CS218, Spring 2025 **Due: 11:59pm, Friday, 4/11**
Assignment #1 [1]
Name: Lakhan Kumar Sunilkumar **Student ID: 862481700**

---

1 A Complex Complexity Problem (1.6pts)

1. $(\sqrt{3})^{\log n} = \_\_\_(n)$
Answer: o
Explanation: Rewrite $\left(\sqrt{3}\right)^{\log n}$ using exponent rules:

$$\left(3^{1/2}\right)^{\log n} = 3^{\frac{\log n}{2}} = n^{\frac{\log 3}{2}} \approx n^{0.5493}$$

After re-writting it as $n^{0.5493}$, we can see that it grows polynomially but can also say that it grows *strictly slower* than n, hence we can conclude the following that $\left(\sqrt{3}\right)^{\log n} = o(n)$.

2. $\log\log n = \_\_\_(\sqrt{\log n})$
Answer: o
Explanation: Let's take two values of n to observe how $\log\log n$ compares to $\sqrt{\log n}$:

n	$\log_2 n$	$\log_2(\log_2 n)$	$\sqrt{\log_2 n}$
$2^{16} = 65{,}536$	16	4	4
$2^{64} \approx 1.84 \times 10^{19}$	64	6	8
$2^{128} \approx 3.40 \times 10^{38}$	128	7.00	11.31

At $n = 2^{16}$, given both the functions are equal. However, at $n = 2^{64}$, $\log\log n$ is clearly smaller than $\sqrt{\log n}$. As we can see, n increases, $\log\log n$ grows very slowly while $\sqrt{\log n}$ increases more quickly. The gap continues to widen. This supports the conclusion:

$$\therefore \log\log n = o(\sqrt{\log n})$$

3. $\log(n!) = \_\_\_(n\log n)$
Answer: Θ
Explanation:
To see how quickly the equation $\log(n!)$ grows, we can use Stirling's approximation as mentioned in the CS-218 slides 03-lower-bounds, the estimations for the given factorials for as large as n are given as:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Let's take the logarithm of both sides:

$$\log(n!) \approx \log\left(\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n\right)$$

Using log rules, we split the expression:

$$= \log(\sqrt{2\pi n}) + \log\left(\left(\frac{n}{e}\right)^n\right) = \frac{1}{2}\log(2\pi n) + n\log n - n$$

[1] Some of the problems are adapted from existing problems from online sources. Thanks to the original authors.

Simplied as:

$$\log(n!) \approx n \log n - n + \frac{1}{2}\log(2\pi n)$$

By considering these equations, the dominant term is $n \log n$. The $-n$ and $\frac{1}{2}\log(2\pi n)$ will be growing more slowly and are considered to be the lower-order terms.

Therefore, asymptotically:

$$\log(n!) = \Theta(n \log n)$$

Reference: Wikipedia – Stirling's Approximation

4. $2^n = \_\_(3^n)$
   Answer: o
   Explanation:
   For this equation, we're comparing two exponential functions: 2^n and 3^n. Mathematically, even both grow very fast, the base of the exponent makes a huge difference in comparison. Since $3 > 2$, 3^n grows significantly faster than 2^n as n increases.

   Let's look at a few values:

n	2^n	3^n
5	32	243
10	1,024	59,049
19	524,288	1,162,261,467

   As n increases, the gap between 2^n and 3^n keeps widening. To analyze, we can consider taking it's ratio:

$$\frac{2^n}{3^n} = \left(\frac{2}{3}\right)^n \to 0 \quad \text{as } n \to \infty$$

   This clearly shows that it's the little o; if the ratio of $f(n)$ to $g(n)$ tends to 0, then $f(n) = o(g(n))$. According to the comparison chart, this is just like saying $a < b$ in real numbers.

   So, 2^n grows strictly slower than 3^n, and we can say that:

$$2^n = o(3^n)$$

2 Solve Recurrences (0.9pts)

For all of them, you can assume the base case is when n is a constant, $T(n)$ is also a constant. Use $\Theta(\cdot)$ to present your answer.

