1. For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \leq n$.

(a) Describe and analyze an algorithm to determine whether $X$ occurs as two disjoint subsequences of $Y$, where “disjoint” means the two subsequences have no indices in common. For example, the string PPAP appears as two disjoint subsequences in the string PENAPPLEAPPLEPEN, but the string PEEPLE does not.

Solution: To simplify the following case analysis, we add a sentinel character $X[0] = \#$ that does not appear anywhere else in $X$ or $Y$.

For any indices $i$, $j$, and $\ell$, we defined a boolean value $\text{DisjSS}(i, j, \ell)$, which is TRUE if the prefixes $X[1..i]$ and $X[1..j]$ both occur as disjoint subsequences of the prefix $Y[1..\ell]$, and FALSE otherwise. We need to compute $\text{DisjSS}(k, k, n)$.

This function satisfies the following recurrence:

$$\text{DisjSS}(i, j, \ell) = \begin{cases} 
\text{TRUE} & \text{if } i = 0 \text{ and } j = 0 \\
\text{FALSE} & \text{if } i + j > \ell \\
\text{DisjSS}(i, j, \ell - 1) & \\
\lor \left( \left( Y[\ell] = X[i] \right) \land \text{DisjSS}(i - 1, j, \ell - 1) \right) & \text{otherwise} \\
\lor \left( \left( Y[\ell] = X[j] \right) \land \text{DisjSS}(i, j - 1, \ell - 1) \right) & 
\end{cases}$$

We can memoize this function into a three-dimensional array $\text{DisjSS}[0..k, 0..k, 0..n]$. Each entry $\text{DisjSS}[i, j, \ell]$ depends only on entries of the form $\text{DisjSS}[\cdot, \cdot, \ell - 1]$. Thus, we can fill the array with three nested for loops, increasing $\ell$ in the outermost loop, and considering $i$ and $j$ in arbitrary order in the inner loops.

```plaintext
\text{DisjSS}(X[1..k], Y[1..n]):
X[0] \leftarrow \#
for \ell \leftarrow 0 \text{ to } n 
  for i \leftarrow 0 \text{ to } k 
    for j \leftarrow 0 \text{ to } k 
      if i + j = 0
        \text{DisjSS}[0, 0, \ell] \leftarrow \text{TRUE}
      else if i + j > \ell
        \text{DisjSS}[i, j, \ell] \leftarrow \text{FALSE}
      else
        \text{DisjSS}[i, j, \ell] \leftarrow \text{DisjSS}[i, j, \ell - 1]
        \lor \left( \left( Y[\ell] = X[i] \right) \land \text{DisjSS}[i - 1, j, \ell - 1] \right)
        \lor \left( \left( Y[\ell] = X[j] \right) \land \text{DisjSS}[i, j - 1, \ell - 1] \right)
return \text{DisjSS}[k, k, n]
```

The resulting algorithm runs in $O(nk^2)$ time.

Rubric: 5 points: standard dynamic programming rubric
(b) Describe and analyze an algorithm to compute the number of occurrences of $X$ as a subsequence of $Y$. For example, the string PPAP appears exactly 23 times as a subsequence of the string PENPINEAPPLEAPPLEPEN. If all characters in $X$ and $Y$ are equal, your algorithm should return $\binom{n}{k}$. For purposes of analysis, assume that each arithmetic operation takes $O(1)$ time.

**Solution:** For any indices $i$ and $j$, let $\text{NumSS}(i, j)$ denote the number of times the prefix $X[1..i]$ appears as a subsequence of the prefix $Y[1..j]$. We need to compute $\text{NumSS}(k, n)$. This function obeys the following recurrence:

$$\text{NumSS}(i, j) = \begin{cases} 
1 & \text{if } i = 0 \\
0 & \text{if } i > j \\
\text{NumSS}(i, j - 1) & \text{if } X[i] \neq Y[j] \\
\text{NumSS}(i, j - 1) + \text{NumSS}(i - 1, j - 1) & \text{if } X[i] = Y[j]
\end{cases}$$

These cases may require some explanation.

- The objects we are actually counting are functions $f : \{1, 2, \ldots, i\} \rightarrow \{1, 2, \ldots, j\}$ such that $f(\ell) > f(\ell - 1)$ and $X[\ell] = Y[f(\ell)]$ for all $1 \leq \ell \leq i$. For any set $S$, there is exactly one function $f : \emptyset \rightarrow S$: the empty function! This function vacuously satisfies whatever condition you want for all $1 \leq \ell \leq 0$. Also, 1 is the only value for $\text{NumSS}(0, j)$ that actually makes the recurrence correct.

