9 Range Minimum Queries

Suppose we are given an array $A[1\,..\,n]$ of real numbers, and we want to build a data structure that supports the following query:

- MinimumBetween(l, r): Return the minimum value in the subarray $A[l\,..\,r]$

In this lecture I'll describe several *range minimum query* data structures, with different tradeoffs between the size of the data structure and the time to answer queries, ultimately leading to a structure that uses $O(n)$ space and answers queries in $O(1)$ worst-case time. These various data structures are summarized in the following table.

technique	space	query time
lookup table	$O(n^2)$	$O(1)$
binary search	$O(n)$	$O(\log n)$
sparse table	$O(n \log n)$	$O(1)$
indirection	$O(n \log\log n)$	$O(1)$
recursive indirection	$O(n \log^* n)$	$O(\log^* n)$
tetrarosic[1] precomputation	$O(n)$	$O(1)$

9.1 Standard Solutions

Lookup Table

The simplest RMQ data structure is a brute-force $n \times n$ lookup table containing the answers to all possible range-minimum queries. We can construct this table in $O(n^2)$ time as follows:

```
AllRangeMinima($A[1\,..\,n]$):
  for $i \gets 1$ to $n$
      for $j \gets 1$ to $i-1$
          $RMQ[i,j] \gets \infty$
      $RMQ[i,i] \gets A[i]$
      for $j \gets i+1$ to $n$
          $RMQ[i,j] \gets \min\{RMQ[i,j-1],\ A[j]\}$
  return $RMQ[1\,..\,n, 1\,..\,n]$
```

The resulting data structure uses $O(n^2)$ space and answers queries in $O(1)$ time.

Sparse Lookup Table

We can reduce the size of the lookup table to $O(n \log n)$ using the following simple observation: Every range $A[l, r]\,..$ is the union of at most two ranges whose lengths are powers of 2. Specifically, for any indices i and j, we have

$$RMQ[i,j] = \min\left\{RMQ[i, i+2^k-1],\ RMQ[j-2^k-1, j]\right\}$$

[1]From the Greek τετρα- ("four") and ρωσικός ("Russian").

where $k = \lfloor \log_2(j-i+1) \rfloor$. So instead of precomputing the minima of *every* range, we can precompute only the minima of ranges *whose lengths are powers of* 2. The following algorithm builds an array RMQ' such that $RMQ'[i,k] = RMQ'[i, i+2^k-1]$.

```
SPARSERANGEMINIMA($A[1..n]$):
  for $i \gets 1$ to $n$
      $RMQ'[i,0] \gets A[i]$
  $L \gets 1$
  for $\ell \gets 1$ to $\lfloor \log_2 n \rfloor$
      $L \gets 2L$   $\langle\langle L = 2^\ell \rangle\rangle$
      for $i \gets 1$ to $n - L$
          $RMQ'[i,\ell] \gets \min\{RMQ'[i,\ell-1],\ RMQ'[i+L/2,\ell-1]\}$
  return $RMQ'[1..n, 1..\lfloor \log_2 n \rfloor]$
```

The array RMQ' clearly uses $O(n \log n)$ space. With this array in hand, we can now answer any range minimum query in $O(1)$ time[2] using two lookups:

```
MINIMUM($i, j$):
  $k \gets \lfloor \log_2(j-i+1) \rfloor$
  return $\min\{RMQ'(i,k),\ RMQ'(j-2^k+1,k)\}$
```

Tournament Tree

The simplest *linear-space* data structure builds a balanced binary *tournaments* tree T, whose leaves store the values in the sequence in order from left to right. Each node v stores the following information:[3]

- $v.value$: the value of v (only if v is a leaf)
- $v.left$: a pointer to v's left child, if any
- $v.right$: a pointer to v's right child, if any
- $v.first$: the minimum *index* among all leaf descendants of v
- $v.last$: the maximum *index* among all leaf descendants of v
- $v.min$: the minimum *value* among all leaf descendants of v

The *min*, *first*, and *last* fields are defined recursively as follows: If v is a leaf, we have

$$v.first = v.last \qquad \text{and} \qquad v.min = v.value,$$

and otherwise,

$$v.first = v.left.first \qquad v.last = v.right.last$$
$$v.min = \min\{v.left.min,\ v.right.min\}.$$

