1. Describe an algorithm to find the largest possible number of *good* districts in a legal partition. In a legal partition, every district has between k and $2k$ residents; a district is good if a strict majority of its residents previously voted for Oceania. Your input consists of the integer k and a boolean array $GoodVote[1..n]$ indicating which residents previously voted for Oceania (TRUE) or Eurasia (FALSE). You can assume that a legal partition exists. Analyze the running time of your algorithm in terms of the parameters n and k.

Solution: Let $MaxGood(i)$ denote the maximum number of good districts in a legal partition of voters i through n. We need to compute $MaxGood(1)$. This function satisfies the following recurrence:

$$MaxGood(i) = \begin{cases} 0 & \text{if } i > n \\ -\infty & \text{if } n - i + 1 < k \\ \max\left\{ \begin{array}{c|c} GoodDistrict(i,j) & k \le j - i + 1 \le 2k \\ \quad + MaxGood(j+1) & j \le n \end{array} \right\} & \text{otherwise} \end{cases}$$

Here $GoodDistrict(i, j) = 1$ if a majority of voters i through j voted for Oceania, and 0 otherwise. We can compute this value in $O(j - i + 1) = O(k)$ time by brute force.

We can memoize this function into a one-dimensional array $MaxGood[1..n]$, which we can fill from right to left (decreasing i). For each index i, we need to compute $GoodDistrict(i, j)$ for $k + 1$ different values of j. Each call to $GoodDistrict(i, j)$ takes $O(k)$ time, so the total time to compute each $MaxGood[i]$ is $O(k^2)$. We conclude that our algorithm runs in $\boldsymbol{O(nk^2)}$ ***time***.

We can speed up this algorithm by pre-computing some information. Let $NumGood(i)$ denote the number of good voters among voters 1 through i. This function satisfies the following recurrence:

$$NumGood(i) = \begin{cases} 0 & \text{if } i = 0 \\ GoodVote[i] + NumGood(i-1) & \text{otherwise} \end{cases}$$

We can memoize this function into an arrays $NumGood[1..n]$, which we can fill in reverse index order in $O(n)$ time.

Once this array has been filled, we can compute $GoodDistrict(i, j)$ for any i and j in $O(1)$ time as follows:

$$GoodDistrict(i, j) \;=\; \big[(NumGood[j] - NumGood[i-1]) > (j - i + 1)/2\big]$$

(Here $j - i + 1$ is the number of voters, and $NumGood[j] - NumGood[i-1]$ is the number of *good* voters, among voters i through j.) With this optimization, each value $MaxGood[i]$ can be computed in $O(k)$ time, so the overall algorithm runs in $\boldsymbol{O(nk)}$ ***time***. ■

Rubric: 10 points: standard dynamic programming rubric. 8 points for an $O(nk^2)$-time algorithm; scale partial credit. This is not the only correct solution.

2. Suppose we want to split an array $A[1..n]$ of integers into k contiguous intervals that partition the sum of the values as evenly as possible. Specifically, define the *cost* of such a partition as the maximum, over all k intervals, of the sum of the values in that interval; our goal is to minimize this cost. Describe and analyze an algorithm to compute the minimum cost of a partition of A into k intervals, given the array A and the integer k as input.

Solution: We define three functions:

- $\mathit{PrefSum}(i)$ is the sum of all elements of the prefix $A[1..i]$.
- $\mathit{Sum}(i,j)$ is the sum of all elements of the interval $A[i..j]$.
- $\mathit{MinCost}(i,\ell)$ is the minimum cost of a partition of $A[i..n]$ into ℓ intervals.

