1. Suppose we are given an array $A[1..n]$ of n distinct integers, which could be positive, negative, or zero, sorted in increasing order.

   (a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

Solution: We can solve the problem in ***$O(\log n)$ time*** using a variant of (or a reduction to) binary search. Here are two pseudocode descriptions of the algorithm, one recursive and one iterative.

```
⟨⟨Find index j such that $i \le j \le k$ and $A[j] = j$⟩⟩
FINDINDEX($i, k$):
    if $i > k$
        return NONE
    $j \gets \lceil (i+k)/2 \rceil$
    if $A[j] = j$
        return $j$
    else if $A[j] > j$
        return FINDINDEX($i, j-1$)
    else
        return FINDINDEX($j+1, k$)
```

```
FINDINDEX($A[1..n]$):
    $lo \gets 1$;  $hi \gets n$
    while $lo \le hi$
        $mid \gets \lceil (lo+hi)/2 \rceil$
        if $A[mid] = mid$
            return $mid$
        else if $A[mid] > mid$
            $hi \gets mid - 1$
        else
            $lo \gets mid + 1$
    return NONE
```

The key observation is that because A is a *sorted* array of *distinct* integers, we have $A[j] \ge A[i] + (j - i)$ for all indices $i < j$. In particular, if $A[i] > i$, then $A[j] > j$ for all $j > i$.

Equivalently, suppose we (implicitly) define a new array $B[1..n]$ by setting $B[i] = A[i] - i$ for all i. Then the elements of B are sorted in *non-decreasing* order (but they are not necessarily distinct), and we are looking for an index i such that $B[i] = 0$. ■

Rubric: 8 points max. For an explicit algorithm: 1 for binary search idea + 1 for base case + 4 for recursive cases + 2 for time analysis. −1 for each off-by-one error. −1 for returning TRUE/FALSE instead of index. −1 for stating running time as a recurrence without solving it. A reduction to binary search is worth full credit. Max 3 points for a $\Theta(n)$-time algorithm; max 2 points for anything slower; scale partial credit.

   (b) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

Solution: If $A[1] = 1$, we can clearly return 1 immediately. On the other hand, if $A[1] > 1$ then $A[i] > i$ for all i, so we can return NONE immediately. So we can solve this problem in ***$O(1)$ time***! ■

Rubric: 2 points: 1 for algorithm + 1 for running time

2. Let G be a directed **acyclic** graph, in which every edge $e \in E$ has a weight $w(e)$, which could be positive, negative, or zero. The *alternating length* of any path $P = v_0 \to v_1 \to v_2 \to \cdots \to v_\ell$ in G is defined as

$$AltLen(P) = \sum_{i=0}^{\ell-1} (-1)^i \cdot w(v_i \to v_{i+1}).$$

Describe an algorithm to find a path from s to t with the largest alternating length, given the graph G, the edge weights $w(e)$, and vertices s and t as input.

Solution (dynamic programming, from the start): Let $G = (V, E)$ be the input dag, with edge weights $w(e)$. We define two functions over the vertices of G:

- $MaxAL(u)$ is the *maximum* alternating length among all paths from u to t.
- $MinAL(u)$ is the *minimum* alternating length among all paths from u to t.

We need to compute $MaxAL(s)$. These two functions satisfy the following mutual recurrences:

$$MaxAL(u) = \begin{cases} 0 & \text{if } u = t \\ -\infty & \text{if } u \neq t \text{ is a sink} \\ \max\left\{ w(u \to v) - MinAL(v) \;\middle|\; u \to v \in E \right\} & \text{otherwise} \end{cases}$$

$$MinAL(u) = \begin{cases} 0 & \text{if } u = t \\ +\infty & \text{if } u \neq t \text{ is a sink} \\ \min\left\{ w(u \to v) - MaxAL(v) \;\middle|\; u \to v \in E \right\} & \text{otherwise} \end{cases}$$

We can memoize these functions into two new fields $u.minAL$ and $u.maxAL$ of each vertex $u \in V$. We compute all function values using a single loop over the vertices u in reverse topological order (or equivalently, depth-first postorder); in each iteration of the loop we compute both functions of u.