Reference: Master Theorem Cases

1. $T(n) = T(n/2) + n \log n$
   Answer: $\Theta(n \log n)$
   Explanation:
   We use the Master Theorem to solve this recurrence. The general form of the Master Theorem is:

$$T(n) = a \cdot T(n/b) + f(n)$$

Let's consider:

$$a = 1, \quad b = 2, \quad f(n) = n \log n$$

Now let's calculate:

$$\log_b a = \log_2 1 = 0$$

Let's compare $f(n)$ with $n^{\log_b a} = n^0 = 1$.

Clearly:

$$f(n) = n \log n = \omega(n^{0+\epsilon}) \quad \text{for any } \epsilon < 1$$

So the work done in the recursive call $f(n/2)$ is found to be consistently smaller than a constant fraction of the total work $f(n)$ as n growing large.

$$f(n/2) = \frac{n}{2}\log(n/2) = \frac{n}{2}(\log n - 1) = \frac{n \log n}{2} - \frac{n}{2}$$

So,

$$a \cdot f(n/2) < f(n) \quad \text{for large } n$$

Hence, the regularity condition is satisfied.

Using Master Theorem (Case 3):

$$T(n) = \Theta(n \log n)$$

2. $T(n) = 2T(n/4) + \sqrt{n}$

   Answer: $\Theta(\sqrt{n} \log n)$

   Explanation:
   We apply the Master Theorem to solve this recurrence. The general form is:

$$T(n) = a \cdot T(n/b) + f(n)$$

   In this case:

$$a = 2, \quad b = 4, \quad f(n) = \sqrt{n} = n^{1/2}$$

   Now compute:

$$\log_b a = \log_4 2 = \frac{\log 2}{\log 4} = \frac{1}{2}$$

   So we can compare $f(n) = n^{1/2}$ to $n^{\log_b a} = n^{1/2}$.
   Which means:

$$f(n) = \Theta(n^{\log_b a})$$

   From **Case 2** of the Master Theorem:

   If $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ then for some $k = 0$, it would be:

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n) = \Theta(\sqrt{n} \log n)$$

$$T(n) = \Theta(\sqrt{n} \log n)$$

3. $T(n) = 4T(n/4) + n^{1/2}$

   Answer: $\Theta(n)$

   Explanation:
   We can solve this using the Master Theorem. The given standard form is:

   $$T(n) = a \cdot T(n/b) + f(n)$$

   In our case:
   $$a = 4, \quad b = 4, \quad f(n) = \sqrt{n} = n^{1/2}$$

   Now substitute it as:
   $$\log_b a = \log_4 4 = 1 \quad \Rightarrow \quad n^{\log_b a} = n^1 = n$$

   We calculate $f(n) = n^{1/2}$ with $n^{\log_b a} = n$.

   $$f(n) = o(n) \Rightarrow f(n) = o(n^{\log_b a})$$

   From **Case 1** of the Master Theorem,

   If $f(n) = o(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then:

   $$T(n) = \Theta(n^{\log_b a}) = \Theta(n)$$

   $$T(n) = \Theta(n)$$

3 Test the candies (2.5 pts + 1 bonus pt)

1. (0.2 pts)

Algorithm 1: Identify the Defective Candy via Divide and Conquer Algorithm

```
Function FindDefectiveCandy(candies):
    // Possible corner case
    if number of candies == 1 then
        return the only candy found in the list ;
    end
    spareCandy ← null;
    if number of candies is odd then
        spareCandy ← remove one candy from the list ;
    end
    // Divide the candies into two nearly equal groups
    total ← number of candies that is left;
    half ← total / 2;
    groupLeft ← first half of candies;
    groupRight ← second half of candies;
    // Use balance scale and now start the comparison of l and r
    weighingResult ← compareWeight(groupLeft, groupRight) ;
    // Make decision based on the result from the scale
    if weighingResult == "is-equal" then
        return spareCandy;
    end
    else if weighingResult == "is-left-lighter" then
        return FindDefectiveCandy(groupLeft);
    end
    else if weighingResult == "is-right-lighter" then
        return FindDefectiveCandy(groupRight);
    end
```

The algorithm checks if only one candy is left if so, then that's the bad one, and then we return it. If by chance the total is odd, one candy will be kept aside to balance it later, such that we can split the rest evenly. This reserved candy is only used in the scale to balance it in the later scenarios.

The remaining candies are divided into two equal groups, and the weight is checked. If the scale is evenly balanced, we should set aside the bad candy. If not, then the lighter group contains the defective candy. The same logic is also applied to the lighter group until one candy remains.

