1. Describe and analyze an efficient algorithm to compute the length of the longest increasing back-and-forth subsequence of a given array of n red and blue integers.

Solution: Let $A[1..n]$ be the given array of integers. For any index i, let $MaxIBF(i)$ denote the length of the longest increasing back-and-forth subsequence of A that starts with $A[i]$. This function satisfies the following variation on the longest-increasing-subsequence recurrence:

$$MaxIBF(i) = \begin{cases} 1 + \max\left\{MaxIBF(j) \;\middle|\; \boldsymbol{j > i} \text{ and } A[j] > A[i]\right\} & \text{if } A[i] \text{ is } \textbf{red} \\ 1 + \max\left\{MaxIBF(j) \;\middle|\; \boldsymbol{j < i} \text{ and } A[j] > A[i]\right\} & \text{if } A[i] \text{ is } \textbf{blue} \end{cases}$$

(As usual for sets of natural numbers, we assume $\max\varnothing = 0$.) We need to compute $\max\{MaxIBF(i) \mid 1 \le i \le n\}$.

We can memoize this function into a one-dimensional array $MaxIBF[1..n]$. Each entry $MaxIBF[i]$ depends only on entries $MaxIBF[j]$ such that $A[i] < A[j]$. Thus, we can compute a suitable evaluation order by sorting (a copy of) A!

```
WhipMyHair(A[1..n]):
  for i ← 1 to n
      I[i] ← i
      B[i] ← A[i]
  sort B and permute I to match
  best ← 0
  for k ← n down to 1
      i ← I[k]                    ⟨⟨A[i] is the kth smallest value in A⟩⟩
      MaxIBF[i] ← 1
      if A[i] is blue
          for j ← 1 to i − 1
              if A[j] > A[i]
                  MaxIBF[i] ← max{MaxIBF[i], 1 + MaxIBF[j]}
      else ⟨⟨A[i] is red⟩⟩
          for j ← i + 1 to n
              if A[j] > A[i]
                  MaxIBF[i] ← max{MaxIBF[i], 1 + MaxIBF[j]}
      best ← max{best, MaxIBF[i]}
  return best
```

The algorithm clearly runs in $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (+5 extra credit): We can speed up the previous algorithm using a data structure that supports the following operations:

- INSERT(k, x): Insert a new item with search key k and value x; both k and x are positive integers.
- MAXRIGHT(k): Return the largest value among all items with search key greater than k, or 0 if no item has search key greater than k.
- MAXLEFT(k): Return the largest value among all items with key search smaller than k, or 0 if no item has search key smaller than k.

We can build a suitable data structure by modifying a binary search tree that uses rotations to stay balanced, such as an AVL tree, red-black tree, splay tree, or treap. In addition to the usual left and right child pointers, each node in our tree stores a search key, a value, *and the maximum value among its descendants*.

- INSERT(k, x): Invoke the usual insertion algorithm, but whenever we perform a rotation to rebalance the tree, update the maximum-descendant-value of the rotated nodes in $O(1)$ time. At the end, update the maximum-descendant-values of the ancestors of the newly inserted node.
- MAXLEFT can be implemented using the usual binary-search-tree algorithm as follows. The second argument defaults to the root of the tree.

```
MaxLeft(k, v):
  if v = Null
      return 0
  ⟨⟨get the maximum value among proper left descendants⟩⟩
  if v.left = Null
      maxL ← 0
  else
      maxL ← v.left.maxval
  ⟨⟨main binary search⟩⟩
  if k < v.key
      return MaxLeft(k, v.left)
  else if k = v.key
      return maxL
  else ⟨⟨k > v.key⟩⟩
      return max{maxL, v.value, MaxLeft(k, v.right)}
```

- MAXRIGHT is implemented symmetrically to MAXRLEFT.

The analysis of whatever balanced binary search tree we're using implies that each operations takes $O(\log n)$ time.

With this data structure in hand, we can modify our earlier algorithm to run in ***$O(n \log n)$ time*** as follows.

```
WHIPMYHAIR(A[1..n]):
  for i ← 1 to n
      I[i] ← i
      B[i] ← A[i]
  sort B and permute I to match
  best ← 0
  for k ← n down to 1
      i ← I[k]                    ⟨⟨A[i] is the kth smallest value in A⟩⟩
      if A[i] is blue
          MaxIBF[i] ← 1 + MAXLEFT(i)
      else ⟨⟨A[i] is red⟩⟩
          MaxIBF[i] ← 1 + MAXRIGHT(i)
      INSERT(i, MaxIBF[i])
      best ← max{best, MaxIBF[i]}
  return best
```

■

Rubric: 10 points, standard dynamic programming rubric. +5 extra credit for $O(n \log n)$ time. These are not the only correct solutions with these running times.