The resulting dynamic programming algorithm runs in ***$O(V+E)$ time***. ■

Solution (dynamic programming, from the end): Let $G = (V, E)$ be the input dag, with edge weights $w(e)$.

For any vertex $v \in V$ and any sign $\sigma \in \{+1, -1\}$, let $MaxAL(v, \sigma)$ denote the maximum alternating length among all paths from s to v where the number of edges is even if $\sigma = -1$ and odd if $\sigma = +1$. Equivalently, $MaxAL(v, \sigma)$ is the maximum alternating length among all paths P from s to v where the last edge $u \to v$ in P contributes $\sigma \cdot w(u \to v)$ to the alternating length of P.

We need to compute $\max\{MaxAL(t, +1), MaxAL(t, -1)\}$.

Thus function obeys the following recurrence:

$$MaxAL(v,\sigma) = \begin{cases} 0 & \text{if } v = s \text{ and } \sigma = -1 \\ -\infty & \text{if } v = s \text{ and } \sigma = +1 \\ -\infty & \text{if } v \neq s \text{ is a source} \\ \max\left\{ MaxAL(u,-\sigma) + \sigma \cdot w(u\to v) \;\middle|\; u\to v \in E \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into two new fields of each vertex in V. We compute all function values using a single loop over the vertices v in topological order; in each iteration of the loop we compute both functions of v.

The resulting dynamic programming algorithm runs in $\boldsymbol{O(V+E)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct DP solutions.

Solution (graph reduction): Given a dag $G = (V,E)$ and edge weights $w(e)$, we construct a new dag $G' = (V', E')$ and edge weights $w'(e')$ as follows:

- $V' = V \times \{+,-\}$. I'll write v^+ and v^- as shorthand for $(v,+)$ and $(v,-)$.
- $E' = \left\{ u^+\to v^-, u^-\to v^+ \;\middle|\; u\to v \in E \right\}$
- For each edge $u\to v \in E$, let $w'(u^+\to v^-) = w(u\to v)$ and $w'(u^-\to v^+) = -w(u\to v)$.

For any topological order $v_1, v_2, \ldots, v_n$ for G, the permutation $v_1^+, v_1^-, v_2^+, v_2^-, \ldots, v_n^+, v_n^-$ is a topological order for G', so G' is in fact a dag.

Let $length'(P') = \sum_{e' \in P'} w'(e')$ denote the length of any path P' in G'.

For any path $P = v_0\to v_1\to v_2\to\cdots\to v_\ell$ in G, there is a corresponding path $P' = v_0^+\to v_1^-\to v_2^+\to\cdots\to v_\ell^\pm$, always starting with a positive vertex, such that $length'(P') = AltLen(P)$. Conversely, every path P' in G' that starts with a positive vertex projects to a path P in G such that $AltLen(P) = length'(P')$.

Thus, finding the maximum-$AltLen$ path in G from s to t is equivalent to finding the maximum-$length'$ path in G' from s^+ to either t^+ or t^-. We can find these longest paths in G', using the LLP algorithm described in class and in the textbook, in $O(V' + E') = \boldsymbol{O(V+E)}$ ***time***. ■

Rubric: Also worth 10 points. This is not the only correct graph-reduction solution.

3. (a) Describe and analyze an efficient algorithm to compute the number of inversions in a given boolean array $B[1..n]$.

Solution: The following algorithm runs in $O(n)$ time. We scan through the array, maintaining two counters; whenever we encounter a FALSE, we add the number of TRUEs seen so far to the inversion counter.

```
CountInversions($B[1..n]$):
    $invs \gets 0$
    $trues \gets 0$
    for $i \gets 1$ to $n$
        if $B[i] = $ True
            $trues \gets trues + 1$
        else
            $invs \gets invs + trues$
    return $invs$
```

■

Rubric: 3 points. A correct $O(n \log n)$-time algorithm is worth 2 points; a correct $O(n^2)$-time algorithm is worth 1 point.

(b) Describe and analyze an efficient algorithm to compute, for every integer $1 \le \ell \le n-1$, the number of inversions of length ℓ in a given boolean array $B[1..n]$.

