1167. Minimum Cost to Connect Sticks

Medium Greedy Array Heap (Priority Queue) Leetcode Link

Problem Description

In this problem, we are given an array `sticks`, where each element represents the length of a stick. We can connect any two sticks to create a longer stick, but the cost of connecting two sticks is equal to the sum of their lengths. Our goal is to combine all the sticks into one by repeatedly connecting two sticks at a time, while minimizing the total cost of combining them.

To visualize, imagine starting with several separate sticks of different lengths. You can choose any two sticks, combine them, and you will get a new stick whose length is the sum of the two sticks you combined. The challenge lies in finding the sequence of connections that results in the smallest possible sum of all the intermediate costs.

For example, if we have sticks of lengths `[2, 4, 3]`, we could:

- Combine sticks of lengths 2 and 3 (costing 5), to get `[5, 4]`.
- Then combine sticks of lengths 5 and 4 (costing 9), to get `[9]`.

The total cost would be `5 + 9 = 14`. We are asked to find the strategy that gives us the least cost to combine all sticks into one.

Intuition

The intuition behind the solution is to always combine the two shortest sticks together. This approach is efficient because by combining the smaller sticks first, we prevent larger numbers from being added repeatedly in subsequent combinations, which would happen if we combined longer sticks early on.

To implement this strategy, we can use a data structure called a priority queue or a min-heap. A priority queue allows us to efficiently retrieve and remove the smallest element, while also efficiently adding new elements.

Here's how we arrive at our solution step by step:

1. First, we add all the sticks lengths to a min-heap. A min-heap is a binary tree where the parent node is always less than its child nodes. This property will be useful because we always want to combine the two smallest sticks.
2. Then, we perform the following until there is only one stick left in the heap:
    - Take out the two smallest sticks from the min-heap (this operation is efficient due to the min-heap's properties).
    - Calculate the cost of connecting these two sticks (simply the sum of the two lengths).
    - Add this cost to the running total cost.
    - Insert the resulting new stick back into the min-heap.
3. This process is repeated until only one stick remains in the min-heap. The running total cost at this point represents the minimum cost to combine all sticks into one, which is what we return.

Using the priority queue ensures that we are always combining the smallest sticks, and the efficient insertions and deletions provided by the min-heap make it optimal for such repetitive operations as required by our approach.

Solution Approach

The solution approach for combining the sticks in a cost-effective manner involves the use of a min-heap, which is a type of priority queue. Let's discuss how this data structure is applied in the given Python code to solve the problem.

1. **Heapify the list of sticks**: The first operation is to convert the list of stick lengths into a min-heap using the `heapify` function. In Python, the standard library `heapq` module provides the functionality for a min-heap. By heapifying the array, we create a data structure that allows us to efficiently retrieve the smallest elements.

```
heapify(sticks)
```

2. **Initialize a variable to keep track of the total cost**: We start by setting the total cost `ans` to 0. This variable will accumulate the cost of connecting the sticks.

```
ans = 0
```

3. **Iterate until one stick remains**: We run a loop until there is only one stick left in the heap. Since we always remove two sticks and add one back into the min-heap in each iteration, the loop stops when we cannot perform this operation anymore, which is when the length of the `sticks` array gets reduced to 1.

```
while len(sticks) > 1:
```

4. **Pop the two smallest sticks and add the cost**: Inside the loop, we use the `heappop` function to pop the two smallest elements in the min-heap. The sum of these two sticks represents the cost to connect them, which we add to our total cost `ans`. Then, a new stick with this sum length is formed.

```
x = heappop(sticks) + heappop(sticks)
ans += x
```

5. **Push the new stick into the heap**: We have a new stick that needs to go into our pile so that we can continue the process. We push the new stick into the min-heap using `heappush`. The heap property will automatically be maintained, ensuring that this new stick goes into the correct place in the priority queue.

```
heappush(sticks, x)
```

6. **Return the total cost**: After the while loop terminates (when only one stick is left), the `ans` contains the minimum cost to connect all sticks together, which is what we return as the solution.

```
return ans
```

Through the use of a min-heap, this algorithm ensures that at every step, the cost is minimized by always combining the shortest sticks available. The operations inside the loop—popping two elements and pushing one element—all have a logarithmic time complexity relative to the number of elements in the heap, making this approach efficient.

