CS29003 Algorithms Laboratory

Assignment No: 6

Last date of submission: 05–March–2019

Heaps and Priority Queues

Come to Foobarland Highway again. The long straight highway is serviced by n mobile-phone towers. The i-th tower has a range $I_i = [l_i, r_i]$ with $l_i < r_i$, for $i = 0, 1, 2, \ldots, n-1$. Let $L = \min_i(l_i)$ and $R = \max_i(r_i)$. The highway stretches from L to R. Assume that each l_i, r_i is an integer. Assume also that the union of the tower ranges $[l_i, r_i]$ is the entire interval $[L, R]$, that is, each point in $[L, R]$ is covered by at least one tower.

The communication minister of Foobarland notices that even if some towers are made non-operational, the remaining functional towers continue to cover the entire interval $[L, R]$. In order to minimize the maintenance cost, he wants to pick a minimum number of towers which cover the entire $[L, R]$. This minimization problem can be solved by a greedy algorithm. An interval $[l_i, r_i]$ is called *active* at a point $x \in [L, R]$ if $l_i \leqslant x < r_i$, that is, if the interval starts at or to the left of x and ends strictly to the right of x. The algorithm described below chooses a sequence of active intervals. Since $O(n)$ intervals are to be added to the output, and Steps II(a) and II(b) can run in $O(n)$ time, this algorithm takes $O(n^2)$ running time.

I. Set $x = L$.

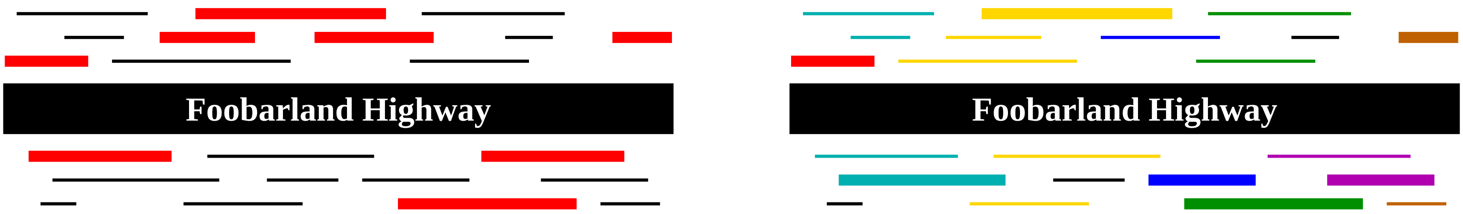
II. While $x < R$, repeat:

  (a) Find all the intervals active at x.

  (b) Choose an active interval $I = [l, r]$ with the largest right endpoint r.

  (c) Output I as an interval chosen, and set $x = r$.

The figure below illustrates the working of this algorithm. The left part shows an arbitrary minimal (but not minimum) cover, whereas the right side shows the greedy (and so minimum) cover.



This assignment deals with an $O(n \log n)$-time $O(n)$-space implementation of this algorithm. Let E denote the array of $2n$ endpoints of the given intervals, sorted in the ascending order (so $E[0] = L$ and $E[2n-1] = R$). Also, maintain a max-priority queue Q of active intervals. The heap ordering in Q is with respect to the right endpoints of the active intervals. Only at the points of E, the queue Q changes. Run the following steps.

1. Let E store the $2n$ endpoints of the given intervals. Each element of E consists of an endpoint e, the number i of the interval of which e is an endpoint, and optionally the information whether e is the left or the right endpoint of the interval I_i. Sort E in the ascending order of the endpoints. After this sorting, denote $E[k] = (e_k, i_k)$ for $k = 0, 1, 2, \ldots, 2n-1$.

2. Output the interval I_{i_0}. Let Q consist only of the interval I_{i_0}. Initialize $x = r_{i_0}$, and $k = 0$.

3. While $x < R$, repeat: /* Interval $[L, x]$ is so far covered */
    Increment k. /* Handle the next endpoint */
    If e_k is the left endpoint of the interval I_{i_k}, then:
        Insert I_{i_k} to Q. /* I_{i_k} now becomes active. */
    else:
        Delete I_{i_k} from Q. /* I_{i_k} now becomes inactive */
        If $e_k = x$, then: /* If the last interval chosen for the output is deleted */
            Choose from Q the interval I_j with the maximum right endpoint.
            Output the interval I_j, and set $x = r_j$.

Part 1: Managing E

The algorithm should run in $O(n \log n)$ time, so write a function to merge sort E. You may also implement E as a min-priority queue with only *heapify*, *makeheap* and *deletemin* (*insert* is not needed for E).