2. Describe and analyze an algorithm that finds the maximum-area rectangular pattern that appears more than once in a given bitmap. Your input is a two-dimensional array $M[1..n, 1..n]$ of bits; your output is the area of the repeated pattern. (The two copies of the pattern might overlap, but must not actually coincide.)

Solution (8/10): For any indices i, j, i', j' such that $(i, j) \neq (i', j')$, and any positive integers h and w, let $\mathit{Equal?}(i, j, i', j', h, w)$ equal TRUE if the $h \times w$ subarrays

$$M[i..i+h-1, j..j+w-1] \quad \text{and} \quad M[i'..i'+h-1, j'..j'+w-1]$$

are identical, and FALSE otherwise. This function satisfies the following recurrence.

$$\mathit{Equal?}(i, j, i', j', h, w) = \begin{cases} \text{FALSE} & \text{if } i = i' \text{ and } j = j' \\ \text{FALSE} & \text{if } i + h > n \text{ or } i' + h > n \\ \text{FALSE} & \text{if } j + w > n \text{ or } j' + w > n \\ \text{FALSE} & \text{if } M[i, j] \neq M[i', j'] \\ \text{TRUE} & \text{if } h = 1 \text{ and } w = 1 \\ \mathit{Equal?}(i, j+1, i', j'+1, 1, w-1) & \text{if } h = 1 \text{ and } w > 1 \\ \mathit{Equal?}(i, j, i', j', 1, w) \\ \quad \wedge\ \mathit{Equal?}(i+1, j, i'+1, j', h-1, w) & \text{otherwise} \end{cases}$$

Most of the cases deal with boundary issues. The first case ensures that we ignore two copies of the same subarray. The second and third cases ensures that we ignore subarrays that reach past the last row and column of the input array. he fourth case ensures that the top-left corners of both subarrays match.

The remaining cases do the real recursive work. In the base case $h = 1$ and $w = 1$, we return TRUE because we already know that $M[i, j] = M[i', j']$. If $h = 1$ and $w > 1$, we've already compared the first columns/pixels, so we recursively compare the rest of the subarrays. Finally, if $h > 1$, we recursively compare the first rows, and then recursively compare the remaining rows.

We can memoize this recurrence into a five-dimensional array $\mathit{Equal?}[1..n, 0..n, 1..n, 1..n, 1..n]$, which we can fill in the following order. (We can nest the two outermost loops in either order, and the orders of the four inner loops doesn't matter at all.) As we fill *Equal?*, we also compute the largest area seen so far.

```
maxA ← 0
for h ← 1 to n
    for w ← 1 to n
        for all i, j, i', j'        ⟨⟨order doesn't matter⟩⟩
            ⟨⟨— compute Equal?[i, j, i', j', h, w] here —⟩⟩
            if Equal?[i, j, i', j', h, w]
                maxA ← max{maxA, h · w}
```

Computing each entry $Equal?[i,j,i',j',h,w]$ requires only $O(1)$ time (assuming the entries it depends on have already been computed), so the entire algorithm runs in ***$O(n^6)$ time***. ■

Solution (10/10): For any indices i,j,i',j' with $(i,j) \neq (i',j')$ and any positive integer h, let $MaxW(i,j,i',j',h)$ denote the maximum width w such that the $h \times w$ subarrays

$$M[i..i+h-1, j..j+w-1] \quad \text{and} \quad M[i'..i'+h-1, j'..j'+w-1]$$

are identical. This function satisfies the following recurrence.

$$MaxW(i,j,i',j',h) = \begin{cases} -\infty & \text{if } i = i' \text{ and } j = j' \\ 0 & \text{else if } j > n \text{ or } j' > n \\ 0 & \text{else if } M[i,j] \neq M[i',j'] \\ 1 + MaxW(i,j+1,i',j'+1,1) & \text{else if } h = 1 \\ \min \left\{ \begin{matrix} MaxW(i,j,i',j',1), \\ MaxW(i+1,j,i'+1,j',h-1) \end{matrix} \right\} & \text{otherwise} \end{cases}$$

Most of the cases deal with boundary issues. The first case ensures that we ignore two copies of the same subarray. The second case is a base case, which ensures that we don't reach past the rightmost column. The third case ensures that the top-left corners of both subarrays match.