We need to compute $\mathit{MinCost}(1,k)$. These functions satisfy the following recurrences:

$$\mathit{PrefSum}(i) = \begin{cases} 0 & \text{if } i = 0 \\ A[i] + \mathit{PrefSum}(i-1) & \text{otherwise} \end{cases}$$

$$\mathit{Sum}(i,j) = \mathit{PrefSum}(j) - \mathit{PrefSum}(i-1)$$

$$\mathit{MinCost}(i,\ell) = \begin{cases} \mathit{Sum}(i,n) & \text{if } k = 1 \\ \min\left\{ \max\left\{ \begin{array}{c} \mathit{Sum}(i,j), \\ \mathit{MinCost}(j+1,\ell-1) \end{array} \right\} \;\middle|\; i \le j \le n \right\} & \text{otherwise} \end{cases}$$

We can memoize $\mathit{PrefSum}$ into an array $\mathit{PrefSum}[0..n]$, which we can fill from left to right in $O(n)$ time. We don't need to memoize Sum. Finally, we can memoize $\mathit{MinCost}$ into an array $\mathit{MinCost}[1..n, 0..k]$, which we can fill in standard column-major order: increasing ℓ in the outer loop, and considering i in any order in the inner loop.

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

This solution assumes that partitions are allowed to contain empty intervals; without loss of generality, all the empty intervals in any partition are at the end of the array. To forbid empty intervals, modify the base cases as follows:

$$\mathit{MinCost}(i,\ell) = \begin{cases} \infty & \text{if } i > n \\ \mathit{Sum}(i,n) & \text{if } i \le n \text{ and } \ell = 1 \\ \min\left\{ \max\left\{ \begin{array}{c} \mathit{Sum}(i,j), \\ \mathit{MinCost}(j+1,\ell-1) \end{array} \right\} \;\middle|\; i \le j \le n \right\} & \text{otherwise} \end{cases}$$

Solution: For any integers i and ℓ, let $\mathit{Min\$}(i,\ell)$ denote the minimum cost of a partition of $A[i\,..\,n]$ into ℓ intervals. We need to compute $\mathit{Min\$}(1,k)$. This function satisfies the following recurrences:

$$\mathit{Min\$}(i,\ell) = \begin{cases} -\infty & \text{if } \ell = 0 \\ 0 & \text{if } i > n \text{ and } \ell > 0 \\ \min\left\{ \max\left\{ \begin{array}{c} \sum_{h=i}^{j} A[h] \\ \mathit{Min\$}(j+1,\ell-1) \end{array} \right\} \;\middle|\; i \le j \le n \right\} & \text{otherwise} \end{cases}$$

In particular, $\mathit{Min\$}(i,0) = -\infty$ because the maximum of the empty set (in this case, the set containing zero interval sums) is $-\infty$, and $\mathit{Min\$}(n+1,\ell) = 0$ when $\ell > 0$ because the only legal partition of the empty sequence consists of empty intervals, each of which sums to 0.

We can memoize $\mathit{Min\$}$ into an array $\mathit{Min\$}[1\,..\,n+1, 0\,..\,k]$, which we can fill row-by-row bottom up in the outer loop, filling each row in arbitrary order in the inner loop. We can compute each entry $\mathit{Min\$}[i,\ell]$ in $O(n^2)$ time by looping over j and h. Thus, the entire algorithm runs in $\boldsymbol{O(n^3k)}$ ***time***.

However, we can speed up this algorithm with some precomputation. For any index j, let $\mathit{PrefSum}(j)$ denote the prefix sum $\sum_{h=1}^{j} A[h]$. This function satisfies the recurrence

$$\mathit{PrefSum}(j) = \begin{cases} 0 & \text{if } j = 0 \\ A[j] + \mathit{PrefSum}(j-1) & \text{otherwise} \end{cases}$$

We can memoize this function into an array $\mathit{PrefSum}[0\,..\,n]$, which we can fill from left to right in $O(n)$ time. Then we can evaluate any sum $\sum_{h=i}^{j} A[h]$ in $O(1)$ time, because

$$\sum_{h=i}^{j} A[h] \;=\; \mathit{PrefSum}[j] - \mathit{PrefSum}[i-1].$$

With this optimization, our algorithm runs in $\boldsymbol{O(n^2k)}$ ***time***. ■

Rubric: 10 points =

- \+ 8 for $O(n^3k)$-time dynamic programming algorithm (standard DP rubric, scaled)
- \+ 2 for $O(n^2k)$-time algorithm, either by memoizing *Sum* (either directly in $O(n^2)$ time or via prefix sums in $O(n)$ time as above), or by computing $\mathit{Sum}(i,j)$ on the fly in the inner loop for *MinCost*.