Part 2: Managing Q

Implementing Q as a max-heap is somewhat involved. It is initialized to a single-element Q (so *makeheap* is not needed). It should also support *insert*. The *delete* from Q is, in general, not *deletemax*. The interval deleted from Q is indeed the one with the smallest right endpoint among the active intervals, so this operation is actually a *deletemin* from a max-heap. A minmax-heap is a solution. But since $O(n)$ space is allowed (E already uses this much space), we can take a simpler approach.

Each element of Q stores the endpoints of the interval $[l_i, r_i]$ along with the number i of the interval. We maintain an additional array $idx[0 \ldots n-1]$. If I_i is not an active interval (at some point of time), we should have $idx[i] = -1$. If I_i is active, it resides in Q at some index l, and we should have $idx[i] = l$. Whenever two elements in Q are swapped, the corresponding *idx* entries must also be appropriately modified. More precisely, if the active interval I_i at index l_1 is swapped with the active interval I_j at index l_2 (so l_2 is either the parent or a child of l_1), we should, after the swap, set $idx[i] = l_2$ and $idx[j] = l_1$.

With this additional information, the deletion of I_{i_k} from Q is carried out as follows. Find the index $l = idx[i_k]$ where I_{i_k} resides in Q. If l is the last element of Q, simply decrement the size of Q. Otherwise, copy the last element of Q to index l (and update the *idx* entry of this interval), decrement the size of Q, and move this newly positioned element of Q up the heap so long as necessary (the minimum was deleted, so the value at index l in Q cannot be decreased by the copy).

Write the functions `insertQ` and `deleteQ` to implement the required operations on Q.

Part 3: Find a minimum cover

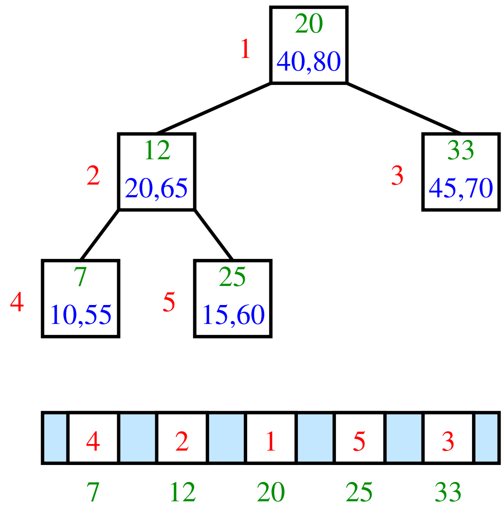
Write a function `mincover` to implement the three-step algorithm described earlier. The data structures E and Q should exist only inside this function (but not in `main()` or globally).

The *main*() function

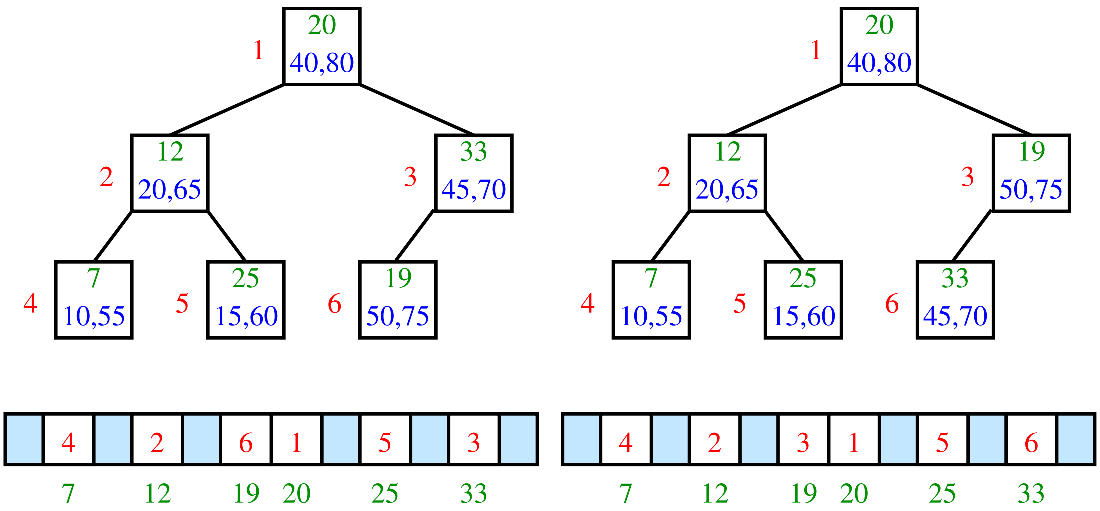
- Read n and the intervals $[l_i, r_i]$ for $i = 0, 1, 2, \ldots, n-1$ from the user.
- The total coverage $[L, R]$ of the intervals can be computed in $O(n)$ time. But for this assignment, you may assume that $\boxed{L = 0 \text{ and } R = 999.}$ Your program should handle $\boxed{n \leqslant 256.}$
- Call `mincover` to compute and print a minimum cover of $[L, R]$.

