1. Describe an algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

Solution: Let $MaxTotal(i)$ denote the maximum score you can achieve dancing songs i through n, assuming you are physically able to dance to song i. We need to compute $MaxTotal(1)$. The function satisfies the following recurrence:

$$MaxTotal(i) = \begin{cases} 0 & \text{if } i > n \\ \max \left\{ \begin{array}{c} MaxTotal(i+1), \\ Score[i] + MaxTotal(i + Wait[i] + 1) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $MaxTotal[1..n]$ (using the value 0 whenever we need to look up $MaxTotal(i)$ for $i > n$.) We can fill this array in constant time per entry by decreasing i in the only loop. ***The resulting algorithm runs in $O(n)$ time.*** ■

Rubric: 10 points: standard dynamic programming rubric.

2. 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:

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

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.

3. 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 $Min\$(i,\ell)$ denote the minimum cost of a partition of $A[i..n]$ into ℓ intervals. We need to compute $Min\$(1,k)$. This function satisfies the following recurrences:

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

In particular, $Min\$(i,0) = -\infty$ because the maximum of the empty set (in this case, the set containing zero interval sums) is $-\infty$, and $Min\$(n+1,\ell) = 0$ 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 $Min\$$ into an array $Min\$[1..n+1, 0..k]$, 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 $Min\$[i,\ell]$ 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 $PrefSum(j)$ denote the prefix sum $\sum_{h=1}^{j} A[h]$. This function satisfies the recurrence

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

We can memoize this function into an array $PrefSum[0..n]$, 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

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

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 $Sum(i,j)$ 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.

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

$$LISE(i,j,k) = \begin{cases} -\infty & \text{if } k < 0 \\ 0 & \text{if } j > n \text{ and } k = 0 \\ -\infty & \text{if } j > n \text{ and } k > 0 \\ LISE(i, j+1, k) & \text{if } A[i] \geq A[j] \\ \max\left\{\begin{array}{c} LISE(i, j+1, k) \\ 1 + LISE(j, j+1, k) \end{array}\right\} & \text{if } A[i] < A[j] \text{ and } A[j] \text{ is odd} \\ \max\left\{\begin{array}{c} LISE(i, j+1, k) \\ 1 + LISE(j, j+1, k-1) \end{array}\right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a three-dimensional array $LISE[0..n, 1..n, 1..n/2]$, 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

$$\max\left\{LISE[0,1,k] \;\middle|\; LISE[0,1,k] \geq 2k+1\right\}.$$

The entire algorithm runs in $\boldsymbol{O(n^3)}$ ***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.