- If $X[i] = Y[j]$, then either each occurrence of $X[1..i]$ as a subsequence of $Y[1..j]$ either includes $Y[j]$ or omits $Y[j]$, and these two choices are exclusive. If $X[i] \neq Y[j]$, then each occurrence of $X[1..i]$ as a subsequence of $Y[1..j]$ definitely excludes $Y[j]$.

We can memoize this function into an array $\text{NumSS}[0..k, 0..n]$. Each entry $\text{NumSS}[i, j]$ depends only on entries $\text{NumSS}[i, j - 1]$ in the previous column, so we can fill the array using two nested loops, considering $j$ in increasing order in the outer loop and considering $i$ in arbitrary order in the inner loop.

```plaintext
NumSS(X[1..k], Y[1..n]):
for j ← 0 to n
  NumSS[0, j] ← True
for i ← 1 to k
  if i > j
    NumSS[i, j] ← False
  else if X[i] = Y[j]
    NumSS[i, j] ← NumSS[i, j - 1] + NumSS[i - 1, j - 1]
  else
    NumSS[i, j] ← NumSS[i, j - 1]
return NumSS[k, n]
```

The algorithm runs in $O(nk)$ time.

**Rubric:** 5 points: standard dynamic programming rubric
2. You are driving a bus along a long straight highway, full of rowdy, hyper, thirsty students and an endless supply of soda. Each minute that each student is on your bus, that student drinks one ounce of soda. Your goal is to drive all students home, so that the total volume of soda consumed by the students is as small as possible.

Your bus begins at an exit (probably not at either end) with all students on board and moves at a constant speed of 37.4 miles per hour. Each student needs to be dropped off at a highway exit. You may reverse directions as often as you like; for example, you are allowed to drive forward to the next exit, let some students off, then turn around and drive back to the previous exit, drop more students off, then turn around again and drive to further exits. (Assume that at each exit, you can stop the bus, drop off students, and if necessary turn around, all instantaneously.)

Describe an efficient algorithm to take the students home so that they drink as little soda as possible. Your algorithm will be given the following input:

- A sorted array \( L[1..n] \), where \( L[i] \) is the location of the \( i \)th exit, measured in miles from the first exit; in particular, \( L[1] = 0 \).
- An array \( N[1..n] \), where \( N[i] \) is the number of students you need to drop off at the \( i \)th exit.
- An integer \( \text{start} \) equal to the index of the starting exit.

Your algorithm should return the total volume of soda consumed by the students when you drive the optimal route.\(^1\)

**Solution:** Because the speed of the bus and the rate of each passenger's soda consumption are both fixed, we can measure the volume of soda in units of passenger-miles. That is, we want to minimize the sum over all passengers of the total number of miles that passenger travels on the bus.

We will drop off everyone who wants off at any particular exit the first time the bus reaches that exit. For all practical purposes, that exit then ceases to exist. If we stop the bus in the middle of its optimal route, we have already dropped off all passengers at all exits \( i \) through \( j \), for some integers \( 0 \leq i < j \leq n + 1 \).

For all indices \( i \leq j \), we define four values:

- \( \text{MinUp}(i, j) \) is the minimum volume of soda still to be consumed if we have already visited exits \( i \) through \( j \), and we are currently at exit \( j \).
- \( \text{MinDn}(i, j) \) is the minimum volume of soda still to be consumed if we have already visited exits \( i \) through \( j \), and we are currently at exit \( i \).
- \( \text{Before}(i) \) is the number of passengers we need to drop off at exits numbered less than \( i \).
- \( \text{After}(j) \) is the number of passengers we need to drop off at exits numbered greater than \( j \).

We need to compute \( \text{MinUp}(\text{start}, \text{start}) \).

The functions \( \text{Before} \) and \( \text{After} \) satisfy the following simple recurrences:

\[
\begin{align*}
\text{Before}(i) &= \begin{cases} 
0 & \text{if } i = 1 \\
\text{Before}(i - 1) + N[i - 1] & \text{otherwise}
\end{cases} \\
\text{After}(j) &= \begin{cases} 
0 & \text{if } j = n \\
\text{After}(j + 1) + N[j + 1] & \text{otherwise}
\end{cases}
\end{align*}
\]

\(^1\)Non-US students are welcome to assume kilometers and liters instead of miles and ounces. Late 18th-century French students are welcome to use decimal minutes.
We can memoize these functions into one-dimensional arrays; we can fill Before\([1 .. n]\) from left to right, and After\([1 .. n]\) from right to left.

