Solution: Put Marbles in Bags

Statement

You are given `k` bags and a 0-indexed integer array, `weights`, where `weights[i]` represents the weight of the i^{th} marble.

Your task is to divide the marbles into the `k` bags according to the following rules:

1. No bag can be empty.
2. If the i^{th} marble and the j^{th} marble are placed in the same bag, then all marbles with indexes between `i` and `j` (inclusive) must also be placed in that same bag.
3. If a bag contains all the marbles from index `i` to `j` (inclusive), its **cost** is calculated as `weights[i] + weights[j]`.

After distributing the marbles, the sum of the costs of all the `k` bags is called the **score**.

Return the difference between the maximum and minimum scores achievable by distributing the marbles into the `k` bags.

Constraints:

- $1 \leq$ `k` $\leq$ `weights.length` $\leq 10^5$
- $1 \leq$ `weights[i]` $\leq 10^9$

Solution

The main idea is to split the weights array into `k` bags by making `k - 1` cuts between marbles. To do this efficiently, we use the sort and search pattern, which helps us quickly identify the best positions to split for both maximum and minimum scores. The key observation is that when we make a cut between two marbles, the marble before the cut becomes the **end** of one bag, and the marble after the cut becomes the **start** of the next bag. So, every cut adds two boundary marbles that affect the total score, because each bag's score is the sum of its first and last marble, and we can include these boundary marbles in the final sum of all the scores.

> **Note:** A cut between marble 3 and 4 means marble 3 ends one bag, and marble 4 starts the next; their sum contributes to the total score.

The first and last marbles of the entire array are always part of the score, so the only part that changes with different splits is the contribution from the cuts. To capture these, we calculate the sum of each pair of adjacent marbles in the array. Then, we iterate through the array from the 0^{th} index up to the second last, and at each step, we add the weight at the current position to the weight of the marble immediately following it. This gives us the sum of each consecutive pair.

> **Example:** If the weights are `[1, 3, 5, 2]`, we get the pairwise sums:
> `[(1 + 3 = 4), (3 + 5 = 8), (5 + 2 = 7)]`

These pairwise sums represent the potential score impact of placing a cut at that position. By sorting these sums, we can easily pick the `k - 1` largest to get the maximum score, and the `k - 1` smallest to get the minimum score. The final answer is the difference between these two totals.

Now, let's look at the solution steps below:

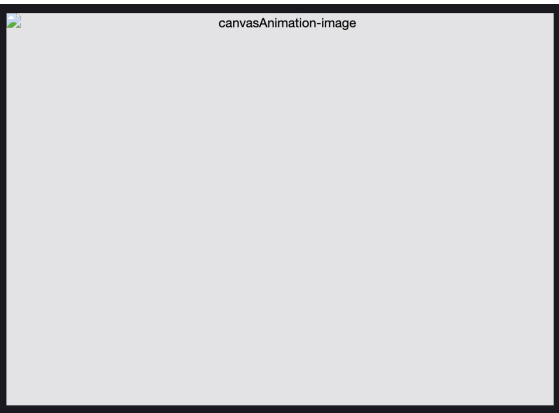
1. If `k` is 1, all marbles must be placed in a single bag. This case has no partitioning; the difference between the maximum and minimum scores is `0`, so we return `0`.
2. Otherwise, we calculate the initial cost of a single bag containing all marbles by summing the first and last weights in the array. We do this by `initial_cost` = `weights[0]` + `weights[-1]`.
3. Then, we compute the pairwise sums of all adjacent marbles. For each index `i` from `0` to `n - 2` (where `n` is the length of `weights`), we calculate `weights[i] + weights[i + 1]` and store it in `pairwise_sums`.
4. Next, we sort the `pairwise_sums` list in ascending order. This allows us to efficiently access the smallest and largest values needed for calculating the minimum and maximum scores.
5. To calculate the maximum score:
    I. We take the largest `(k - 1)` values from the end of the sorted pairwise sums list using `pairwise_sums[-(k - 1):]`, and add their sum to `initial_cost`. After calculating, we store the result in `max_score`.
6. To calculate the minimum score:
    I. Similar to what we did for the maximum score, we take the smallest `(k - 1)` values from the beginning of the sorted pairwise sums list using `pairwise_sums[:k - 1]`, and add their sum to `initial_cost`. After calculating, we store the result in `min_score`.
7. Finally, we return the difference between the maximum and minimum scores using `max_score - min_score`.

Let's look at the following illustration to get a better understanding of the solution:

Solution | Commented Solution

```python
def put_marbles(weights, k):
    # If k is 1, all marbles must be in one bag,
    if k == 1:
        return 0

    # Calculate the cost of the initial single ba
    n = len(weights)
    initial_cost = weights[0] + weights[-1]

    # Calculate the pairwise sums
    pairwise_sums = [weights[i] + weights[i + 1]

    # Sort the pairwise sums to help find the max
    pairwise_sums.sort()

    # Calculate the maximum score by selecting la
    max_score = initial_cost + sum(pairwise_sums[

    # Calculate the minimum score by selecting sm
    min_score = initial_cost + sum(pairwise_sums[

    # Return the difference between maximum and m
    return max_score - min_score


def main():
    test_cases = [
        ([5], 1),
        ([1, 7, 23, 29, 47], 3),
        ([1, 2, 3, 5], 4),
        ([1, 2, 3, 5], 2)
```

Run



1 / 11

Let's look at the code for the solution we just discussed.

Time complexity

The overall time complexity of the solution above is $O(n \log n)$, here's how:

- Calculating the pairwise sums requires a single pass through the array, which is $O(n)$.
- Sorting these sums takes $O(n \log n)$.
- Calculating the maximum and minimum score $(k - 1)$ sums from the sorted list is $O(k)$, which is negligible compared to the sorting step.

Space complexity

The space complexity of the solution provided is dominated by the pairwise sums that are stored in an auxiliary list, which requires $O(n)$ space.