1. (a) Describe an algorithm to compute the maximum amount of joy you can get from taking days off across your first n days of work. You may assume your algorithm is given an array $J[1..n]$ where $J[i]$ is the amount of joy received for taking off the ith workday. Scores could be positive, zero (you have nothing better to do those days), or even negative (the yearly Halloween party is supposed to be a blast!).

Solution: For any integers a, b, i such that $0 \le a < 28$, $a \le b < 28$, and $1 \le i \le n+1$, let $MaxJoy(a, b, i)$ denote the maximum joy achievable from taking off any subset of days i through n under the additional condition that days i through $i+a$ cannot be taken off and at most one day can be taken off between days i through $i+b$. We need to compute $MaxJoy(0, 0, 1)$.

Let $dec(x) := \max 0, x-1$. The function $MaxJoy$ obeys the following recurrence:

$$MaxJoy(a,b,i) = \begin{cases} 0 & \text{if } i > n \\ MaxJoy(a-1, b-1, i+1) & \text{if } i \le n \text{ and } a > 0 \\ \max \left\{ \begin{array}{c} MaxJoy(0, dec(b), i+1), \\ J[i] + MaxJoy(dec(b), 27, i+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a 3D array evaluated in decreasing order of i. Each function evaluation takes $O(1)$ time given its dependencies. Numbers 2 and 28 are constants, so there are $O(n)$ subproblems to solve. ***The algorithm runs in $O(n)$ time.*** ■

Rubric: 5 points: standard dynamic programming rubric. A correct $O(n^2)$ time algorithm is worth 4 points. A correct $O(n^3)$ time algorithm is worth 3.5 points.

(b) Describe an algorithm to compute the maximum amount of joy you can get from taking days off across the first n days of work where you can take off at most ℓ days in any set of k consecutive work days. Give your running time using big-Oh notation in terms of n, ℓ, and k. You may assume you're given an array $J[1..n]$ as before along with the values ℓ and k. Assume ℓ is a **small integer constant**, but k is an integer variable such that $1 \le k \le n$.

Solution: For any integer i such that $1 \le i \le n+1$ and any *sequence* $\langle d_1, d_2, \ldots, d_\ell \rangle$ of integers such that $0 \le d_1,\ d_\ell < k$, and $d_j \le d_{j+1}$ for $1 \le j < \ell$, let $MaxJoy2(\langle d_1, \ldots, d_\ell \rangle, i)$ denote the maximum joy achievable from taking off any subset of days i through n under the additional condition that for any j with $1 \le j \le \ell$, at most j of days i through $i + d_j$ can be taken off. We need to compute $MaxJoy(\bar{0}, 1)$ where $\bar{0}$ denotes the sequence of ℓ 0s.

Let $dec(x) := \max 0, x-1$. The function $MaxJoy2$ obeys the following recurrence:

$$MaxJoy2(\langle d_1, \ldots, d_\ell \rangle, i) = \begin{cases} 0 & \text{if } i > n \\ MaxJoy2(\langle d_1 - 1, \ldots, d_\ell - 1 \rangle, i+1) & \text{if } i \le n \text{ and } d_1 > 0 \\ \max \left\{ \begin{array}{c} MaxJoy2(\langle dec(d_1), \ldots, dec(d_\ell) \rangle, i+1), \\ J[i] + MaxJoy2(\langle dec(d_2), \ldots, k-1 \rangle, i+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a $\ell + 1$-dimentional array evaluated in decreasing order of i. Each function evaluation takes $O(\ell) = O(1)$ time given its dependencies. There are $O(k^\ell n)$ subproblems to solve, so ***the algorithm runs in $O(k^\ell n)$ time.*** ■

Rubric: 5 points: standard dynamic programming rubric. A correct $O(n^{\ell+1})$ time algorithm is worth **2.5** points.

2. Given an array $X[1..k]$ of characters from an arbitrary alphabet, we define a ***run*** as a maximal non-empty contiguous subarray (i.e., substring) whose characters are all equal. The ***flatness*** of X is sum of the squares of the lengths of runs in X.

   Suppose you are given an array $A[1..n]$ of characters over some fixed alphabet Σ. Describe an algorithm to compute the maximum flatness of any *subsequence* of A.