Solution: The number of inversions of length ℓ is given by the formula

$$\sum_{j-i=\ell} [B[i] = \textsc{True}] \cdot [B[j] = \textsc{False}]$$

This is essentially the convolution of B with the bitwise negation of the reversal of B. The following algorithm runs in $\boldsymbol{O(n \log n)}$ ***time*** if we use the FFT algorithm to compute convolutions.

```
CountInversionLengths($B[1..n]$):
    for $i \gets 0$ to $n-1$
        if $B[i] = $ True
            $X[i-1] \gets 1$
            $Y[n-i] \gets 0$
        else
            $X[i-1] \gets 0$
            $Y[n-i] \gets 1$
    $Z \gets$ Convolution($X[0..n-1], Y[0..n-1]$)
    for $\ell \gets 1$ to $n-1$
        $I[\ell] \gets Z[n-1+\ell]$
    return $I[1..n-1]$
```

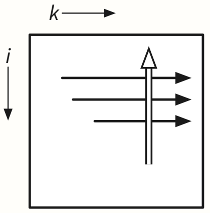
■

Rubric: 7 points = 2 for attempting to use FFTs + 3 for other algorithmic details + 2 for time analysis. A correct $O(n^2)$-time algorithm is worth 4 points.

4. Describe and analyze an algorithm to compute, given a sequence of integers separated by @ signs, the ***smallest*** possible value the expression can take by adding parentheses. Your input is an array $A[1\,..\,n]$ listing the sequence of integers.

Solution: Let $A[1\,..\,n]$ be the input array. For any indices $i \le k$, let $\mathit{MinAve}(i,k)$ denote the largest possible value that can be obtained from the interval $A[i\,..\,k]$ by adding parentheses. We need to compute $\mathit{MinAve}(1,n)$. This function satisfies the following recurrence:

$$\mathit{MinAve}(i,k) = \begin{cases} A[i] & \text{if } i = k \\ \min\left\{\mathit{MinAve}(i,j) \mathbin{@} \mathit{MinAve}(j+1,k) \;\middle|\; i \le j < k\right\} & \text{otherwise} \end{cases}$$



We memoize using two nested loops, one decreasing i and the other increasing k. (It doesn't matter which of these is the inner loop and which is the outer loop.) Each entry $\mathit{MinAve}[i,k]$ in our memoization array takes $O(n)$ time to compute, so the resulting dynamic programming algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming rubric. This is not the only correct evaluation order. This is the fastest algorithm Jeff knows for this problem.

Non-solution: Consider the following greedy algorithm: Merge the adjacent pair of numbers with the smallest average (breaking ties arbitrarily), replace them with their average, and recurse. For example:

$$\begin{gathered}
\underline{8 \mathbin{@} 6} \mathbin{@} 7 \mathbin{@} 5 \mathbin{@} 3 \mathbin{@} 0 \mathbin{@} 9 \\
\underline{7 \mathbin{@} 7} \mathbin{@} 5 \mathbin{@} 3 \mathbin{@} 0 \mathbin{@} 9 \\
\underline{7 \mathbin{@} 5} \mathbin{@} 3 \mathbin{@} 0 \mathbin{@} 9 \\
6 \mathbin{@} 3 \mathbin{@} \underline{0 \mathbin{@} 9} \\
\underline{6 \mathbin{@} 3} \mathbin{@} 4.5 \\
\underline{4.5 \mathbin{@} 4.5} \\
4.5
\end{gathered}$$

With the right data structures, this algorithm can be implemented to run in $O(n \log n)$ time; the only real bottleneck is maintaining a priority queue of adjacent pairs.

Unfortunately, this greedy algorithm does ***not*** always compute the optimal expression. Consider the input $2 \mathbin{@} 5 \mathbin{@} 0 \mathbin{@} 6$. The greedy algorithm outputs $(2 \mathbin{@} 5) \mathbin{@} (0 \mathbin{@} 6) = 3.25$, but the optimal expression is $2 \mathbin{@} (5 \mathbin{@} (0 \mathbin{@} 6)) = 3$. ♣