[2]Careful readers might object that logarithms can't be computed using only $O(1)$ standard arithmetic operations, so why does the first line take only $O(1)$ time? Most modern CPUs can compute $\lfloor \log_2 \ell \rfloor$ for any integer ℓ using either a single `bsr` (*bit scan reverse*) instruction, or a single `clz` or `lzcnt` (*count leading zeros*) instruction and a subtraction. But even if your CPU doesn't have `bsr` or `clz` or `lzcnt` instructions—weird flex, but okay—it is easy to precompute a separate lookup table containing $\lfloor \log_2 \ell \rfloor$, for every integer ℓ from 1 to n, in $O(n)$ time. Similarly, 2^k can be computed in $O(1)$ time using a single *left-shift* instruction.

[3]In fact, we can compute $v.left$ and $v.right$ on the fly if each node stores the size of its subtree. If n is a power of 2, we don't even need to store that!

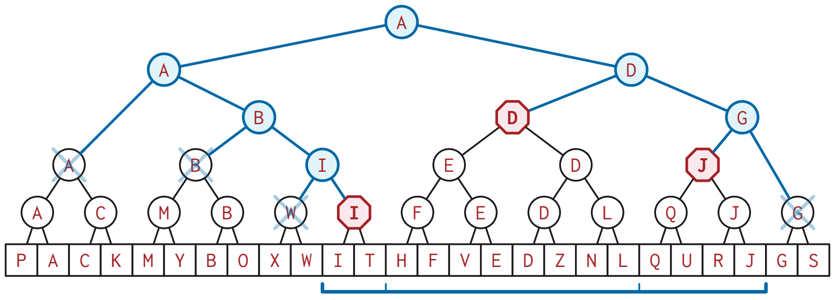
Initializing this data structure in $O(n)$ time is straightforward.

To answer Minimum, we use the following recursive algorithm. The first argument v is a node in the tournament tree; specifically, in the top-level function call, v is the root.

```
Minimum(v, i, j):
  if $i > v.last$ or $j < v.first$
      return $\infty$
  else if $i \le v.first$ and $j \ge v.last$
      return $v.min$
  else
      $lmin \leftarrow$ Minimum($v.left, i, j$)
      $rmin \leftarrow$ Minimum($v.right, i, j$)
      return $\min\{lmin, rmin\}$
```

Minimum(v, i, j) calls itself recursively if and only if $v.first < i \le v.last$ or $v.first \le j < v.last$. At each level of the tree, there is at most one node v that meets each of these conditions. (These are the blue nodes in the figure below.) It follows that the total number of recursive calls is at most $4\log_2 n$; we conclude that Minimum runs in $O(\log n)$ time.

Said differently, the preprocessing algorithm precomputes the minima of $O(n)$ *canonical* ranges, each associated with a node in the tournament tree.[4] The Minimum algorithm recursively partitions the query range $A[i \,..\, j]$ into $O(\log n)$ disjoint *canonical* ranges. The output of Minimum$(\cdot, i, j)$ is the smallest *min* value among these $O(\log n)$ nodes. (These are the red octagonal nodes in the figure below.)



Answering a range-minimum query using a tournament tree

While this data structure does use optimal $O(n)$ space, its design is inextricably linked to a recursive bisection of the data. Beating the $O(\log n)$ query time with linear space using this approach seems unlikely; all our later data structures are variants of lookup tables.

9.2 Indirection

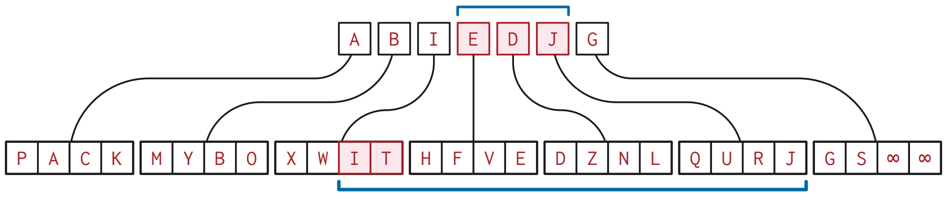
Partition the original array of length n into n/b *blocks*, each of length b. (Pad the array with ∞s if necessary so that n/b is an integer.) Equivalently, reindex the input array A as a two-dimensional array $A[1 \,..\, n/b, 1 \,..\, b]$.

