1167. Minimum Cost to Connect Sticks Heap (Priority Queue) Medium Greedy Array

Leetcode Link

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

Problem Description

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:

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

retrieve and remove the smallest element, while also efficiently adding new elements.

Calculate the cost of connecting these two sticks (simply the sum of the two lengths).

queue. Let's discuss how this data structure is applied in the given Python code to solve the problem.

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].

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

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).

 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.

structure that allows us to efficiently retrieve the smallest elements.

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

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

1 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

and add one new stick on 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.

1 while len(sticks) > 1:

1 ans = 0

cost of connecting the sticks.

stick with this sum length is formed.

all sticks together, which is what we return as the solution.

z = heappop(sticks) + heappop(sticks)

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

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

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

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.

1 return ans

1 heappush(sticks, z)

Example Walkthrough

the root as it's the smallest.

looks like [4, 6].

effective manner.

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

27

28

11

12

13

14

Java Solution

class Solution {

C++ Solution

1 class Solution {

int connectSticks(vector<int>& sticks) {

for (auto& stickLength : sticks) {

min_heap.push(stickLength);

int stick1 = min_heap.top();

int stick2 = min_heap.top();

int newStick = stick1 + stick2;

// Combine the two sticks

totalCost += newStick;

min_heap.push(newStick);

1 // Definition of our comparison function type.

let heapData: Array<number | null> = [null];

type CompareFunction<T> = (lhs: T, rhs: T) => number;

// The comparison function, defaulting to a min-heap behavior.

// Initializes the heap with optional data and comparison function.

while (min_heap.size() > 1) {

min_heap.pop();

min_heap.pop();

return totalCost;

Typescript Solution

// Insert all the sticks into the min heap

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

// Add the cost of combining the two sticks

// Return the total cost to connect all the sticks

// Put the combined stick back into the min heap

// The data (heap) structure with a null placeholder as the first element.

let compare: CompareFunction<number> = (lhs, rhs) => (lhs < rhs ? -1 : lhs > rhs ? 1 : 0);

function initializeHeap(data?: number[], customCompare?: CompareFunction<number>): void {

2 public:

9

11

12

13

14

15

16

19

20

21

23

24

25

26

27

28

29

30

31

33

34

36

35 };

9

59

60

62

65

66

67

70

72

75

76

78

79

80

81

82

83

85

84 }

71 }

61 }

Python Solution

from heapq import heapify, heappop, heappush

Initialize total cost to 0

stick1 = heappop(sticks)

stick2 = heappop(sticks)

while len(sticks) > 1:

return total_cost

int totalCost = 0;

total_cost = 0

1]. Here's how we apply the steps of the solution:

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

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,

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

maintain the efficiency of operations for what would otherwise be a more computationally expensive problem.

2. Initialize the total cost: We'll create a variable ans to keep track of the total cost, initially setting it to 0.

6. Return the total cost: After the while loop terminates (when only one stick is left), the ans contains the minimum cost to connect

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

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

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,

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-

3. Iterate until one stick remains: We know that the number of sticks will reduce by one with each iteration because we're

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].

making it ans = 9 + 10 = 19. The min-heap is now [10], but since there's only one stick left, we're done.

back into our min-heap. The min-heap will reorder itself and look like [3, 3, 4].

Continue combining sticks until there's only one stick remaining

Pop the two smallest sticks from the heap

The cost to combine these two sticks

Add the combined_cost to the total cost

Push the combined stick back into the heap

Return the total cost incurred to combine all sticks

// Initialize the total cost to connect all the sticks

// Continue combining sticks until only one stick remains

// Use a min heap to store and retrieve sticks with the smallest lengths first

// Continue this process until only one stick remains in the min heap

// Extract the two smallest sticks from the min heap

priority_queue<int, vector<int>, greater<int>> min_heap;

combined_cost = stick1 + stick2

heappush(sticks, combined_cost)

total_cost += combined_cost

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

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);

```
while (minHeap.size() > 1) {
15
               // Retrieve and remove the two smallest sticks
16
                int stick1 = minHeap.poll();
                int stick2 = minHeap.poll();
18
19
               // Calculate the cost of connecting the two smallest sticks
20
               int cost = stick1 + stick2;
21
               // Add the cost to the total cost
24
                totalCost += cost;
25
               // Add the combined stick back to the heap
26
27
               minHeap.offer(cost);
28
29
30
           // Return the total cost to connect all the sticks
31
           return totalCost;
33 }
34
```

12 if (customCompare) -13 compare = customCompare; 14 15

```
heapData = [null, ...(data || [])];
        for (let i = size(); i > 0; i--) {
17
            heapify(i);
18
19 }
20
21 // Returns the size of the heap.
22 function size(): number {
        return heapData.length - 1;
23
24 }
25
   // 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) {</pre>
31
            swap(i, i \gg 1);
32
            i >>= 1;
33
34
35
   // Pops the top value from the heap.
   function pop(): number {
38
        swap(1, size());
        const top = heapData.pop()!;
39
40
        heapify(1);
41
        return top;
42 }
43
   // Returns the top value in the heap without removing it.
   function top(): number {
        return heapData[1]!;
46
47 }
48
   // Heapifies the data at the specified index.
   function heapify(i: number): void {
        let min = i;
51
        const n = heapData.length;
52
53
        const l = i * 2;
54
        const r = i * 2 + 1;
55
        if (l < n && compare(heapData[l]!, heapData[min]!) < 0) min = l;</pre>
56
        if (r < n && compare(heapData[r]!, heapData[min]!) < 0) min = r;</pre>
57
        if (min !== i) {
58
            swap(i, min);
```

Time and Space Complexity

O((n-1) * log n), which simplifies to O(n log n).

heapify(min);

// Clears the heap data.

function clear(): void {

heapData = [null];

// Swaps two values in the heap data.

initializeHeap(sticks);

const x = pop();

const y = pop();

totalCost += x + y;

let totalCost = 0;

while (size() > 1) {

push(x + y);

return totalCost;

function swap(i: number, j: number): void {

function connectSticks(sticks: number[]): number {

[heapData[i], heapData[j]] = [heapData[j], heapData[i]];

// Main function that calculates the minimum cost to connect the sticks.

Time Complexity

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

The given Python code implements a min-heap to connect sticks with minimum cost. The operation heapify(sticks) has a

iterations because in each iteration, two sticks are removed and one is added back to the heap.

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

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 z use constant space. Thus, the space complexity of the algorithm is O(n).