The last two cases do the real recursive work. If $h = 1$, we compare $M[i,j]$ and $M[i',j']$ and then recursively ask for the maximum width starting just to the right of those pixels. If $h > 1$, we recursively compute the maximum width of first row and the maximum width of the remaining rows, and return the smaller of those two widths.

We can memoize this recurrence into a five-dimensional array $MaxW[1..n+1, 1..n+1, 1..n+1, 1..n+1, 1..n]$, which we can fill in the following order. (Except for the outer loop, the nesting order of the loops doesn't matter.) As we fill $MaxW$, we also compute the largest area seen so far.

```
maxA ← 0
for h ← 1 to n
    for all i, j, i', j'        ⟨⟨order doesn't matter⟩⟩
        ⟨⟨— compute MaxW[i, j, i', j', h] here —⟩⟩
        maxA ← max{maxA, MaxW[i, j, i', j', h]}
```

Computing each entry $MaxW[i,j,i',j',h]$ requires only $O(1)$ time (assuming the entries it depends on have already been computed), so the entire algorithm runs in ***$O(n^5)$ time***. ■

Solution (10/10 + 5 extra credit): For any integers Δi and Δj, let $MaxArea(\Delta i, \Delta j)$ be the maximum area among all identical subarrays whose top row indices differ by Δi and whose left column indices differ by Δj:

$$M[\quad i..i+h-1,\quad j..j+w-1\quad],$$
$$M[i+\Delta i..i+\Delta i+h-1,\ j+\Delta j..j+\Delta j+w-1]$$

To answer the original problem, we consider all possible values of Δi and Δj (except $\Delta i = \Delta j = 0$) by brute force.

For the rest of the solution, fix arbitrary values of Δi and Δj. We compute a new bitmap M', where

$$M'[i,j] = 1 \iff M[i,j] = M[i+\Delta i, i+\Delta j].$$

Now every pair of identical $h \times w$ subarrays of M with offset $(\Delta i, \Delta j)$ corresponds to a $h \times w$ subarray of M' containing only 1s. Thus, to compute $MaxArea(\Delta i, \Delta j)$, it suffices to compute ***the maximum-area rectangle in M' that contains only 1s.***

Let $m = n - |\Delta i|$ and $m' = n - |\Delta j|$, and for ease of presentation, assume the new bitmap M' is indexed as $M'[1..m, 1..m']$. For any indices i and h, let $\boldsymbol{H(i,j)}$ denote the number of consecutive 1s in M', starting at $M'[i,j]$ and moving upward. This function obeys the following recurrence:

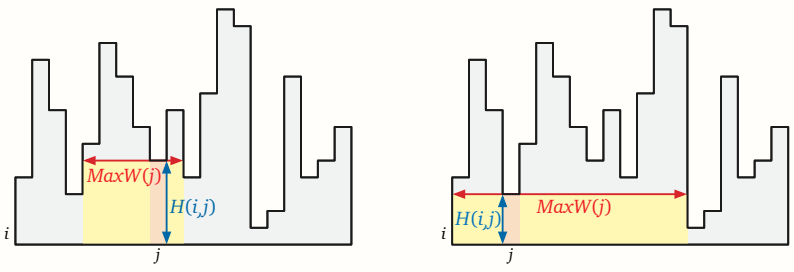
$$H(i,j) = \begin{cases} 0 & \text{if } i > n \text{ or } M'[i,j] = 0 \\ 1 + H(i-1,j) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $H[1..m, 1..m']$, which we can fill from the top row downward in $O(mm')$ time. ***This is the only part of the solution that uses dynamic programming.***

Now for the tricky part. Fix the row index i. For any column index j, let $\boldsymbol{MaxW(j)}$ denote the maximum width of an all-1s rectangle in M' that contains pixel $M'[i,j]$ in its bottom row and has height $H[i,j]$. Said differently:

- Let $j^\flat$ be the largest index such that $1 \le j^\flat \le j$ and $H(i,j^\flat) \ge H(i,j)$.
- Let $j^\sharp$ be the smallest index such that $j \le j^\sharp \le n$ and $H(i,j^\sharp) \ge H(i,j)$

Then $MaxW(j) = j^\sharp - j^\flat + 1$. The following figure shows the definition of $MaxW(j)$ for two different values of j; the gray area contains all contiguous 1s in or directly above row i in M'.



