計算方法設計 Design and Analysis of Algorithms (11310CS431102)

Extra Assignment: Programming Assignment

Problem report and Explanation of Solutions

Student: 葛洋晴 ID: 108006205

All problems were solved using C/C++, if you would like to know how to compile and execute each file go to the "README.md" file. In addition, important comments can be found in each problem script.

Problem 1

Description:

You are given an array containing n positive integers. Your task is to divide the array into k subarrays so that the maximum sum in a subarray is as small as possible. *Explanation:* An optimal division is [2,4], [7], [3,5] where the sums of the subarrays are 6, 7, 8. The largest sum is the last sum 8.

Input

The first input line contains two integers n and k: the size of the array and the number of subarrays in the division. The next line contains n integers $x_1, x_2, \ldots, x_n$: the contents of the array.

Output

Print one integer: the maximum sum in a subarray in the optimal division.

Constraints

- $1 \leq n \leq 2 \cdot 10^5$
- $1 \leq k \leq n$
- $1 \leq x_i \leq 10^9$

Example

Input:

```
5 3
2 4 7 3 5
```

Output:

```
8
```

Solution proposed:

Pseudocode of the solution

```
MAX-SUBARRAY-SUM(A, k)

1   Left = the max element in array A
2   Right = total sum of the elements in A
3   Result = Right
4   while Left ≤ Right:
5       middle = (Right + Left) / 2
6       if IS-VALID-PARTITION(A, k, middle):
7           Result = middle
8           Right = middle – 1
9       else
10          Left = middle + 1
```

```
IS-VALID-PARTITION(A, k, middle)

1   Sum = 0
2   Subarrays = 1
3   for each number in the array
4       if Sum + number > middle
5           Subarrays = Subarrays + 1
6           Sum = number
7           if Subarrays > k:
8               return false
9       else
10          Sum = Sum + number
11  return true
```

Time complexity and how it works:

Array Initialization for n elements takes $\boldsymbol{O}(n)$ and the use of Binary Search reduces the time complexity to $\boldsymbol{O}\big(n \cdot log(sum\ of\ array)\big)$, where sum of array is from the maximum element to the sum of all elements.

The smallest possible maximum subarray sum Left is the largest single element in the array, this is because, in the worst case, a subarray may consist of only one element, so the maximum subarray sum cannot be smaller than the largest element.

The function IS-VALID-PARTITION checks whether it is possible to divide the array into at most k subarrays such that no subarray's sum exceeds mid. And if the number of subarrays exceeds k, return false (partition is invalid).

Problem 2

Description:

There are ***n*** cities and ***m*** flight connections between them. Your task is to determine the length of the shortest route from Syrjälä to every city.

Input

The first input line has two integers n and m: the number of cities and flight connections. The cities are numbered 1,2, … , n, and city 1 is Syrjälä. After that, there are *m* lines describing the flight connections. Each line has three integers *a, b* and *c*: a flight begins at city *a*, ends at city *b*, and its length is *c*. Each flight is a one-way flight.

You can assume that it is possible to travel from Syrjälä to all other cities.

Output

Print *n* integers: the shortest route lengths from Syrjälä to cities 1,2, … , n.

Constraints

- $1 \leq n \leq 10^5$
- $1 \leq m \leq 2 \cdot 10^5$
- $1 \leq a, b \leq n$
- $1 \leq c \leq 10^9$

Example

Input:

```
3 4
1 2 6
1 3 2
3 2 3
1 3 4
```

Output:

```
0 5 2
```

Solution proposed:

Dijkstra's Algorithm is efficient for finding the shortest paths from a single source node with non-negative edges. Below is the Pseudocode of the Algorithm inspired mostly from the book *Introduction to Algorithm*, 4th edition, T.H. Cormen et al.

Pseudocode of my solution:

```
DIJKSTRA(G, distance, n)
// G is a vector of vectors which stores pairs of numbers, that is G[a] is the vertex we are at,
// and the vertex b and weight w as a pair (b, w).
1   Q = ∅        // min-heap as priority queue for storing the pairs according to their weights.
2   distance[1] = 0
3   Add the pair (b, w) to Q
4   while Q is not empty:
5       d = w from the pair (b, w) from the priority queue Q
6       u = b from the pair (b, w)
7       Pop the first pair from Q
8       if d > distance[u]:      //only the most up-to-date and valid distances are processed.
9           skip to the next pair
10      for each vertex v adjacent to G[u]:
11          w = weight between u and v
12          // RELAX
13          if distance[u] + w < distance[v]:
14              distance[v] = distance[u] + w
15              Add the pair (distance[v], v) to Q
```

Time complexity and how it works:

Initialization of distances may require $\boldsymbol{O}(V)$.
Each push and pop to Q requires $\boldsymbol{O}(\log V)$ since Q is implemented as a min heap. To maintain the heap property. Therefore, because of a priority queue implementation, the time complexity of the Algorithm is $\boldsymbol{O}((V + E) \log V)$. Since each edge is relaxed at most once, and the priority queue operations dominate the runtime.

At the relaxation step found in the block RELAX, the condition ensures that a shorter path to node v is found only when we found a better path u to v through an edge with smaller weight w, and thus finding a better path thanks to the greedy principle of the Algorithm.

For the small optimization done at lines 8 ~ 9, when a shorter path to u is found, its updated distance is pushed into the priority queue. However, the outdated distances may still exist in the queue. If the distance d currently being processed from the priority queue is greater than the shortest distance[u] it means that d is outdated.