Store a sparse table for each block, and store an array $M[1 \,..\, n/b]$ of row minima. Then we can answer a range-minimum query as follows:

[4] If n is a power of 2, then the length of each canonical range is a power of 2, and the final index of any canonical range is an integer multiple of its length.

```
MINIMUM(A, i, j):
  $\hat{\imath} \gets \lceil i/b \rceil$
  $\hat{\jmath} \gets \lceil j/b \rceil$
  if $\hat{\imath} = \hat{\jmath}$
      return MINIMUM($A[\hat{\imath}, \cdot], i - (\hat{\imath}-1)b, j - (\hat{\imath}-1)b$)    ⟨⟨sparse table⟩⟩
  else
      $left \gets$ MINIMUM($A[\hat{\imath}, \cdot], i - (\hat{\imath}-1)b, b$)    ⟨⟨sparse table⟩⟩
      $mid \gets$ MINIMUM($M, \hat{\imath}, \hat{\jmath}$)    ⟨⟨sparse table⟩⟩
      $right \gets$ MINIMUM($A[\hat{\jmath}, \cdot], 1, j - (\hat{\jmath}-1)b$)    ⟨⟨sparse table⟩⟩
      return min{$left, mid, right$}
```

The query algorithm clearly runs in $O(1)$ time. We need $(n/b) \cdot O(b \log b) = O(n \log b)$ space for the sparse tables for each row, plus $O((n/b)\log(n/b))$ space for the sparse table for the row minima. If we set $b = \lceil \alpha \lg n \rceil$ for some constant α, the row tables use a total of $O(n \log\log n)$ space, and the row-minima table uses $O(n)$ space.



Answering a range-minimum query with one level of indirection

9.3 Recursive Indirection

Instead of immediately falling back to sparse tables, suppose we apply a second level of indirection to the row tables. Then for each of the n/b rows, we need $O(b \log\log b)$ space, and therefore $O(n \log\log b) = O(n \log\log\log n)$ space overall.

More generally, if we apply k levels of indirection, the size of the data structure is $O(kn + n \log^{(k)} n)$, and the query time is $O(k)$. The k-fold logarithm function $\log^{(k)} n$ is defined recursively as follows:

$$\log^{(k)} n = \begin{cases} n & \text{if } k = 0 \\ \log \log^{(k-1)} n & \text{otherwise} \end{cases}$$

Each level of indirection adds another $O(n)$ term to the space bound and two more table look-ups in the query algorithm in the worst case.

For large enough k, the $O(kn)$ time in the space bound is actually larger than the $O(n \log^{(k)} n)$ term. The crossover happens approximately when k is equal to the *iterated logarithm* $\log^* n$; this is the smallest value of k such that $\log^{(k)} n \le 1$. If we use $\log^* n$ levels of indirection, the overall size of our data structure is $O(n \log^* n)$, and we can answer any range-minimum query in $O(\log^* n)$ time.

9.4 Lowest Common Ancestors

Before we go any further, let me first describe a seemingly unrelated problem. Let u and v be two vertices in an arbitrary rooted tree T. The lowest common ancestor $lca_T(u, v)$ of u and v is the deepest node in T that is both an ancestor of u and an ancestor of v. (Recall that an *ancestor* of v is either v itself or an ancestor of v's parent.) The *LCA problem* asks us to preprocess T into a data structure, so that later the lowest-common ancestor of any two vertices of T can be reported quickly.

