CS166 HW1

Colin Man (colinman@stanford.edu), Kenny Xu (kenxu95@stanford.edu)

April 7, 2016

Problem 1 In order to calculate the largest value of k for which $2^k \leq j - i + 1$, we can create a table and go through each of the indices sequentially from 1 to $n+1$ since those are the only values that $j-i+1$ can take on $(0 \leq i, j \leq n)$. As we go through these indices, we keep a counter for the k value that the numbers correspond to and insert into the map, starting from $k = 0$ for 1 and incrementing k every time we pass a power of 2. We can then retrieve the k for any $j - i + 1$ with an $O(1)$ map lookup.

Problem 2

i Given a 2D array with dimensions m by n we can construct an AMQ by building $min\{m, n\}$ RMQ's along the longer dimension, where each RMQ has the runtime $< O(max\{m, n\})), O(1) >$ (for example, the FischerHeun RMQ). We see that constructing each of these RMQ's takes $O(min\{m, n\} * max\{m, n\}) = O(mn)$. When answering queries, we simply run each RMQ that has a slice of the range, getting us $O(m)$ values. We can then loop over these values to find the minimum. Since querying each of our RMQ's takes $O(1)$ time, and we must query $O(min\{m, n\})$ of them, the total query time is $O(min\{m, n\})$. Hence, we have created a $< O(mn), O(min\{m, n\}) >$-time AMQ structure.

ii We can construct the appropriate AMQ by extending our sparse table RMQ in multiple dimension. For each index in the 2D array, we build a 2D sparse table of minimum values spanning the entire region from that index to the bottom-right hand corner (essentially we exponentially scale the size of our region along both x and y axis). Just as building a 1D sparse table relies on dynamic programming, so does the building of a 2D sparse table. We must build by starting, with every index, from a 1 by 1 to 1 by 2 to 1 by 3, etc. This works for the same reason sparse tables for 1D arrays work. After all horizontal strips have been calculated, we can expand vertically: 2 by 1, 3 by 1... 2 by 2, 3 by 2..etc. As long as we build systematically from smaller to larger regions, our DP solution will work. Since DP of a sparse table takes $O(mlog(m))$ for one dimension and $O(nlog(n))$ for the other, our total preprocessing runtime for this 2D array will be $O(mnlog(m)log(n))$.

For queries, we extend the sparse table idea again. Given an arbitrary query ((i, j), (k, l)), we must pick two regions such that, when they overlap, they cover the distance from i to k exactly, yet also cover the appropriate distance between j and l. Another two regions are needed to fully cover the distance between j and l (as well as from i to k). This means we will need to query four sparse tables and then compare the results for the lowest index, which takes $O(1)$. Therefore, all we need to prove is that finding the bounds for these four regions also takes $O(1)$. We can find the appropriate k in $O(1)$ with $O(mn)$ preprocessing (because its over two dimensions). Therefore we simply use the sparse table algorithm along two dimensions: in the x-direction, the ranges are $[j, j + 2^k - 1]$ and $[d, d - 2^k + 1]$. And in the y-direction, the ranges are $[a, a+2^k-1]$ and $[c, c-2^k+1]$. Calculating the indexes of these four regions therefore takes $O(1)$, so overall lookup will be $O(1)$.

We have thereby proven a data structure for AMQ that runs in $< O(mnlog(m)log(n)), O(1) >$.

Problem 3

i We can construct hybrid structures of depth $k-1$ by breaking them up into block sizes $b_1, b_2, b_3, \ldots, b_{k-1}$ and use a sparse table RMQ on each level to achieve $< O(n \log n), O(1) >$ for that level. The preprocessing time of the hybrid structure is then

$$O(n+p(\frac{n}{b_1})+(\frac{n}{b_1})p(\frac{b_1}{b_2})+(\frac{n}{b_1})(\frac{b_1}{b_2})p(\frac{b_2}{b_3})+\cdots+(\frac{n}{b_{k-2}})p(\frac{b_{k-2}}{b_{k-1}})+(\frac{n}{b_{k-1}})p(b_{k-1}))$$

If we let

$$\begin{aligned} b_1 &= \log n \\ b_2 &= \log b_1 \\ &\cdots \\ b_{k-1} &= \log b_{k-2} \end{aligned}$$

Since $p(x) = x \log x$ using a sparse table RMQ, we can use the trick mentioned in lecture to reduce every term but the last as follows:

$$\begin{aligned} O((\frac{n}{b_{i-1}})p(\frac{b_{i-1}}{b_i})) &= O((\frac{n}{b_{i-1}})\frac{b_{i-1}}{b_i}\log(\frac{b_{i-1}}{b_i})) \\ &= O((\frac{n}{b_{i-1}})\frac{b_{i-1}}{\log b_{i-1}}\log b_{i-1}) \\ &= O((\frac{n}{b_{i-1}})b_{i-1}) \\ &= O(n) \end{aligned}$$

Upon reduction, we have a preprocessing time of:

$$\begin{aligned} O(n+(\frac{n}{b_{k-1}})(b_{k-1})\log b_{k-1}) &= O(n \log b_{k-1}) \\ &= O(n \log\log b_{k-2}) \\ &= O(n \log^{(k-1)} b_1) \\ &= O(n \log^{(k-1)} \log n) \\ &= O(n \log^{(k)} n) \end{aligned}$$

Lookup is performed similar to a two level hybrid query: at each of the k levels, we perform RMQ queries one level deeper on the blocks that contain the two boundary indices and perform RMQ on the blocks between them (at the current level). This gives a query time of $O(k) = O(1)$ since k is constant.

Thus, the time complexity of our hybrid structure is $< O(n \log^{(k)} n), O(1) >$ as required.

ii The increased query time arises from the fact that the query time is based on k as shown in part (i): every level of depth that we add requires another lookup per query. Thus, as k increases, the runtime increases. However, since k is a constant, all the hybrid structures still have a query time of $O(1)$, which does not contradict our result in (i).