The functions \(\text{MinUp}\) and \(\text{MinDn}\) obey the following mutual recurrence:

\[
\begin{align*}
\text{MinUp}(i, j) &= \begin{cases} 
\infty & \text{if } j > n \\
0 & \text{if } i = 1 \text{ and } j = n \\
\min \left\{ (\text{Before}(i) + \text{After}(j)) \cdot (L[j + 1] - L[j]) + \text{MinUp}(i, j + 1) \right\} & \text{otherwise}
\end{cases} \\
\text{MinDn}(i, j) &= \begin{cases} 
\infty & \text{if } i < 1 \\
0 & \text{if } i = 1 \text{ and } j = n \\
\min \left\{ (\text{Before}(i) + \text{After}(j)) \cdot (L[j + 1] - L[i]) + \text{MinUp}(i, j + 1) \right\} & \text{otherwise}
\end{cases}
\end{align*}
\]

We can memoize each of these functions into its own two-dimensional array. Assuming the arrays Before and After are already filled, we simultaneously fill \(\text{MinUp}\) and \(\text{MinDn}\) with two nested loops, one increasing \(i\) and the other decreasing \(j\).

```plaintext
\text{MinSoda}(L[1 .. n], N[1 .. n], start):
\langle \langle \text{- compute Before and After counts -} \rangle \rangle
\text{Before}[1] \leftarrow 0
\text{for } i \leftarrow 2 \text{ to } n
\quad \text{Before}[i] \leftarrow \text{Before}[i - 1] + N[i - 1]
\text{After}[n] \leftarrow 0
\text{for } j \leftarrow n - 1 \text{ down to } 1
\quad \text{After}[j] \leftarrow \text{After}[j + 1] + N[j + 1]
\langle \langle \text{- main mutual recurrence -} \rangle \rangle
\text{for } j \leftarrow 1 \text{ to } n + 1
\quad \text{MinDn}[0, j] \leftarrow \infty
\text{for } i \leftarrow 1 \text{ to } n
\quad \text{MinUp}[i, n + 1] \leftarrow \infty
\quad \text{for } j \leftarrow n \text{ down to } i
\quad \quad \text{if } i = 1 \text{ and } j = n
\quad \quad \quad \text{MinUp}[i, j] \leftarrow 0
\quad \quad \quad \text{MinDn}[i, j] \leftarrow 0
\quad \quad \text{else}
\quad \quad \quad \text{count} \leftarrow \text{Before}[i] + \text{After}[j]
\quad \quad \quad \text{up} \leftarrow \text{count} \cdot (L[j + 1] - L[j]) + \text{MinUp}[i, j + 1]
\quad \quad \quad \text{dn} \leftarrow \text{count} \cdot (L[j] - L[i - 1]) + \text{MinDn}[i - 1, j]
\quad \quad \quad \text{MinUp}[i, j] \leftarrow \min\{\text{up}, \text{dn}\}
\quad \quad \quad \text{up} \leftarrow \text{count} \cdot (L[j + 1] - L[i]) + \text{MinUp}[i, j + 1]
\quad \quad \quad \text{dn} \leftarrow \text{count} \cdot (L[i] - L[i - 1]) + \text{MinDn}[i - 1, j]
\quad \quad \quad \text{MinDn}[i, j] \leftarrow \min\{\text{up}, \text{dn}\}
\text{return MinUp(start, start)}
```

The overall algorithm runs in \(O(n^2)\) time.

\[\blacksquare\]

**Rubric:** 10 points: standard dynamic programming rubric. Reduce maximum score by 2 points for every extra factor of \(n\) (the number of stops) and 3 points for every factor of \(N\) (the total number of passengers) in the final running time; scale partial credit. This is not the only correct formulation of this algorithm.
3. **Vankin’s Mile** is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

For example, given the grid below, the player can score $8 - 6 + 7 - 3 + 4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is not the best possible score for these values.)

$$
\begin{array}{cccc}
-1 & 7 & -8 & 10 \\
-4 & -9 & 8 & -6 \\
5 & -2 & -6 & 7 \\
-7 & 4 & 7 & -3 \\
7 & 1 & -6 & 4 & -9 \\
\end{array}
$$

(a) Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin’s Mile, given the $n \times n$ array of values as input.