Computing the indices $j^\flat$ and $j^\sharp$ for any given j is almost identical to Problem 1 from Homework 4! Specifically, the "left target" of j is $j^\flat - 1$, the "right target" of j is $j^\sharp + 1$, and these are the closest indices to j with *smaller* heights instead of larger heights. Applying the fastest iterative algorithm from the Homework 4 solutions, we can compute $MaxW(j)$ for *all* j in $O(m')$ time.

It follows that we can compute the largest all-1s rectangle in M' in $O(mm')$ time. Equivalently, for any offset vector $(\Delta i, \Delta j)$, we can compute $MaxArea(\Delta i, \Delta j)$ in $O(mm') = O(n^2)$ time.

Finally, our top-level algorithm considers $O(n^2)$ different offset vectors $(\Delta i, \Delta j)$. We conclude that the largest repeated rectangular pattern in the original bitmap M can be computed in $\boldsymbol{O(n^4)}$ ***time***. ■

Rubric: 10 points: Standard dynamic programming rubric. Correct algorithms with larger or smaller running times are worth different points as follows:

- Max 8 points for $O(n^6)$ time; scale partial credit
- Max 5 points for $O(n^7)$ time; scale partial credit
- Max 3 points for $O(n^8)$ time or more; scale partial credit
- +5 extra credit for $O(n^4)$ time
- +10 extra credit for $O(n^3)$ time — I don't think this is possible, but I've been wrong before.

These are not the only correct algorithms with these running times, or (except for the recurrence for $H(i, j)$) the only correct evaluation orders for these recurrences.

3. Describe and analyze an efficient algorithm to construct an optimal AVL tree for a given set of keys and frequencies. Your input consists of a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches for $A[i]$. Your task is to construct an AVL tree for the given keys such that the total cost of all searches is as small as possible. This is exactly the same cost function that we considered in Thursday's class; the only difference is that the output tree must satisfy the AVL balance constraint.

Solution: Without loss of generality, assume $A[i] = i$ for all i, since only the ranks of the search keys actually matter. In a preprocessing phase, we precompute $F[i,k] = \sum_{j=i}^{k} f[j]$ for all indices i and k, in $O(n^2)$ time, as described in the lecture notes.

For any indices i and j and any integer h, let $MinAVL(i,k,h)$ denote the total cost of the cheapest AVL tree ***of height h*** for the keys $i..k$. We need to compute $\min_h MinAVL(1,n,h)$. AVL trees with a given height can be defined recursively:

- An AVL tree of height -1 is a NULL pointer (that is, nothing).
- An AVL tree of height 0 is a single node.
- For any $h > 0$, an AVL tree of height h is one of the following:
  - A root node, an AVL tree of height $h-1$, and an AVL tree of height $h-1$
  - A root node, an AVL tree of height $h-2$, and an AVL tree of height $h-1$
  - A root node, an AVL tree of height $h-1$, and an AVL tree of height $h-2$

Thus, the *MinAVL* function satisfies the recurrence

$$MinAVL(i,k,h) = F[i,k] + \min \left\{ \begin{array}{l} MinAVL(i,j-1,h-1) + MinAVL(j+1,k,h-1), \\ MinAVL(i,j-1,h-2) + MinAVL(j+1,k,h-1), \\ MinAVL(i,j-1,h-1) + MinAVL(j+1,k,h-2) \end{array} \middle| \; i \le j \le k \right\},$$

when $h > 0$, assuming $\min \varnothing = \infty$, along with the following base cases when $h \le 0$:

$$MinAVL(i,k,h) = \begin{cases} f[i] & \text{if } h = 0 \text{ and } i = k \\ \infty & \text{if } h = 0 \text{ and } i \neq k \\ 0 & \text{if } h = -1 \text{ and } i = k+1 \\ \infty & \text{if } h = -1 \text{ and } i \neq k+1 \end{cases}$$

Every AVL tree with n nodes has height $\Theta(\log n)$, so we only have to consider heights up to $\alpha \log_2 n$ for some known constant α. It follows that we can memoize the *MinAVL* function into a 3d array $MinAVL[1..n+1,\ 0..n,\ -1..\alpha \lg n]$.

We can fill the array by increasing h in the outermost loop, and considering all i and k in arbitrary order in two inner loops. Computing each entry $MinAVL[i,k,h]$ takes $O(n)$ time (looping over all j), so the overall running time is $\boldsymbol{O(n^3 \log n)}$. ■

Rubric: 10 points, standard dynamic programming rubric. This is not the only correct evaluation order. I'm pretty sure this problem can be solved in $O(n^2 \log n)$ time, but I haven't worked through the technical details.