Priority Queue One Shot

Assignment Solutions

PW SKILLS

Q1. Given K sorted arrays of size N each, merge them and print the sorted output.

Input:

Arrays are given as a 2D array:

```
arr = {{1, 3, 5, 7},
       {2, 4, 6, 8},
       {0, 9, 10, 11}}
```

Output:

Merged array: 0 1 2 3 4 5 6 7 8 9 10 11

A1. Solution Code: https://pastebin.com/ebsf8WwB

Output:

```
Merged array: 0 1 2 3 4 5 6 7 8 9 10 11
```

Explanation:

1. First, we initialize a priority queue to store the smallest element from each array.
2. We calculate the total number of elements in all arrays and add the first element of each array to the priority queue if it exists.
3. We create an empty array to store the merged elements and iterate over the priority queue, removing the smallest element and adding it to the merged array.
4. If there are still elements remaining in the array that the removed element belonged to, we add the next element from that array to the priority queue.
5. We repeat this process until all elements have been removed from the priority queue and added to the merged array.
6. Finally, we return the merged array.

Time Complexity: O(N log k), where N is the total number of elements across all arrays and k is the number of input arrays. This is because the algorithm involves adding and removing elements from a priority queue, which has a log k time complexity for both operations, and we perform these operations for each of the N elements in the input arrays.

Space complexity: O(k), which is the space required to store the priority queue. In the worst case, all elements from all arrays are added to the priority queue at the same time, which results in a total of k elements in the priority queue. The output array also requires O(N) space to store the merged elements.

Q2. Given a list of intervals, merge overlapping intervals using a priority queue

Input:

```
Intervals:
1 3
2 6
8 10
15 18
```

Output:

```
Merged Intervals:
1 6
8 10
15 18
```

A2. Solution Code: https://pastebin.com/UzYZ9xkS

Output:

```
Merged Intervals:
1 6
8 10
15 18
```

Explanation:

1. Create an Interval class with two integer fields representing the start and end points of an interval.
2. Create a priority queue to hold the intervals sorted by their start points in ascending order.
3. Iterate through the list of intervals and add each interval to the priority queue.
4. Create a new list to hold the merged intervals.
5. Initialize a variable named "previous" to hold the first interval in the priority queue.
6. While the priority queue is not empty:
   a. Remove an interval from the priority queue.
   b. Compare its start point to the end point of the "previous" interval.
   c. If the start point of the removed interval is less than or equal to the end point of the "previous" interval, merge the two intervals by updating the end point of the "previous" interval to the maximum of its current value and the end point of the removed interval.
   d. If the start point of the removed interval is greater than the end point of the "previous" interval, add the "previous" interval to the merged list and update the "previous" interval to the removed interval.
7. Add the last "previous" interval to the merged list.
8. Return the merged list.

Time complexity: O(N log N), where N is the number of intervals. This is because we need to iterate over each interval once, which takes O(N) time, and for each interval, we need to add it to the priority queue and remove intervals from the queue, which takes O(log N) time per operation.

Space complexity: O(N), where N is the number of intervals. This is because we need to store the input intervals in a priority queue and the merged intervals in a new list. In the worst case, when there are no overlapping intervals, the merged list will contain all the input intervals, so it will have the same size as the input list.

Q3. Implement a stack using a priority queue.
Your stack should perform the following operations:

1. **push**
2. **pop**
3. **peek**
4. **getSize**
5. **isEmpty**

A3. Solution Code: https://pastebin.com/ZACM5SaP

Output:

Explanation:

1. The program implements a stack data structure using a priority queue.
2. The Stack class has three private variables: elements (a priority queue of StackElement objects), size (an integer representing the number of elements in the stack), and StackElement (a struct containing the value of an element and its index).
3. The push method adds a new element to the stack by creating a new StackElement with the given value and the current size as its index. The element is then added to the priority queue, and the size is incremented.
4. The pop method removes and returns the top element from the stack by first checking if the stack is empty. If it is not empty, the size is decremented, and the top element's value is returned. The element is also removed from the priority queue.
5. The peek method returns the value of the top element without removing it. It also checks if the stack is empty before accessing the top element.
6. The getSize method returns the current size of the stack.
7. The isEmpty method returns true if the stack is empty, and false otherwise.
8. The main function creates an instance of the Stack class, adds some elements to it using the push method, and then removes and prints the elements using the pop method. It also prints the size of the stack and whether it is empty.

Time complexity:

1. Push operation: O(log n), where n is the number of elements in the priority queue.
2. Pop operation: O(log n), where n is the number of elements in the priority queue.
3. Peek operation: O(1).
4. Get size operation: O(1).
5. Is empty operation: O(1).