**Solution:** Let $VMile(i, j)$ denote the maximum possible score starting at square $(i, j)$. We need to compute $\max \{VMile(i, j) \mid 1 \leq i \leq n \text{ and } 1 \leq j \leq n\}$.

Suppose the scores are given in an array $A[1..n, 1..n]$. The function $Vmile$ satisfies the following recurrence:

$$VMile(i, j) = \begin{cases} 
0 & \text{if } i > n \text{ or } j > n \\
A[i, j] + \max \{VMile(i + 1, j), VMile(i, j + 1)\} & \text{otherwise}
\end{cases}$$

We can memoize this function into a two-dimensional array $VMile[1..n+1, 1..n+1]$, which we can fill with two nested loops considering both $i$ and $j$ in decreasing order. (It doesn’t matter which index we use in which loop.)

```python
VANKIN'S MILE[A[1..n, 1..n]]:
maxScore ← 0
for i ← n + 1 down to 1
   VMile[i, n + 1] ← 0
for j ← n down to 1
   VMile[n + 1, j] ← 0
   for i ← n down to 1
      VMile[i, j] ← A[i, j] + max{VMile[i + 1, j], VMile[i, j + 1]}
      maxScore ← max{maxScore, VMile[i, j]}
return maxScore
```

The resulting algorithm runs in $O(n^2)$ time.

**Rubric:** 5 points: standard dynamic programming rubric
(b) In the Canadian version of this game, appropriately called Vankin’s Kilometer, the player can move the token either one square down, one square right, or one square left in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin’s Kilometer, given the \( n \times n \) array of values as input.\(^2\)

Solution: We define two values for each pair of indices \( i \) and \( j \):

- \( VKiloL(i, j) \) is the maximum score starting at \((i, j)\) if we only move left in row \( i \).
- \( VKiloR(i, j) \) is the maximum score starting at \((i, j)\) if we only move right in row \( i \).

We need to compute \( \max \{VKilo(i, j) | 1 \leq i \leq n \text{ and } 1 \leq j \leq n\} \), where \( VKilo(i, j) = \max\{VKiloL(i, j), VKiloR(i, j)\} \).

As in part(a), assume the scores are given in an array \( A[1..n, 1..n] \). These functions \( VKiloL \) and \( VKiloR \) satisfy the following mutual recurrences:

\[
\begin{align*}
VKiloL(i, j) &= \begin{cases} 
0 & \text{if } i > n \text{ or } j < 1 \\
A[i, j] + \max \{VKiloL(i, j - 1), VKiloL(i + 1, j)\} & \text{otherwise} 
\end{cases} \\
VKiloR(i, j) &= \begin{cases} 
0 & \text{if } i > n \text{ or } j > n \\
A[i, j] + \max \{VKiloR(i, j + 1), VKiloR(i + 1, j)\} & \text{otherwise} 
\end{cases}
\end{align*}
\]

We can memoize these functions into two-dimensional arrays. We fill both arrays simultaneously with two nested loops, decreasing \( i \) in the outer loop, increasing \( j \) in the inner loop for \( VKiloL \), and decreasing \( j \) in the inner loop for \( VKiloR \).

```plaintext
VANKIN's KILOL(A[1..n, 1..n]):
maxScore ← 0
for j ← n down to 0
    VKiloL[i, j] ← 0
    VKiloR[i, j + 1] ← 0
for i ← n down to 1
    VKiloL[0, j] ← 0
for j ← 1 to n
    VKiloL[i, j] ← A[i, j] + \max\{VKiloL[i, j - 1], VKiloL(i + 1, j), VKiloR(i + 1, j)\}
    maxScore ← \max\{maxScore, VKiloL[i, j]\}
VKiloL[n + 1, j] ← 0
for j ← n down to 1
    VKiloR[i, j] ← A[i, j] + \max\{VKiloR(i, j + 1), VKiloL(i + 1, j), VKiloR(i + 1, j)\}
    maxScore ← \max\{maxScore, VKiloR[i, j]\}
return maxScore
```

The resulting algorithm runs in \( O(n^2) \) time.

**Rubric:** 5 points: standard dynamic programming rubric. This is one of the few cases where a boxes-and-arrows cartoon is more confusing than actual pseudocode.

---

\(^2\)If we also allowed upward movement, the resulting game (Vankin’s Fathom?) would be NP-hard.