Since the search space is halved each time, the algorithm runs in $\theta(\log_2 n)$ time.

2. (0.2 pts)

Algorithm 2: Check the Lighter Candy Using Ternary Divide and Conquer Algorithm

```
Function FindDefectiveCandy(candies):
    if size(candies) == 1 then
        return the only candy in the list;
    end
    total ← size(candies);
    three-div ← floor(total / 3);
    groupA ← first third of candies;
    groupB ← second third of candies;
    groupC ← remaining candies;
    result ← checkWeight(groupA, groupB);
    if result == "is-balanced" then
        return FindDefectiveCandy(groupC);
    end
    else if result == "is-left-lighter" then
        return FindDefectiveCandy(groupA);
    end
    else
        return FindDefectiveCandy(groupB);
    end
```

So The plan is to divide the candies into three nearly equal parts. Each weight reduces the problem size by roughly a factor of three, the number of steps—and therefore the cost—is exactly $\lceil \log_3 n \rceil$ dollars. This is the concept used in the ternary search.

3. (0.7pts) **Explanation:**

To efficiently find the bad candy among n candies by using a balance scale, we can consider that any algorithm must make at least $\lceil \log_3 n \rceil$ weighings. This represents the minimum effort or the lower-bound required to solve the problem in an optimal approach.

The balance scale can provide one of these expected outcomes:

- Maybe both sides are equal, or
- Given the left side is lighter, or
- Given the right side is lighter.

So, with k weighings, the algorithm could maybe have at most 3^k distinct exhaustive outcome paths (could be a possibility). Since the goal is to identify one bad candy among n total distinct ones.

$$3^k \geq n$$

Taking the logarithm of base 3:

$$k \geq \log_3 n$$

Can round up as the weighings must be only whole numbers:

$$k \geq \lceil \log_3 n \rceil$$

Therefore, the best strategy can only reliably detect the defective candy in fewer than $\lceil \log_3 n \rceil$ weighings. If used any other method, it would probably not cover all possibilities and corner cases. This shows that $\lceil \log_3 n \rceil$ is the most feasible lower bound that we can come up with

4. (0.3pts)

Algorithm 3: Detect Two Fake Candies with Divide-and-Weigh Strategy

```
Function CheckedTwoFakes(bag):
    if size(bag) ≤ threshold then
        return CheckSmallGroup(bag);
    end
    leftGroup ← first $\lfloor n/2 \rfloor$ candies;
    rightGroup ← remaining candies;
    outcome ← compareWeight(leftGroup, rightGroup);
    if outcome == "is-balanced" then
        return { CheckedOneFake(leftGroup), CheckedOneFake(rightGroup) };
    end
    else if outcome == "is-left-lighter" then
        return CheckedTwoFakes(leftGroup);
    end
    else
        return CheckedTwoFakes(rightGroup);
    end

// Possibility of finding a single fake despite two candies
Function CheckedOneFake(candies):
    if size(candies) == 1 then
        return candies[0];
    end
    if size(candies) is odd then
        spare ← removeOne(candies);
    end
    else
        spare ← null;
    end
    A, B ← splitEqually(candies);
    result ← compareWeight(A, B);
    if result == "is-balanced" then
        return spare;
    end
    else if result == "is-left-lighter" then
        return CheckedOneFake(A);
    end
    else
        return CheckedOneFake(B);
    end

Function CheckSmallGroup(smallBag):
    return two lightest candies found through basic comparisons;
```

The first step in this algorithm is that it will split up the candies into two equal parts and weigh them. If the scale balances, we can assume that each half contains exactly one fake. In that case, the method function FindOneFakes is selected and is made to run independently on both halves to find the single fake. The helper function for a single fake works like a binary search, that is, it splits the group (adjusting for odd size if needed) and uses the weighing result to narrow down the search until one remains.

The second scenario is that if the scale tips, the lighter side might contain both fake candies, as we know that a split would have balanced. The algorithm performs recursion on that segment. Since each step cuts the problem roughly in half, the total number of weighings is $O(\log n)$ in all cases. Thus, even with two bad candies, we can identify both efficiently using a smart divide-and-conquer approach guided by the balance scale.