Notes

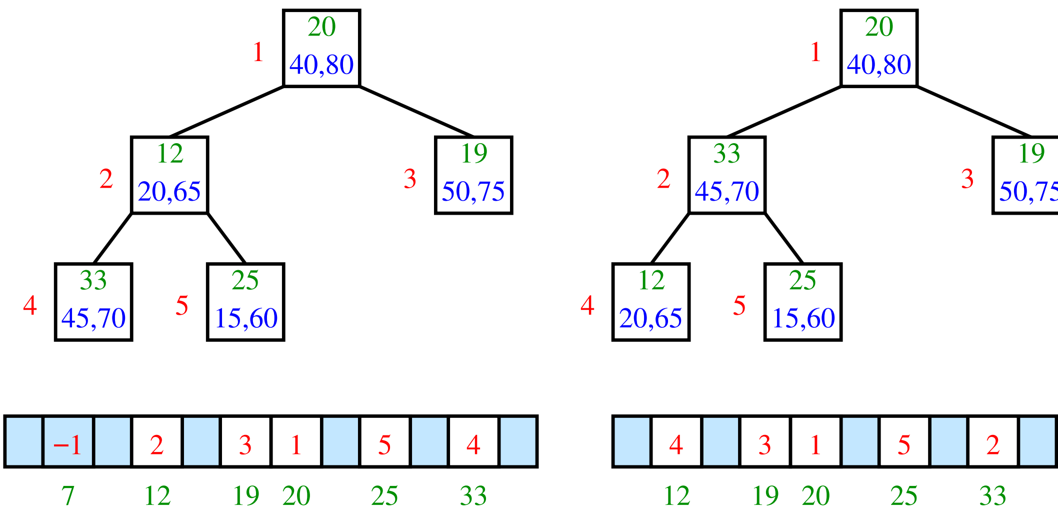
- Convince yourself that this algorithm runs in $O(n \log n)$ time.
- Like merge sort, heap sort is another worst-case $O(n \log n)$-time sorting algorithm. You may heap sort E using a max-heap. But if E is maintained as a min-heap, an explicit sorting of E is not necessary. The management of E illustrates a situation where sorting and using a priority queue offer equivalent benefits. The management of Q is not quite like that.
- Q can also be maintained as a min-heap. If so, each deletion is the natural *deletemin* operation from Q. But then, finding the next interval I_j for output is doing a *findmax* operation in a min-heap, that is, we always need to keep track of the maximum in a min-heap. This is certainly doable, but we promote Q be designed as a max-heap for exposing you to the indexing approach (see the next point).
- This assignment illustrates a situation where we always do *deletemin* from a max-heap. The indexing approach is, however, equally applicable to *arbitrary* deletions. If the value being deleted is larger than the last element in the heap, then a smaller value replaces the deleted value, and the potential violation of heap ordering is to be readjusted by moving the smaller value *down* the tree (as in *heapify*).
- The algorithm can be easily adapted to the situations where endpoints are repeated, and where the input intervals do not cover the entire $[L, R]$ (so your output should maintain the total coverage).



(a) Intervals active at 50



(b) Insertion of the interval [50,75]



(c) Deletion of the interval [10,55]

Sample output

```
n = 50

408 495   479 553   592 657   195 248   832 921   312 364    58 134   755 826
214 297     0  73   131 194    28 117   687 768   520 590   190 289   356 429
727 792   301 400   967 999   484 571   613 707   542 617   853 934   744 805
136 218   362 453   346 412   751 843   642 717    99 153   662 730   876 971
749 820   395 481   898 970   799 869   899 950   862 922   917 993   269 350
812 905   890 977   376 438   714 797   532 630   129 225   828 913   652 723
669 757   223 308

+++ Finding minimum cover
    Added interval   9 [  0, 73]
    Added interval   6 [ 58,134]
    Added interval  45 [129,225]
    Added interval  49 [223,308]
    Added interval  17 [301,400]
    Added interval  33 [395,481]
    Added interval   1 [479,553]
    Added interval  44 [532,630]
    Added interval  20 [613,707]
    Added interval  12 [687,768]
    Added interval  27 [751,843]
    Added interval   4 [832,921]
    Added interval  38 [917,993]
    Added interval  18 [967,999]
    Total number of intervals = 14
```

Submit a single C/C++ source file. Do not use global/static variables.