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 $LIB\&FS(i)$ 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:

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

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

We can memoize this function into a one-dimensional array $LIB\&FS[1..n]$. Each entry $LIB\&FS[i]$ depends only on entries $LIB\&FS[j]$ 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]
      LIB&FS[i] ← 1
      if A[i] is blue
          for j ← 1 to i − 1
              if A[j] > A[i]
                  LIB&FS[i] ← max LIB&FS[i], 1 + LIB&FS[j]
      else ⟨⟨A[i] is red⟩⟩
          for j ← i + 1 to n
              if A[j] > A[i]
                  LIB&FS[i] ← max LIB&FS[i], 1 + LIB&FS[j]
      best ← max{best, LIB&FS[i]}
  return best
```

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

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: 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:

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

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

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. (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 i ← n+1 down to 1
        for j ← n+1 down to 1
            for i' ← n+1 down to 1
                for j' ← n+1 down to 1
                    ⟨⟨compute MaxW[i, j, i', j', h]⟩⟩
                    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 $\boldsymbol{O(n^5)}$ ***time***. ■

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 $\mathit{MinAVL}(i,k,h)$ denote the total cost of the cheapest AVL tree ***of height h*** for the keys $i..k$. We need to compute $\min_h \mathit{MinAVL}(1,n,h)$. The *MinAVL* function obeys the recurrence

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

for all $h > 0$, assuming $\min \varnothing = \infty$, along with the base cases

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

Any AVL tree with n nodes has height $O(\log n)$, 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 $\mathit{MinAVL}[1..n+1,\ 0..n,\ -1..\alpha \lg n]$.

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