5. (0.3pts)

Algorithm 4: Find k Fake Candies Using Ternary Partitioning

```
Function FindGivenKFakes(candies, k):
    if size(candies) $\le$ threshold then
        return SolveSmallGroup(candies, k);
    end
    third $\leftarrow$ floor(size(candies) / 3);
    groupA $\leftarrow$ first third of candies;
    groupB $\leftarrow$ second third of candies;
    groupC $\leftarrow$ remaining candies;
    outcome $\leftarrow$ compareWeight(groupA, groupB);
    if outcome == "is-balanced" then
        $r \leftarrow s \leftarrow$ CountBalanced(groupA, groupB, k);
        $t \leftarrow k - 2r$;
    end
    else if outcome == "is-groupA-lighter" then
        $(r, s) \leftarrow$ CountWithDifference(groupA, groupB, +1, k); $t \leftarrow k - (r + s)$;
    end
    else
        $(s, r) \leftarrow$ CountWithDifference(groupB, groupA, +1, k); $t \leftarrow k - (r + s)$;
    end
    fakesA $\leftarrow$ FindGivenKFakes(groupA, r);
    fakesB $\leftarrow$ FindGivenKFakes(groupB, s);
    fakesC $\leftarrow$ FindGivenKFakes(groupC, t);
    return merge(fakesA, fakesB, fakesC);
```

Explanation:
The algorithm splits the given candies into three parts and puts the first two on the balance scale and compares them. If the given scale is shown as balanced, it means that both the groups have the same number of fakes (let's call it r), so the third group should actually be having $t = k - 2r$. If it shows unbalanced, then the lighter group could have one more fake. For example, if Group-A is lighter, then $r = s + 1$, and $t = k - (r + s)$.

Since k is a constant, we can easily determine r and s as extra weighings per level as it will only take $O(1)$.

Each recursive call in this algorithm is made to handle a section of subproblems of size about $n/3$, and its level is $O(\log n)$. At each level, the total weighing cost is set to remain constant. Thus, the given complete algorithm will run in $O(\log n)$ time.

In short, by spreading k fakes across smaller groups at each step, we can identify all k bad candies efficiently using only $O(\log n)$ weighings.

6. (0.8pts)

Explanation: We are given n candies, and we know that there is exactly one of them that could be defective. However, unlike previous cases that have been covered so far, we clearly don't know whether the bad candy could be lighter or heavier than the rest. This shows that for each of the n candies, there could be two distinct faulty possibilities; it could be either lighter or heavier. Therefore, we can estimate that there are a total of $2n$ possible defective cases that must be distinguished and segregated.

Each weighing on a balance scale yields one of three possible outcomes: the left pan might be heavier, the right pan is heavier, or both pans are balanced. So, with k weighings, we can estimate that the number of distinct outcome sequences is at most 3^k.

To efficiently determine which candy is bad and whether it is heavier or lighter, the algorithm must be able to look through all $2n$ possible cases. This requires the decision tree formed by the weighings to have at least $2n$ leaves. Therefore, we should be having:

$$3^k \geq 2n$$

Taking logarithms base 3 on both sides:

$$k \geq \log_3(2n)$$

Since k must only be an integer (as each weighing costs 1 dollar), the number of weighings must satisfy:

$$k \geq \lceil \log_3(2n) \rceil$$

Takeaway: This shows that any algorithm designed to identify the defective candy and determine whether it is heavier or lighter must use at least $\lceil \log_3(2n) \rceil$ dollars. No fewer weighings would suffice to uniquely resolve all $2n$ scenarios, confidently making this the lower bound.

7. (bonus, 1pt)

Explanation: *A Three-Weighing Strategy for 13 Candies (with Unknown Weight Deviation)*

With the 13 candies that we are considering, we should start labelling them from 1 through 13, exactly one of which is found to be bad is what we know (either heavier or lighter), and only a reference candy R known to be good as per the question. Based on the lower bound $\lceil \log_3(2 \cdot 13) \rceil = 3$, we know that three weighings (i.e., three dollars) should be enough to find the defective candy and it can determine whether it is heavier or lighter.

Weighing 1

With the given 13 candies, we should place the candies 1, 2, 3, 4 on the left pan and 5, 6, 7, 8 on the right pan.

Case 1A (Balanced): All of the candies 1–8 are known to be now genuine (a simple possibility). The bad candy must probably be one of $\{9, 10, 11, 12, 13\}$.