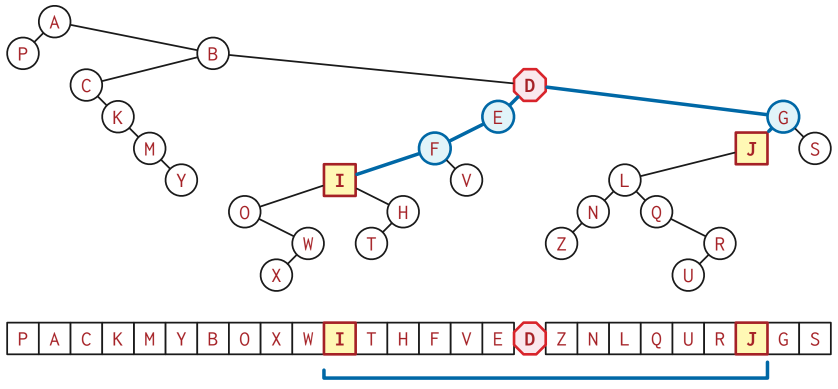
Reducing RMQ to LCA: Cartesian Trees

We can reduce the range-minimum query problem for an array $A[1\,..\,n]$ to the least-common ancestor problem by constructing the *Cartesian tree* of A. A Cartesian tree is a binary tree T with n nodes, each labeled with a unique input value $A[i]$, satisfying the following properties:

- An inorder traversal of T yields the original input sequence $A[1\,..\,n]$.
- The values in T satisfy the minimum heap property: If u is the parent of v, then u's value is smaller than v's value.

Said differently, if we associate a unique *rank* from 1 to n with each node, a Cartesian tree is simultaneously a binary *search* tree with respect to ranks, and a min-heap with respect to the corresponding *values* $A[v.rank]$.

The minimum value in any range $A[i\,..\,j]$ is precisely the value of the least common ancestor of the nodes in T with ranks i and j. Conversely, the least common ancestor of two arbitrary nodes u and v of T is the node with minimum value whose rank is between $u.rank$ and $v.rank$. Thus, if we can preprocess any Cartesian tree to answer LCA queries, we can also preprocess any array of numbers to answer range-minimum queries, with the same asymptotic space and query time bounds.



Reducing range minimum queries to LCA queries in a Cartesian tree.

Reducing LCA to ±1RMQ: Euler Tours

It turns out that the simplest way to preprocess trees for LCA queries is to reduce back to (a special case of) range-minimum queries!

Let T be an arbitrary (not necessarily binary) rooted tree. We use a particular method for "flattening" T into an array called an *Euler tour*. An Euler tour of a rooted tree T is essentially the result of walking around T, starting at the root and keeping your left hand on the wall, recording each node each time your left hand touches it.

Suppose T is represented by storing at each node v a pointer $v.child$ to its first child and a pointer $v.next$ to its next sibling, exactly as we did in the previous lecture for Fibonacci heaps and pairing heaps. Then each of the following algorithms executes an Euler tour of T. (The first algorithm is invoked at the root of T; the second algorithm requires each node to store am additional pointer to its parent.)

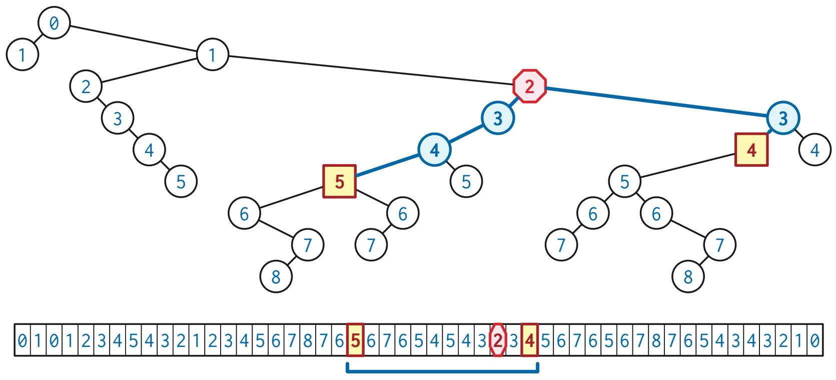
```
EULERTOUR($v$):
    visit $v$
    $w \leftarrow v.child$
    while $w \neq$ NULL
        EULERTOUR($w$)
        visit $v$
```

```
EULERTOUR($T$):
    $v \leftarrow T.root$
    while $v \neq$ NULL
        visit $v$
        if $v.child \neq$ NULL
            $v \leftarrow v.child$
        else if $v.next \neq$ NULL
            $v \leftarrow v.next$
        else
            $v \leftarrow v.parent$
```

Each node v in T appears in the Euler tour of T exactly $\deg(v)+1$ times, so the total length of the Euler tour is $\sum_v(\deg(v)+1) = \sum_v \deg(v) + n = (n-1)+n = 2n-1$. (We can also derive this length by observing that the Euler tour traverses each of the $n-1$ edges of T exactly twice.) It follows that both of our algorithms for computing the Euler tour run in $O(n)$ time.

