Q1-a

Considering the following counter example:

	1	2	3
1	5	1	2
2	2	1	7
3	1	2	3

The given algorithm in Q1-a would find the path (1, 1) → (2, 1) → (3, 2) with total value 9.
However, there exists a valid path with larger value.
Considering the path (3, 1) → (3, 2) → (3, 3) which is a valid path with value 12.
Therefore, the path constructed by this algorithm is not necessary optimal.

There is no counter example for n = 2.
Since all the paths in the grid (i.e. (1, 1) → (2,1), (1, 1) → (2,2), (1, 2) → (2,1), (1, 2) → (2,2)) would be covered by the algorithm, and finds the summation of max{value(1, 1), value(1, 2)} and max{value(2, 1), value(2, 2)}. Therefore, the algorithm works for n = 2.

Q1-b

Dynamic Programming Algorithm:

(a) A definition of a polynomial number of subproblems that will be solved:
Compute V(i, j) = maximum path value ending at (i, j), where $1 \leq i \leq n$, $1 \leq j \leq n$.

(b) A recursive formula to compute the solution to each subproblems:

$$V(i, j) = \begin{cases} G(i, j) & i = 1 \\ \max\{V(i - 1, j), V(i - 1, j + 1)\} + G(i, j) & j = 1 \\ \max\{V(i - 1, j - 1), V(i - 1, j)\} + G(i, j) & j = n \\ \max\{V(i - 1, j - 1), V(i - 1, j), V(i - 1, j + 1)\} + G(i, j) & O/W \end{cases}$$

(c) Derive solution to original problems from solutions to subproblems.

```
MaxValue(G)
# j_1 := a column that contains the maximum element in row 1
for j := 1 to n do
        V[1, j] = G[1, j]
i := 1
# get the maximum path value ending at each entry in the grid
while i < n do
        i := i + 1
        for j := 1 to n do
                if (j = 1) then
                        V[i, j] = max{V[i - 1, j], V[i - 1, j + 1]} + G[i, j]
                else if (j = n) then
                        V[i, j] = max{V[i - 1, j - 1], V[i - 1, j]} + G[i, j]
                else then
                        V[i, j] = max{V[i - 1, j - 1], V[i - 1, j], V[i - 1, j + 1]} + G[i, j]
# find the maximum path value at row n
max_value := -infinite
for j := 1 to n do
        if (max_value < V[n, j]) then
                max_value = V[n, j]
return max_value
```

Proving Correctness:

Let $V[i, j_i]$ be the maximum **path value** of a path from row 1 to row i.

Let P be a **path** with maximum path value ending at row i.

Thus $P = P' \rightarrow (i, j_i)$, where P' is the path with maximum path value ending at row (i - 1), and able to get to the square (i, j_i) [i.e. $j_i = j_{i-1}$ or $j_{i-1}-1$ (and $j_{i-1}>1$) or $j_{i-1}+1$ (and $j_{i-1}<n$)]

Case 1: for i = 1,

then the maximum path value to reach each square in the first row is the value of every single square. In this case, V[1, j] = G[1, j].

Case 2: for $1 < i \leq n$,

By a straightforward cut-and-paste argument, P' is

(1) (a path of G ending at $(i-1, j_{i-1}-1)$ (and $j_{i-1}>1$) or $(i-1, j_{i-1})$ or $(i-1, j_{i-1}+1)$ (and $j_{i-1}<n$).

(2) has maximum path value among the paths ending at $(i-1, j_{i-1}-1)$ (and $j_{i-1}>1$) or $(i-1, j_{i-1})$ or $(i-1, j_{i-1}+1)$ (and $j_{i-1}<n$).