The reference solution approach appropriately cites "Priority queue," which is the concept employed throughout the solution to maintain the efficiency of operations for what would otherwise be a more computationally expensive problem.

Example Walkthrough

Let's use a small example to understand how the solution approach is applied. Suppose we have an array of stick lengths `[2, 4, 3]`. Here's how we apply the steps of the solution:

1. **Heapify the list of sticks**: We start by converting our array `[2, 4, 3, 1]` into a min-heap. The `heapify` function arranges the sticks so that the smallest ones can be accessed quickly. Our min-heap will look something like this: `[1, 2, 3, 4]`, where 1 is at the root as it's the smallest.
2. **Initialize the total cost**: We'll create a variable `ans` to keep track of the total cost, initially setting it to 0.
3. **Iterate until one stick remains**: We know that the number of sticks will reduce by one with each iteration because we're combining two sticks at a time. We stop when we have one stick left.
4. **Pop the two smallest sticks and add the cost**: Now we'll look at our min-heap `[1, 2, 3, 4]`, and pop out the two smallest elements, which are 1 and 2. We combine them at a cost of `1 + 2 = 3`, and add that to our running total `ans`. Now, `ans = 3`.
5. **Push the new stick into the heap**: After combining the sticks with lengths 1 and 2, we get a new stick of length 3 which we push back into our min-heap. The min-heap will reorder itself and look like `[3, 3, 4]`.
6. **Repeat the process**: Again, we pop out two smallest sticks from `[3, 3, 4]`, which are both of length 3, combine them at a cost of `3 + 3 = 6`, and add that to our `ans` making it `3 + 6 = 9`. We push the new stick of length 6 back into the heap, which now looks like `[4, 6]`.
7. **Final iteration**: We pop out the remaining two sticks, combine them at a cost of `4 + 6 = 10`, and add that to our total cost, making it `ans = 9 + 10 = 19`. The min-heap is now `[10]`, but since there's only one stick left, we're done.
8. **Return the total cost**: The variable `ans` now contains the value 19, which is the minimum cost to connect all the sticks in our original array `[2, 4, 3, 1]`.

The sequence of costs incurred at each step were `[3, 6, 10]`, and the sum of these costs, 19, is the least possible total cost to combine the sticks into one. Through the use of heap operations, we've been able to efficiently combine the sticks in a cost-effective manner.

Python Solution

```python
from heapq import heapify, heappop, heappush

class Solution:
    def connectSticks(self, sticks: List[int]) -> int:
        # First, convert the list of sticks into a min-heap
        heapify(sticks)

        # Initialize total cost to 0
        total_cost = 0

        # Continue combining sticks until there's only one stick remaining
        while len(sticks) > 1:
            # Pop the two smallest sticks from the heap
            stick1 = heappop(sticks)
            stick2 = heappop(sticks)

            # The cost to combine these two sticks
            combined_cost = stick1 + stick2

            # Add the combined_cost to the total cost
            total_cost += combined_cost

            # Push the combined stick back into the heap
            heappush(sticks, combined_cost)

        # Return the total cost incurred to combine all sticks
        return total_cost
```

Java Solution

```java
class Solution {
    public int connectSticks(int[] sticks) {
        // Initialize a minimum heap (priority queue) to store the sticks in ascending order
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        // Add all the sticks to the heap
        for (int stick : sticks) {
            minHeap.offer(stick);
        }

        // Initialize the total cost to connect all the sticks
        int totalCost = 0;

        // Continue combining sticks until only one stick remains
        while (minHeap.size() > 1) {
            // Retrieve and remove the two smallest sticks
            int stick1 = minHeap.poll();
            int stick2 = minHeap.poll();

            // Calculate the cost of connecting the two smallest sticks
            int cost = stick1 + stick2;

            // Add the cost to the total cost
            totalCost += cost;

            // Add the combined stick back to the heap
            minHeap.offer(cost);
        }

        // Return the total cost to connect all the sticks
        return totalCost;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int connectSticks(vector<int>& sticks) {
        // Use a min heap to store and retrieve sticks with the smallest lengths first
        priority_queue<int, vector<int>, greater<int>> min_heap;

        // Insert all the sticks into the min heap
        for (int& stickLength : sticks) {
            min_heap.push(stickLength);
        }

        int totalCost = 0;  // Initialize total cost to 0

        // Continue this process until only one stick remains in the min heap
        while (min_heap.size() > 1) {
            // Extract the two smallest sticks from the min heap
            int stick1 = min_heap.top();
            min_heap.pop();
            int stick2 = min_heap.top();
            min_heap.pop();

            // Combine the two sticks
            int newStick = stick1 + stick2;

            // Add the cost of combining the two sticks
            totalCost += newStick;

            // Put the combined stick back into the min heap
            min_heap.push(newStick);
        }

        // Return the total cost to connect all the sticks
        return totalCost;
    }
};
```