Case 1B (Unbalanced): We assume that the left pan is lighter (the other case is symmetric). Then, either one of $\{1, 2, 3, 4\}$ is light, or one of $\{5, 6, 7, 8\}$ is heavy.

Weighing 2

This setup now entirely will depends on the outcome of the first weighing.

If 1A (Balanced):

Weighing 2A: We now compare the following:

- Left pan: 1, 2, 9, 10
- Right pan: 3, 4, 11, R

Here, coins 1–4 and R are known to be good. So:

- **If balanced:** The bad candy could be found either at 12 or 13 (just making an assumption).

- **If unbalanced:** The imbalance tells us whether the defect is heavier or lighter, and the bad coin must be among 9, 10, or 11.

If 1B (Unbalanced):

Weighing 2B: We will swap some of the coins and bring them into the reference:

- Left pan: 1, 5, 6, 9
- Right pan: 2, 7, 10, R

This setup will help us to keep away the defective candy to a smaller group while determining whether it is heavy or light based on the outcome.

Weighing 3

By this point, the suspect set has been reduced to maybe two or three coins, and we know whether the bad one is of heavier or of lighter weight.

If 1A:

- **If we think the suspects are 12 and 13:** Compare 12 vs R.
  - If balanced: 13 is defective.
  - If unbalanced: 12 is defective, and the direction indicates if it's light or heavy.
- **If suspects are 9, 10, 11 with known direction:** Compare 9 vs R.
  - If balanced: check and reduce it maybe between 10 and 11 based on prior weighing.
  - If unbalanced: 9 is known to be defective (should be consistent with the known deviation).

If 1B: We use the outcome of Weighing 2B to choose one suspect and then we will compare it with R to identify the fake and its weight deviation.

Explanation

There are $13 \times 2 = 26$ possible outcomes (each candy could be heavier or lighter). Three weighings yield $3^3 = 27$ distinct result sequences so it's just enough to distinguish all cases.

Each weighing uses already cleared up candies and the known good candy R to highlight and shown the comparison, helping to reduce the suspects and determine the nature of the deviation.

Takeaway

This strategy is known to solve the problem using only exactly 3 weighings:

(a) **Weighing 1:** We divide the candies into two groups of 4. A balance clears 8 candies otherwise, the bad candy is among the lighter or heavier side.

(b) **Weighing 2:** We mix the cleared coins with new suspects and compare against the reference. This will reveals a smaller suspect set and whether the bad candy is light or heavy.

(c) **Weighing 3:** Then compare a final suspect against R to correctly identify the fake and its weight deviation.

This demonstrates a valid and efficient 3-dollar solution, matching the theoretical lower bound of $\lceil \log_3(2 \cdot 13) \rceil = 3$.

Bonus Problems

4 (Programming) Finding the Minimum Value (1pt)

References

- Prof. Yan Gu, Slides 04 – Divide and Conquer
- Ternary Search Algorithm — `https://www.youtube.com/watch?v=o3HPRpbGlbI`
- Ternary vs Binary Search — `https://www.youtube.com/watch?v=8AE8aZ7kEkk&t=40s`

Code Submission ID: 314801368, Username: lsuni001

Algorithm 5: Recursive Ternary Search to Minimize a Function

```
Function TernarySearch($f$, $l$, $r$, $e = 10^{-8}$):
    if $r - l < e$ then
        return $f\left(\frac{l+r}{2}\right)$;
    end
    $m_1 \leftarrow l + \frac{r-l}{3}$;
    $m_2 \leftarrow r - \frac{r-l}{3}$;
    if $f(m_1) < f(m_2)$ then
        return TernarySearch($f$, $l$, $m_2$, $e$);
    end
    else
        return TernarySearch($f$, $m_1$, $r$, $e$);
    end
Function Main():
    Read a real number $a$;
    Define $f(x) = \frac{x^2}{\log x} + a \cdot x$;
    $minVal \leftarrow$ TernarySearch($f$, 1.0000001, 1.9999999);
    Print $minVal$ rounded to 9 decimal places;
```

Explanation of the Algorithm

As we know that the function we are dealing with is a unimodal function that is it just has one dip in the range, I found ternary search to be more suitable to solve this problem rather than binary search. It is more suitable than binary search for optimization problems where the function increases and then decreases (or vice versa), which is typical in continuous domains. So instead of dividing the interval list into two parts, it then splits it into three, and compares the values at two interior points:

- So if the left point is lower, the minimum is in the left two-thirds.
- So if the right point is lower, it's in the right two-thirds.