We can now reduce the LCA problem in T to an RMQ problem as follows:

1. Compute the depth of every node in T using a using a standard breadth-first search.
2. Record an Euler tour of T into a new array $ET[1..2n-1]$. For each node v, record the index $v.index$ of *any* occurrence of v in this array.
3. Build a depth array $D[1..2n-1]$ by defining $D[i] = ET[i].depth$ for each index i.
4. Finally, preprocess the depth array D for range-minimum queries. Vertex $ET[i]$ is the least common ancestor of nodes u and v if and only if $D[i]$ is the smallest depth in the range $D[u.index..v.index]$.



Reducing LCA queries to range minimum queries in an Euler tour of depths

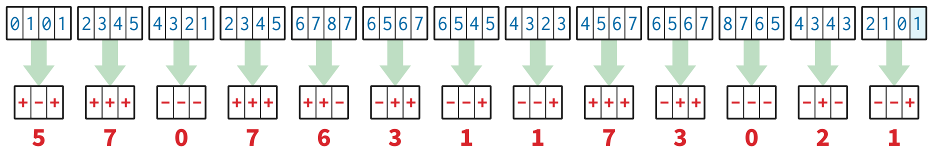
Thus, if we can preprocess any array of numbers to answer range-minimum queries, we can also preprocess any rooted tree to answer LCA queries, with the same asymptotic space and query time bounds. Moreover, by combining this reduction with the previous one, we can reduce the RMQ problem for *arbitrary* arrays to the special case of RMQ *where each array entry is either one greater or one smaller than its neighbors*, with the same performance bounds. Tthis special case of the range-minimum query problem is often called **±1RMQ**.

9.5 Four Russians: Precompute Everything!

The last trick we need to reduce the space to $O(n)$ is one of the simplest examples of the *Four Russians* technique.

Suppose we are given an array $D[1..N]$ where adjacent entries differ by 1, perhaps as output of our earlier reductions to and from the LCA problem. Following our earlier indirection technique, we partition the array into N/b blocks, each of length b, and build a sparse lookup table for the sequence of N/b block minima. But we do *not* build an independent RMQ data structure for each block.

Instead, we observe that two blocks $A[1..b]$ and $B[1..b]$ can use the *same* RMQ data structure if they have the same sequences of differences, that is, $A[i+1]-A[i] = B[i+1]-B[i]$ for all i. Because each difference $A[i+1]-A[i]$ is either -1 or $+1$, there are at most 2^{b-1} equivalence classes of blocks; moreover, we can encode the equivalence class of any block with a $(b-1)$-bit integer, whose $i$ht bit is 1 if and only if $A[i+1]-A[i] = 1$.



Identifying blocks of length b in a $\pm$1RMQ instance by $(b-1)$-bit integers.
In this example, there are 13 blocks, but in only 7 equivalence classes.

Now instead of building an independent RMQ structure for each block, we compute the equivalence class of each block, and then compute an RMQ structure for every equivalence class. If we use naïve lookup tables, the total size of all these RMQ structures is $O(2^b b^2)$. If we set $b = \lfloor \frac{1}{2} \log_2 n \rfloor$, the block RMQ srtuctures use only $O(\sqrt{N} log^2 N) = o(N)$ space, and the sparse table for the block minima uses $O(N)$ space. And we can still answr queries in $O(1)$ time!

The Four Russians technique is more often used to improve the *running time* of certain divide-and-conquer or dynamic-programming algorithms, but the principle is the same. Unlike the log-log-improvements that we get from multiple layers of indirection, quadrimoscovian precomputation leads to significant performance improvements in practice, even using fixed block sizes like $b = 8$ or $b = 16$.