Typescript Solution

```typescript
// Definition of our comparison function type.
type CompareFunction<T = (...args: any) => number>;

// The data (heap) structure with a null placeholder as the first element.
let heapData: Array<number | null> = [null];

// The comparison function, defaulting to a min-heap behavior.
let compare: CompareFunction<number> = (lhs, rhs) => (lhs < rhs ? -1 : lhs > rhs ? 1 : 0);

// Initializes the heap with optional data and comparison function.
function initializeHeap(data?: number[], customCompare?: CompareFunction<number>): void {
    if (customCompare !== undefined) {
        compare = customCompare;
    }
    heapData = [null, ...(data || [])];
    for (let i = size(); i > 0; i--) {
        heapify(i);
    }
}

// Returns the size of the heap.
function size(): number {
    return heapData.length - 1;
}

// Pushes a new value into the heap.
function push(value: number): void {
    heapData.push(value);
    let i = size();
    while (i > 1 && compare(heapData[i], heapData[i >> 1]) < 0) {
        swap(i, i >> 1);
        i >>= 1;
    }
}

// Pops the top value from the heap.
function pop(): number {
    swap(1, size());
    const poppedValue = heapData.pop();
    heapify(1);
    return top;
}

// Returns the top value in the heap without removing it.
function top(): number {
    return heapData[1];
}

// Heapifies the data at the specified index.
function heapify(i: number): void {
    let min = i;
    const n = heapData.length;
    const l = i * 2;
    const r = i * 2 + 1;
    if (l < n && compare(heapData[l], heapData[min]) < 0) min = l;
    if (r < n && compare(heapData[r], heapData[min]) < 0) min = r;
    if (min !== i) {
        swap(i, min);
        heapify(min);
    }
}

// Clears the heap data.
function clear(): void {
    heapData = [null];
}

// Swaps two values in the heap data.
function swap(i: number, j: number): void {
    [heapData[i], heapData[j]] = [heapData[j], heapData[i]];
}

// Main function that calculates the minimum cost to connect the sticks.
function connectSticks(sticks: number[]): number {
    initializeHeap(sticks);
    let totalCost = 0;
    while (size() > 1) {
        const x = pop();
        const y = pop();
        totalCost += x + y;
        push(x + y);
    }
    return totalCost;
}
```

Time and Space Complexity

Time Complexity

The given Python code implements a min-heap to connect sticks with minimum cost. The operation `heapify(sticks)` has a complexity of $O(n)$ where n is the number of sticks. The `while` loop runs until there is only one stick left, which happens after $n-1$ iterations because in each iteration, two sticks are removed and one is added back to the heap.

Inside the loop, two operations of `heappop()` are performed with a complexity of $O(\log n)$ each, to remove the two smallest sticks, and one operation of `heappush()`, also with a complexity of $O(\log n)$, to add the combined stick back to the heap. Considering that there are $n-1$ iterations, and each iteration has a constant number of heap operations, the total time complexity inside the loop is $O((n-1) * \log n)$, which simplifies to $O(n \log n)$.

Therefore, the overall time complexity of the code is $O(n) + O(n \log n)$, which we can simplify to $O(n \log n)$ since $O(n)$ is dominated by $O(n \log n)$ for large n.

Space Complexity

The space complexity of the code comes from the storage used for the heap. The `heapify()` function is performed in-place, and no additional space is used besides the `sticks` list, which means that the space complexity is $O(n)$ where n is the initial number of sticks in the input list. The integer `ans` and temporary variable `x` use constant space.

Thus, the space complexity of the algorithm is $O(n)$.