We don't care whether you allow or forbid empty intervals, as long as you're consistent.

3. A sequence of integers is *mostly odd* if strictly more than half of its elements are odd. Describe an algorithm that computes the length of the longest *mostly odd* increasing subsequence of a given array $A[1..n]$ of integers. (You can assume that A has at least one mostly-odd increasing subsequence.)

Solution: To simplify case analysis at the end, we add a sentinel value $A[0] = -\infty$.

For any integers i, j, and k, let $LISO(i, j, k)$ denote the length of the longest increasing subsequence of $A[j..n]$ containing only numbers greater than $A[i]$ and containing exactly k odd elements. (Crucially, $A[i]$ is not included in either the subsequence length or the count of odd elements.) This function satisfies the following recurrence:

$$LISO(i,j,k) = \begin{cases} 0 & \text{if } j > n \text{ and } k = 0 \\ -\infty & \text{if } j > n \text{ and } k > 0 \\ LISO(i, j+1, k) & \text{if } A[i] \geq A[j] \\ \max \begin{Bmatrix} LISO(i, j+1, k) \\ 1 + LISO(j, j+1, k) \end{Bmatrix} & \text{if } A[i] < A[j] \text{ and } A[k] \text{ is even} \\ \max \begin{Bmatrix} LISO(i, j+1, k) \\ 1 + LISO(j, j+1, k-1) \end{Bmatrix} & \text{if } A[i] < A[j] \text{ and } A[k] \text{ is odd} \end{cases}$$

We can memoize this function into a three-dimensional array $LISO[0..n, 1..n, 1..n/2]$, which we can fill with three nested for-loops, decreasing j in the outermost loop and considering i and k in any order in the two inner loops.

Finally, we find and return

$$\max \left\{ LISO[0, 1, k] \;\middle|\; LISO[0, 1, k] \leq 2k - 1 \right\}.$$

The entire algorithm runs in ***$O(n^3)$ time.*** ■

Rubric: 10 points, standard dynamic programming rubric. This is not the only correct solution. I do not know whether this is the fastest algorithm for this problem.

4. The StupidScript language includes a binary operator @ that computes the *average* of its two arguments. Expressions like 3 @ 7 @ 4 that use the @ operator more than once yield different results when they are parenthesized in different ways.

   Given a sequence of integers separated by @ signs, represented as an array $A[1..n]$, we want to find the ***largest possible*** value the expression can take by adding parentheses.

   (a) Prove that the following greedy algorithm is incorrect: Merge the adjacent pair of numbers with the smallest average (breaking ties arbitrarily), replace them with their average, and recurse.

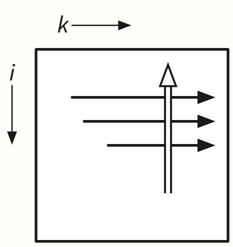
   Solution: Consider the input 7 @ 4 @ 9 @ 3. Tommy's greedy algorithm outputs (7 @ 4) @ (9 @ 3) = 5.75, but 7 @ (4 @ (9 @ 3)) = 6. (In fact, the latter parenthesization is optimal.) ■

   Rubric: 2 points: 1 for bad example + 1 for proof that example is bad

   (b) Describe and analyze a correct algorithm for this problem.

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

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



   Each entry $MaxAve[i,k]$ in our memoization array takes $O(n)$ time to compute, so the resulting dynamic programming algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

   Rubric: 8 points: standard dynamic programming rubric (scaled). Yes, the drawing is enough detail to specify the memoization structure and evaluation order. No, iterative pseudocode is not required. This is not the only correct evaluation order.