Solution: Let $MaxFlat(i)$ denote the maximum flatness of any subsequence of $A[i..n]$. We need to compute $MaxFlatness(1)$. For a proposition P, we let $[P] = 1$ if P is True and $[P] = 0$ otherwise.

The function *MaxFlat* obeys the following recurrence:

$$MaxFlat(i) = \begin{cases} 0 & \text{if } i > n \\ \max\left\{ \begin{array}{l} MaxFlat(i+1), \\ \max\left\{ \left(\sum_{j=i}^{k} [A[j] = A[i]] \right)^2 + MaxFlat(k+1) \;\middle|\; \begin{array}{l} i \le k \le n, \\ A[k] = A[i] \end{array} \right\} \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a 1D array evaluated in decreasing order of i. Each individual function evaluation can be performed in $O(n)$ time by either preprocessing all the sums (using dynamic programming! observe $\sum_{j=i}^{k}[A[j] = A[i]] = [A[k] = A[i]] + \sum_{j=i}^{k-1}[A[j] = A[i]]$) or by keeping a running count of how many copies of $A[i]$ have been encountered while looping over possible choices of k. There are $O(n)$ subproblems to solve, so ***the algorithm runs in $O(n^2)$ time.*** ■

Rubric: 10 points: standard dynamic programming rubric. A correct $O(n^2|\Sigma|)$ time algorithm is worth full credit; we did say "fixed" alphabet. A correct $O(n^3)$ time algorithm is worth 8 points.

3. ***Practice only. Do not submit solutions.***

(a) Describe and analyze an efficient algorithm to find the length of the shortest decomposition of a given input string into palindromes.

Solution (design): Let $A[1..n]$ be the input string. We define two functions:

- $IsPal?(i,j)$ is TRUE if the substring $A[i..j]$ is a palindrome, and FALSE otherwise.
- $MinPals(k)$ is the minimum number of palindromes that make up the suffix $A[k..n]$.

We need to compute $MinPals(1)$.

Every string with length at most 1 is a palindrome. A string of length 2 or more is a palindrome if and only if its first and last characters are equal *and* the rest of the string is a palindrome. Thus:

$$IsPal?(i,j) = \begin{cases} \text{TRUE} & \text{if } i \geq j \\ (A[i] = A[j]) \wedge IsPal?(i+1, j-1) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $IsPal?[1..n, 0..n]$. Each entry $IsPal[i,j]$ depends only on $IsPal[i+1, j-1]$. Thus, we can fill this array row-by-row, from the bottom row upward, in $O(n^2)$ time.

The empty string can be partitioned into zero palindromes. Otherwise, the best palindrome decomposition has at least one palindrome. If the first palindrome in the *optimal* decomposition of $A[1..n]$ ends at index ℓ, the remainder must be the *optimal* decomposition for the remaining characters $A[\ell+1..n]$. The following recurrence considers all possible values of ℓ.

$$MinPals(k) = \begin{cases} 0 & \text{if } k > n \\ 1 + \min\left\{ MinPals(\ell+1) \;\middle|\; k \leq \ell \leq n \text{ and } IsPal?(k,\ell) \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $MinPals[1..n]$. Each entry $MinPals[k]$ depends only on entries $MinPals[\ell+1]$ with $\ell \geq k$. Thus, we can fill this array from right to left, in $O(n^2)$ time.

The subroutine COMPUTEISPAL? runs in $O(n^2)$ time, and the rest of the algorithm also clearly runs in $O(n^2)$ time. So the total running time is $\boldsymbol{O(n^2)}$.

If we had used the obvious iterative algorithm to test whether each substring is a palindrome, instead of precomputing the array *IsPal?*, our algorithm would have run in $O(n^3)$ time. ■

Solution (pseudocode): The following algorithms both run in $O(n^2)$ ***time***. This first algorithm precomputes a boolean array *IsPal?*, where *IsPal?*$[i, j] = \textsc{True}$ if and only if the substring $A[i\,..\,j]$ is a palindrome.

```
ComputeIsPal?(A[1..n]):
  for i ← n down to 1
      IsPal?[i, i−1] ← True
      IsPal?[i, i] ← True
      for j ← i+1 to n
          if A[i] = A[j]
              IsPal?[i, j] ← IsPal?[i+1, j−1]
          else
              IsPal?[i, j] ← False
```