The function defined in the question is:

$$f(x) = \frac{x^2}{\log x} + a \cdot x$$

for $1 < x < 2$, where a is a real number which is provided as the input.
Let's see how the code works like:

- We read a single real number a from input.
- We define the function $f(x)$ using a Python lambda expression it's exactly the math formula given in the problem.
- We call a function called $r - ternary - search$ which performs the recursive ternary search:
  - So first it checks if the interval is small enough, if not, it computes two midpoints.
  - Then it compares $f(m1)$ and $f(m2)$ to decide which part of the interval to keep.
  - It will keep repeating until the range is narrow enough.
- Then the minimum value is printed with 9 decimal digits for precision

Why Ternary Search is better:

- It avoids complex math like derivatives
- precision threshold ensures the result is within the required error margin of 10^{-6}

5 Multiple Medians in Linear Time (1pt)

6 (Programming) Share candies (1pt)

References

- Prof. Yan Gu — Slide 05: Greedy: Binary Search and Flower Vines

Code Submission ID: 314952150, Username: lsuni001

Algorithm 6: Binary Search for Maximum Distributable Candies

```
Function CanDistributeMax($x$, $positions$, $candies$):
    $n \leftarrow$ length of candies;
    $surplus \leftarrow 0$;
    for $i \leftarrow 0$ to $n-2$ do
        $surplus \leftarrow surplus + (candies[i] - x)$;
        $d \leftarrow positions[i+1] - positions[i]$;
        if $surplus \geq 0$ then
            $surplus \leftarrow \max(0, surplus - d)$;
        end
        else
            $surplus \leftarrow surplus - d$;
        end
    end
    $surplus \leftarrow surplus + (candies[n-1] - x)$;
    return $surplus \geq 0$;
Function HighestCandiesPoss($students$):
    Extract $positions$ and $candies$ from $students$;
    $low \leftarrow 0$;
    $high \leftarrow \max(candies)$;
    $best \leftarrow 0$;
    while $low \leq high$ do
        $mid \leftarrow (low + high)//2$;
        if CanDistributeMax($mid$, $positions$, $candies$) then
            $best \leftarrow mid$; $low \leftarrow mid + 1$;
        end
        else
            $high \leftarrow mid - 1$;
        end
    end
    return $best$;
Function Main():
    Read $n$ from input;
    Read $n$ pairs of $(position, candies)$ into $students$;
    Print HighestCandiesPoss($students$);
```

Explanation:

So the goal of the problem is to find the maximum number of candies each student can get after redistributing candies, where some candies are lost during traveling due to the distance between students. To solve this, I used the binary search with a greedy simulation strategy.

Code Implementation Logic:

The function $CanDistributeMax$:
It shows the passing candies from one student to the next from left to right, by keeping track of a surplus which gives us an idea of how many candies are there left over (or in need) after adjusting based on each student's requirement and its transfer penalty.
For each student, it would:

- Subtract the target x from the current student's candies.
- Add that result to the *surplus*.

- Calculate the distance to the next student.
- If the surplus is positive (i.e., extra candies), reduce it by the distance (some candies are lost during transfer).
- If it's negative, the deficit increases by that same distance (it'll be harder to catch up).

For the last student, we adjust the surplus again and return whether it's still non-negative.
For the function $HighestCandiesPoss$:
So this function applies binary search to find the highest possible value of x that passes the check.
It would do this in a loop:

- Picks the middle value mid between low and high.
- Calls $CanDistributeMax$ to see if that value works.
- If it works, it's stored as a candidate answer, and we try for a higher value.
- If not, we lower our expectations by adjusting the high.

When the search ends, the variable will holds the maximum number of candies that each student can get (most possible).

Takeaway:

- **Greedy Forward Simulation:** students with extra pass their surplus to the next, but some candies get lost along the way due to the distance, just like in real life where they may eat it during the trip.
- **Binary Search Efficiency:** We avoid testing every single value by applying binary search, which cuts down the number of tests dramatically.

7 Being Unique (2 pts)