<https://leetcode.com/problems/maximum-value-at-a-given-index-in-a-bounded-array/description/>

You are given three positive integers: n, index, and maxSum. You want to construct an array nums (0-indexed) that satisfies the following conditions:

nums.length == n

nums[i] is a **positive** integer where 0 <= i < n.

abs(nums[i] - nums[i+1]) <= 1 where 0 <= i < n-1.

The sum of all the elements of nums does not exceed maxSum.

nums[index] is **maximized**.

Return nums[index] of the constructed array.

Note that abs(x) equals x if x >= 0, and -x otherwise.

**Example 1:**

**Input:** n = 4, index = 2, maxSum = 6

**Output:** 2

**Explanation:** nums = [1,2,**2**,1] is one array that satisfies all the conditions.

There are no arrays that satisfy all the conditions and have nums[2] == 3, so 2 is the maximum nums[2].

**Example 2:**

**Input:** n = 6, index = 1, maxSum = 10

**Output:** 3

**Constraints:**

1 <= n <= maxSum <= 10^9

0 <= index < n

**Attempt 1: 2024-12-07**

**Solution 1: Binary Search + Greedy (30 min)**

**Style 1: canConstructArray**

class Solution {

    public int maxValue(int n, int index, int maxSum) {

        int lo = 1;

        int hi = maxSum;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // Because we are looking for maximum nums[index],

            // so when we can construct the array, the left

            // range boundary 'lo' can move forward to 'mid + 1'

            // otherwise, if cannot construct the array, the

            // right range boundary 'hi' can move backward to

            // 'mid - 1'

            if(canConstructArray(n, index, maxSum, mid)) {

                lo = mid + 1;

            } else {

                hi = mid - 1;

            }

        }

        // Find upper boundary

        return lo - 1;

    }

    private boolean canConstructArray(int n, int index, int maxSum, int maximizedNumAtIndex) {

        // left side index range [0, index - 1],

        // left side length is 'index - 1 - 0 + 1 = index'

        long leftSum = sumUp(maximizedNumAtIndex - 1, index);

        // right side index range [index + 1, n - 1],

        // right side length is 'n - 1 - (index + 1) + 1 = n - index - 1'

        long rightSum = sumUp(maximizedNumAtIndex - 1, n - index - 1);

        long total = leftSum + rightSum + maximizedNumAtIndex;

        // The sum of all the elements of nums does not exceed maxSum

        return total <= maxSum;

    }

    // We have two constraints:

    // 1.nums[i] is a positive integer where 0 <= i < n

    // 2.abs(nums[i] - nums[i+1]) <= 1 where 0 <= i < n-1

    // For left side, value range for nums[index - 1] to nums[0]

    // can decreasing from peak to 1, and append 1 if not full

    // fill the length

    // For right side, value range for nums[index + 1] to nums[n - 1]

    // can decreasing from peak to 1, and append 1 if not full

    // fill the length

    private long sumUp(int peak, int len) {

        // If peak is larger than length, the sum will include full

        // sequence {peak - len + 1, ..., peak}

        if(peak >= len) {

            return (long) (peak + peak - len + 1) \* len / 2;

        } else {

            // Sum of sequence {1,2, ..., peak}

            long fullPeakSum = (long) (peak + 1) \* peak / 2;

            // Remaining elements are all 1

            long appendOnesSum = (long) (len - peak);

            return fullPeakSum + appendOnesSum;

        }

    }

}

Time Complexity: O(log(maxSum))

Space Complexity: O(1)

**Style 2: cannotConstructArray**

class Solution {

    public int maxValue(int n, int index, int maxSum) {

        int lo = 1;

        int hi = maxSum;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // Because we are looking for maximum nums[index],

            // so when we cannot construct the array, the right

            // range boundary 'hi' can move backward to 'mid - 1'

            // otherwise, if cannot construct the array, the

            // left range boundary 'lo' can move forward to

            // 'mid + 1'

            if(cannotConstructArray(n, index, maxSum, mid)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        // Find upper boundary

        return lo - 1;

    }

    private boolean cannotConstructArray(int n, int index, int maxSum, int maximizedNumAtIndex) {

        // left side index range [0, index - 1],

        // left side length is 'index - 1 - 0 + 1 = index'

        long leftSum = sumUp(maximizedNumAtIndex - 1, index);

        // right side index range [index + 1, n - 1],

        // right side length is 'n - 1 - (index + 1) + 1 = n - index - 1'

        long rightSum = sumUp(maximizedNumAtIndex - 1, n - index - 1);

        long total = leftSum + rightSum + maximizedNumAtIndex;

        // The sum of all the elements of nums exceeds maxSum

        return total > maxSum;

    }

    // We have two constraints:

    // 1.nums[i] is a positive integer where 0 <= i < n

    // 2.abs(nums[i] - nums[i+1]) <= 1 where 0 <= i < n-1

    // For left side, value range for nums[index - 1] to nums[0]

    // can decreasing from peak to 1, and append 1 if not full

    // fill the length

    // For right side, value range for nums[index + 1] to nums[n - 1]

    // can decreasing from peak to 1, and append 1 if not full

    // fill the length

    private long sumUp(int peak, int len) {

        // If peak is larger than length, the sum will include full

        // sequence {peak - len + 1, ..., peak}

        if(peak >= len) {

            return (long) (peak + peak - len + 1) \* len / 2;

        } else {

            // Sum of sequence {1,2, ..., peak}

            long fullPeakSum = (long) (peak + 1) \* peak / 2;

            // Remaining elements are all 1

            long appendOnesSum = (long) (len - peak);

            return fullPeakSum + appendOnesSum;

        }

    }

}

Time Complexity: O(log(maxSum))

Space Complexity: O(1)

**Refer to chatGPT**

**Approach:**

This is a **binary search** problem where we try to maximize the value at the index while adhering to the constraints.

**Key Observations:**

If we place a value x at index, **the values to the left and right will form decreasing sequences starting from x−1,x−2,…, but values can't drop below 1**.

Calculate the sum of the left and right portions:

Left: Sum of decreasing values from index - 1 to 0.

Right: Sum of decreasing values from index + 1 to n - 1.

Add x to get the total sum.

Use binary search to find the maximum x such that the total sum ≤maxSum.

class Solution {

public int maxValue(int n, int index, int maxSum) {

// Binary search bounds

int left = 1, right = maxSum;

while (left < right) {

int mid = (left + right + 1) / 2;

if (canConstruct(mid, n, index, maxSum)) {

left = mid; // Try for a larger value

} else {

right = mid - 1; // Reduce the value

}

}

return left;

}

private boolean canConstruct(int x, int n, int index, int maxSum) {

long leftSum = sumOfSide(x - 1, index); // Sum for the left side

long rightSum = sumOfSide(x - 1, n - index - 1); // Sum for the right side

long total = leftSum + rightSum + x; // Total sum with `x` at the index

return total <= maxSum;

}

private long sumOfSide(int peak, int length) {

// If peak is larger than length, the sum will include full sequence 1, 2, ..., peak

if (peak >= length) {

long fullSequenceSum = (long) (peak + peak - length + 1) \* length / 2;

return fullSequenceSum;

} else {

long fullPeakSum = (long) peak \* (peak + 1) / 2; // Sum of peak values

long onesSum = length - peak; // Remaining elements are all 1

return fullPeakSum + onesSum;

}

}

}

**Refer to**

<https://leetcode.com/problems/maximum-value-at-a-given-index-in-a-bounded-array/solutions/1119801/java-c-python-binary-search/>

**Explanation**

We first do maxSum -= n, then all elements needs only to valid A[i] >= 0

We binary search the final result between left and right, where left = 0 and right = maxSum.

For each test, we check minimum sum if A[index] = a.

The minimum case would be A[index] is a peak in A.

It's arithmetic sequence on the left of A[index] with difference is 1.

It's also arithmetic sequence on the right of A[index] with difference is -1.

On the left, A[0] = max(a - index, 0),

On the right, A[n - 1] = max(a - ((n - 1) - index), 0),

The sum of arithmetic sequence {b, b+1, ....a}, equals to (a + b) \* (a - b + 1) / 2.

**Complexity**

Because O(test) is O(1)

Time O(log(maxSum))

Space O(1)

public int maxValue(int n, int index, int maxSum) {

maxSum -= n;

int left = 0, right = maxSum, mid;

while (left < right) {

mid = (left + right + 1) / 2;

if (test(n, index, mid) <= maxSum)

left = mid;

else

right = mid - 1;

}

return left + 1;

}

private long test(int n, int index, int a) {

int b = Math.max(a - index, 0);

long res = (long)(a + b) \* (a - b + 1) / 2;

b = Math.max(a - ((n - 1) - index), 0);

res += (long)(a + b) \* (a - b + 1) / 2;

return res - a;

}

**Refer to**

[L410.Split Array Largest Sum (Ref.L1011,L704,L1482)](note://WEB4d9d6f26a965381b137615128e2648be)

[L1011.Capacity To Ship Packages Within D Days (Ref.L410,L1482)](note://WEB87ca8f9dd9a14fb4535856a4e0f8f4b1)

[L1482.Minimum Number of Days to Make m Bouquets (Ref.L410,L1011)](note://WEBd5ce394160da45a79e92dbbc806111ec)

[L1283.Find the Smallest Divisor Given a Threshold (Ref.L410,L1011,L1482)](note://76952362C4844CB3B68E28EA09F2EE58)