from the last placed customer.
- If we can place all `c` customers, return True.

3. **Binary Search for Maximum Minimum Distance**

```python
def solve(n, c, arr):
    low = 0
    high = arr[n - 1] - arr[0]
    ...
```

- `low` starts at 0, `high` at the maximum possible distance (between the first and last seat).
- For a guess `mid`, we check if it's possible to seat `c` people with at least `mid` distance.

4. **Update Range**

- If `isPossible` is True, we try a bigger distance (`low = mid + 1`).
- Otherwise, we reduce the distance (`high = mid - 1`).

5. **Return `low - 1`**

- The largest minimum distance is `low - 1` after the loop finishes.

---

6) Average Chocolates

```python
def solve(n, k, arr, avg):
    if (k <= avg[0]):
        return 0
    elif (k > avg[n - 1]):
        return n

    low = 0
    high = n - 1
    while (low <= high):
        mid = low + ((high - low) // 2)

        if (avg[mid] < k):
```

```python
            low = mid + 1
        else:
            high = mid - 1

    return low

def inpt():
    n = int(input())
    arr = list(map(int, input().split()))
    q = int(input())
    arr.sort()
    sm = 0
    avg = [-1] * n

    for i in range(n):
        sm += arr[i]
        avg[i] = (sm / (i + 1))

    for _ in range(q):
        k = int(input())
        print(solve(n, k, arr, avg))

inpt()
```

Explanation

1. **Problem Context**

    - We have `n` friends, each with a certain number of chocolates (`arr`), sorted in ascending order.
    - We compute a prefix average array `avg[i]` = average of the first `i+1` elements in the sorted list.
    - We have `q` queries, each query has a number `k`, and we want to find how many prefix averages are `< k` (or something similar based on the code's logic).

2. **Compute Prefix Averages**

    ```python
    sm = 0
    avg = [-1] * n
    for i in range(n):
        sm += arr[i]
        avg[i] = sm / (i + 1)
    ```

    - `avg[i]` stores the average of the first `i+1` sorted chocolates.

3. **`solve(n, k, arr, avg)`**

```
if k <= avg[0]: return 0
elif k > avg[n - 1]: return n
```

- If `k` is less than or equal to the smallest prefix average, the answer is `0`.
- If `k` is greater than the largest prefix average, the answer is `n`.

4. **Binary Search**

```
while (low <= high):
    mid = ...
    if avg[mid] < k:
        low = mid + 1
    else:
        high = mid - 1
return low
```

- We find the first position where `avg[mid] >= k`.
- `low` ends up being the index of that position.

5. **Answer**

- We print `low` for each query, presumably meaning the count of prefix averages that are `< k`.

---

7) Coding Practice Time

```
def can_complete_with_time(problems, n, m, T):
    days_needed = 1
    current_time = 0

    for time in problems:
        if time > T:
            return False
        if current_time + time > T:
            days_needed += 1
            current_time = time
            if days_needed > m:
                return False
        else:
            current_time += time

    return days_needed <= m
```

```python
def minimum_training_time(n, m, problems):
    left, right = 0, sum(problems)
    result = right

    while left <= right:
        mid = (left + right) // 2
        if can_complete_with_time(problems, n, m, mid):
            result = mid
            right = mid - 1
        else:
            left = mid + 1

    print(result)
```

Explanation

1. **Parameters**

   - `n`: Number of problems.
   - `m`: Number of days.
   - `problems`: A list of times required to solve each problem.

2. **`can_complete_with_time(...)`**

   ```python
   def can_complete_with_time(problems, n, m, T):
       days_needed = 1
       current_time = 0
       for time in problems:
           if time > T:
               return False
           if current_time + time > T:
               days_needed += 1
               current_time = time
               if days_needed > m:
                   return False
           else:
               current_time += time
       return days_needed <= m
   ```

   - We try to solve all problems in **at most `m` days** if each day has a limit of `T` time.
   - If a single problem `time` exceeds `T`, it's impossible.
   - We keep adding problem times to `current_time` until we exceed `T`, then increment `days_needed`.

3. **Binary Search for Minimum `T`**

```python
left, right = 0, sum(problems)
result = right
while left <= right:
    mid = (left + right) // 2
    if can_complete_with_time(..., mid):
        result = mid
        right = mid - 1
    else:
        left = mid + 1
print(result)
```

- We guess a daily limit `mid`.
- If we can solve all problems in `m` days or fewer with daily limit `mid`, we try a smaller limit.
- Otherwise, we need a bigger `mid`.

---

8) Everything Related to Binary Search (First Occurrence, Last Occurrence, Count)

```python
def binary_search_first(arr, key):
    left, right = 0, len(arr) - 1
    first_occurrence = -1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == key:
            first_occurrence = mid
            right = mid - 1
        elif arr[mid] < key:
            left = mid + 1
        else:
            right = mid - 1
    return first_occurrence

def binary_search_last(arr, key):
    left, right = 0, len(arr) - 1
    last_occurrence = -1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == key:
            last_occurrence = mid
            left = mid + 1
        elif arr[mid] < key:
            left = mid + 1
        else:
```

```python
            right = mid - 1
    return last_occurrence

def find_first_last_count(arr, key):
    first = binary_search_first(arr, key)
    last = binary_search_last(arr, key)
    if first == -1 or last == -1:
        return "-1 -1 0"
    else:
        count = last - first + 1
        print(f"{first} {last} {count}")
```

Explanation

1. **`binary_search_first`**

   ```python
   if arr[mid] == key:
       first_occurrence = mid
       right = mid - 1
   ```

   - Once we find `key`, we keep moving `right` left to find an earlier occurrence.

2. **`binary_search_last`**

   ```python
   if arr[mid] == key:
       last_occurrence = mid
       left = mid + 1
   ```

   - Once we find `key`, we keep moving `left` right to find a later occurrence.

3. **`find_first_last_count`**

   - We call both searches.
   - If either returns `-1`, the key does not exist in `arr`.
   - Otherwise, we calculate the number of occurrences as `last - first + 1` and print them.