Problem 3

17-2 Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil lg(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$ We have k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i \cdot 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

The data structure contains only positive integers, and duplicates are not allowed.

The first line inputs a number n, representing how many pieces of data are in the data structure.

The second line inputs n numbers, representing the values stored in the data structure.

The third line inputs a number m, representing how many operation commands are given.

The following m lines contain operation commands:

s represents search,

d represents delete,

i represents insert.

The number following the command letter represents the target of the operation.

Flatten the array and sort is not allowed!

Input Format:

```
7
0 1 2 3 4 5 6
6
s 5
d 2
i 7
s 25
d 36
i 3
```

Output Format:

```
Found
Delete Success
Insert Success
Not Found
Delete Failed
Insert Failed
```

Solution proposed:

Simple representation of the data structure:

```
Structure LAYER
A vector as underlying structure
A vector of flags indicating which slots are logically removed.
ALIVE-COUNT()      // Check how many elements in the layer are not deleted.
```

```
Class DS-LAYERS
A vector of the structure LAYER
BUILD(A)        // Builds the layers using the initial values from array A.
SEARCH(X)       // Binary Search
INSERT(X)
REMOVE(X)
```

For each operation inside the DS-LAYERS Class

```
BUILD(A)
1   for each value in A:
2         INSERT(value)
```

```
SEARCH(x)
// Check each non-empty layer using binary search.
1   for each layer_i up to k:
2         if layer_i is not empty:
3         Left = 0
4         Right = layer_i.size
5         while left ≤ right:
6               middle = (Left + Right) / 2
7               if layer_i[middle] == x:
8                     if layer_i is not deleted:   // From the Flag in our structure.
9                           return (i, middle)     // from layer_i and middle the index array
10                    else:
11                          break the while loop  // Found, but the layer is marked as deleted.
12              else if layer_i[middle] < x:
13                    Left = middle + 1
14              else:
15                    Right = middle - 1
16  return (-1, -1)      // "Not Found", return invalid indices.
```

```
INSERT(x)
1   if SEARCH(x) is not equal to -1:
2       return false            // Duplicated element, do not insert
3   Let Carry be an array containing the element x   // Similar to a binomial array.
4   for each layer_i up to k:
5       if layer_i is empty:    // Insert the new element into the layer.
6           layer_i = Carry
7           Set the Flags to false
8           return true
9       else:
10          mergedArray = MERGE-SORTED-ARRAYS(layer_i, Carry) // Merge the
            arrays because the layer is full.
11      layer_i.clear   // Clear the contents of this layer.
12      Carry = move(mergedArray)
```

The merged array must have size $2^{(k+1)}$ if it merges exactly, or possibly less if we started with fewer elements. But in this approach, we expect exactly $2^{(k+1)}$ if the layer had 2^k elements and carry had 2^k as well.

Each Layer contains:

- array: sorted array of size 2^k (if nonempty)
- deleted: same size, marking which slots are logically removed

A visualization of the structure following the offer testcase is shown as below:

$layer_0$:	**6**			
$layer_1$:	**4**	**5**		
$layer_2$:	**0**	**1**	**2**	**3**
$layer_3$:	*empty*			
…	*empty*			
$layer_k$	*empty*			

After all the insertion of our initial values. In my code you fill find that I thought my solution for a maximum number of k =16 layers, which is 2^{16} elements. However, in case of need of more layers change the value of the static constant variable MAX_LAYERS inside the DS_LAYERS class.

```
REMOVE(x)
1   returned pair as (layer_k, pos) from SEARCH(X)
2   Index = layer_k
3   Position = pos
4   if Index == −1:
5       return false      // Element x doesn't exist
6   Alive = get the not deleted elements from layer_Index
7   Capacity = layer_Index.size
8   if 2 · live < capacity:
9       Let LiveElements be an array with the not deleted elements in layer_Index
10      Clear up layer_Index
11      MERGE-UP(Index + 1, LiveElements)
12  return true
```

- Utility functions -

```
MERGE-SORTED-ARRAYS(layer_i, Carry)
1   Let Alive be array with the elements of layer_i that hasn't been delete it yet
2   for value in layer_i that are not flagged as deleted
3       Add value to Alive array
4   Let Result be the array resulting after merging Alive and Carry
```

```
MERGE-UP(L, Carry)
1   for layer_i from L up to k:
2       if layer_i is empty:   // We insert our new element into the layer
3           layer_i = Carry
4           Set the Flags to false
5           Return
6       else:
7           mergedArray = MERGE-SORTED-ARRAYS(layer_i, Carry)
8           layer_i.clear      // Remove the contents of this layer
9           Carry = move(mergedArray)
```

Time complexity analysis:

- SEARCH(X): Each layer's binary search is $\boldsymbol{O}(log\ 2^k) = \boldsymbol{O}(k)$ In the worst case with $log\ n$ layers, searching running time is $\boldsymbol{O}(log^2\ n)$.
- INSERTION(X): From an Amortized sense, each insertion can cause merges up through multiple layers, but each element only moves up a constant number of times. Based on this knowledge then we have an amortized time of $\boldsymbol{O}(log(n))$.
- REMOVE(X): If a layer has more than half deleted, we physically rebuild it once in a while. By its amortized time again $\boldsymbol{O}(log(n))$.