Space complexity: O(n), where n is the number of elements in the stack. This is because the priority queue is used to store the elements of the stack, and its space usage is proportional to the number of elements it contains.

Q4. You are given two integer arrays sorted in ascending order and an integer k. Return the k pairs with the smallest sums.

Input:

nums1 = {1, 7, 11}
nums2 = {2, 4, 6}
Output:
Pairs:
1 2
1 4
1 6

A4. Solution Code: https://pastebin.com/aRASiyu6

Output:

```
Pairs:
1 2
1 4
1 6
```

Explanation:

1. Given two sorted integer arrays nums1 and nums2, and an integer k, we need to find the k pairs with the smallest sums.
2. We create a priority queue of pairs {nums1[i], nums2[j], j} (where j is the index of the second element of the pair) and initially add the pairs {nums1[i], nums2[0], 0} to the priority queue for i ranging from 0 to k-1. The priority queue is ordered based on the sum of the pair elements.
3. We repeatedly pop the smallest pair from the priority queue, add it to the result vector, and if the second element of the pair is not the last element of nums2, we add the next pair {nums1[i], nums2[j+1], j+1} to the priority queue. We repeat this process k times or until the priority queue is empty.
4. Finally, we return the k pairs with the smallest sums in the result vector.

Time complexity: O(k log(min(n1, n2, k))), where n1 and n2 are the sizes of the two input arrays. The priority queue can have at most k elements, and each operation of pushing and popping from the queue takes log(k) time. We push k elements to the queue in the worst case, and then we pop k elements from the queue while adding new elements, until we have k elements in the result vector. Since we only keep at most k elements in the queue at any time, the size of the queue is at most min(n1, n2, k).

Space complexity: O(min(n1, n2, k)), because we only need to store at most k elements in the priority queue and at most k elements in the result vector. Therefore, the space used by the program is proportional to the minimum of the sizes of the two input arrays and the value of k.

Q5. You are given two 0-indexed integer arrays nums1 and nums2 of equal length n and a positive integer k. You must choose a subsequence of indices from nums1 of length k.
For chosen indices i0, i1, ..., ik - 1, your score is defined as:

- **The sum of the selected elements from nums1 multiplied with the minimum of the selected**

elements from nums2.
It can defined simply as: (nums1[i0] + nums1[i1] +...+ nums1[ik - 1]) * min(nums2[i0] , nums2[i1], ... ,nums2[ik - 1]).
Return the maximum possible score.
A subsequence of indices of an array is a set that can be derived from the set {0, 1, ..., n-1} by deleting some or no elements.
Input: nums1 = [1,3,3,2], nums2 = [2,1,3,4], k = 3
Output: 12
Explanation:
The four possible subsequence scores are:
- We choose the indices 0, 1, and 2 with score = (1+3+3) * min(2,1,3) = 7.
- We choose the indices 0, 1, and 3 with score = (1+3+2) * min(2,1,4) = 6.
- We choose the indices 0, 2, and 3 with score = (1+3+2) * min(2,3,4) = 12.
- We choose the indices 1, 2, and 3 with score = (3+3+2) * min(1,3,4) = 8.
Therefore, we return the max score, which is 12.

Code link: https://pastebin.com/gNQBj2pq

Explanation:

1. Create a vector of pairs v to store pairs of elements from nums2 and nums1. Each pair represents a combination of one element from nums2 and its corresponding element from nums1.
2. Sort the vector v in descending order based on the first element of each pair. This step ensures that elements with larger values in nums2 are considered first.
3. Initialize ans and currSum as 0. ans will store the maximum score, and currSum will keep track of the sum of selected elements from nums1.
4. Create a priority queue pq with the greater<int> comparator. This priority queue will store the selected elements from nums1 in ascending order.
5. Iterate over the first k-1 pairs in v. For each pair:
   - Add the second element (from nums1) to currSum.
   - Push the second element into the priority queue pq.
6. This step selects the largest k-1 elements from nums1 and stores their sum in currSum.
7. Iterate from the k-1-th pair in v up to the end. For each pair:
   - Add the second element (from nums1) to currSum.
   - Push the second element into pq.
   - Calculate the score by multiplying currSum with the first element (from nums2) of the current pair, and update ans if necessary.
   - Subtract the smallest element from pq (top of the priority queue) from currSum.
   - Pop the smallest element from pq.
8. This step selects the remaining elements from nums1 and calculates the score by considering the current element from nums2. It also updates currSum by subtracting the smallest element to maintain a sliding window of size k.
9. Return ans as the maximum possible score.

Output:

```
Maximum possible score: 12


...Program finished with exit code 0
Press ENTER to exit console.
```