Here is the main algorithm; *MinPals*$[k]$ stores the minimum number of palindromes whose concatenation is the suffix $A[k\,..\,n]$.

```
MinPals(A[1..n]):
  ComputeIsPal?(A[1..n])
  MinPals[n+1] ← 0
  for k ← n down to 1
      MinPals[k] ← ∞
      for ℓ ← k to n
          if IsPal?[k, ℓ]
              MinPals[k] ← min{MinPals[k], 1 + MinPals[ℓ+1]}
  return MinPals[1]
```

■

Rubric: 5 points, standard dynamic programming rubric. These are not the only correct solutions. As usual, each of these solutions is independently worth full credit. A correct algorithm that runs in $O(n^3)$ time (the same *MinPals* recurrence, but checking for palindromes by brute force) is worth at most 4 points; scale partial credit.

This was the fastest algorithm known for minimum palindrome decomposition until 2014, when three independent groups of researchers announced algorithms that run in $O(n \log n)$ time; all three algorithms rely on similar structural properties of overlapping palindromes. Finally, in 2017, Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur announced the first $O(n)$-time algorithm; their algorithm uses a combination of "four Russians" optimization and range searching to shave off the last log factor.

Not surprisingly, the greedy algorithm *"Remove the longest palindrome substring and recurse"* does ***not*** yield an optimal decomposition. The string AABAAAABBA can be decomposed into three palindromes AABAA • A • ABBA, but the best decomposition that includes the longest palindrome substring is AA • **BAAAAB** • B • A. Similar examples show that repeatedly removing the longest palindrome *prefix* or *suffix* is also incorrect.

(b) Describe and analyze an efficient algorithm to find the length of the shortest metapalindrome for a given string.

Solution: Again, let $A[1..n]$ be the input string. For any index i, let $ShMeta(i)$ denote the length of the shortest metapalindrome of the substring $A[i+1..n-i]$; this substring is the result of removing the first i and last i characters of A.

We need to compute $ShMeta(0)$.

This function satisfies the following recurrence:

$$ShMeta(i) = \begin{cases} 0 & \text{if } i > \lceil n/2 \rceil \\ 1 & \text{if } i \le \lceil n/2 \rceil \text{ and } IsPal?(i+1, n-i) \\ 2 + \min \left\{ ShMeta(j) \;\middle|\; \begin{array}{ll} i < j \le \lceil n/2 \rceil & \text{and} \\ IsPal?(i+1, j) & \text{and} \\ IsPal?(n-j+1, n-i) & \end{array} \right\} & \text{otherwise} \end{cases}$$

Here we are using the same *IsPal?* function from part (a). We can memoize this function into an array $ShMeta[0..\lceil n/2 \rceil + 1]$, which we can fill from right to left in $\boldsymbol{O(n^2)}$ ***time*** after memoizing all values of *IsPal?*. ■

Solution: The following algorithm, which calls the ComputeIsPal? subroutine from part (a), runs in $\boldsymbol{O(n^2)}$ ***time***.

```
ShortestMetaPalindrome(A[1..n]):
  ComputeIsPal?(A[1..n])
  m ← ⌈n/2⌉                    ⟨⟨middle index⟩⟩
  ShMeta[m+1] ← 0
  for i ← m−1 down to 0
      if IsPal?[i+1, n−i]
          ShMeta[i] ← 1
      else
          ShMeta[i] ← ∞
          for j ← i+1 to m
              if IsPal?[i+1, j] and IsPal?[n−j+1, n−i]
                  ShMeta[i] ← min {ShMeta[i], 2 + ShMeta[j]}
  return ShMeta[0]
```

Here $ShMeta[i]$ stores the length of the shortest metapalindrome of the substring $A[i+1..n-i]$. ■

Rubric: 5 points, standard dynamic programming rubric. As usual, each of these solutions is independently worth full credit. These are not the only correct solutions. A correct algorithm that runs in $O(n^3)$ time (the same main recurrence, but checking for palindromes by brute force) is worth at most 4 points; scale partial credit.

Given the recent results for part (a), an algorithm that runs in $O(n \log n)$ or even $O(n)$ time seems likely, but as far as I know this is still an open problem.