If P' does not satisfy (a) and (b),

Either $P = P' \rightarrow (i, j_i)$ is not a path with maximum value to (i, j_i), which contradicts the definition of P;

Or P' can be replaced by a path with larger path value ending at $(i-1, j_{i-1}-1)$ (and $j_{i-1}>1$) or $(i-1, j_{i-1})$ or $(i-1, j_{i-1}+1)$ (and $j_{i-1}<n$) resulting in a path in G ending at (i, j_i), that is the path to

row i with larger path value P[by arithmetic calculation], which again contradicts the definition of P.
Therefore, in Case 2, V(i, j) = max{V[i-1, j-1] (if j>1), V[i-1, j], V[i-1, j+1] (if j<n)}+G[i, j].

Combining Case 1 and 2,

$$V(i,j) = \begin{cases} G(i,j) & i = 1 \\ \max\{V(i-1,j), V(i-1,j+1)\} + G(i,j) & j = 1 \\ \max\{V(i-1,j-1), V(i-1,j)\} + G(i,j) & j = n \\ \max\{V(i-1,j-1), V(i-1,j), V(i-1,j+1)\} + G(i,j) & \text{O/W} \end{cases}$$

(ii) The computation in (c) yields a solution to the given problem:
By definition the path with the largest value ending at the **first** row is max{G(1, j)} as wanted.
Now take row $i > 1$, and suppose by way of induction that MaxValue(G') correctly computes max{V(i, j)} where G' is a $i \times n$ grid for all $i < n$ and for all $1 \leq j \leq n$. By the induction hypothesis, we know that MaxValue(G*) = max{V(n - 1, j)}, where G* is a $(n - 1) \times n$ grid.
max{V(n, j)} = **max{**max{V(n-1, j-1) (if j>1), V(n-1, j), V(n-1, j+1) (if j<n)}+G(n, j)**}**
[by the definition of V]
= MaxValue(G) [by the definition of V] as wanted. Q.E.D

Running Time: $\Theta(n^2)$

Q1-c

```
PathWithMaxValue(G):
# j_1 := a column that contains the maximum element in row 1
for j := 1 to n do
        V[1, j] = G(1, j)  ; pre(i, j) = j
i := 1
while i < n do
        i :=  i  + 1
        for j := 1 to n do
                if (j = 1) do
                        V[i, j] = max{V[i - 1, j], V[i - 1, j + 1]}  + G[i, j]
                        if V[i - 1, j] > V[i - 1, j + 1] then
                                pre(i, j) = j
                        else then
                                pre(i, j) = j + 1
                else if (j = n) do
```

```
                    V[i, j] = max{V[i - 1, j - 1], V[i - 1, j]}  + G[i, j]
                    if V[i - 1, j - 1] > V[i - 1, j] then
                        pre(i, j) = j - 1
                    else then
                        pre(i, j) = j
                else do
                    V[i, j] = max{V[i - 1, j - 1], V[i - 1, j], V[i - 1, j + 1]} + G[i, j]
                    if V[i-1, j-1] > V[i-1, j] and V[i-1, j-1] > V[i-1, j+1] then
                        pre(i, j) = j - 1
                    else if V[i-1, j] > V[i-1, j-1] and V[i-1, j] > V[i-1, j+1] then
                        pre(i, j) = j
                    else then
                        pre(i, j) = j + 1
# find the maximum path value at row n and the path with maximum value
max_value := -infinite
P := list of size n
for j := 1 to n do
    if (max_value < V[n, j]) then
        max_value = V[n, j]
        P[n] = j
k := n - 1
while (k > 0) do
    P[k] = pre(k, pre(k + 1, P[k+1]))
    k = k - 1
return max_value, P
```

Q2

a. Define a sequence $A = a_1, a_2, \ldots, a_n$ of numbers.(i.e. a_i is the i^{th} element of A)
Let **L**[i] be the length of a Longest Single-Peak Subsequence (LSPS) of A ending at position i.
Let **S** be a Longest Single-Peak subsequence of A ending at position i. (i.e. $S = S'a_i$)
S' is a longest Single-Peak subsequence of A ending at some position j such that $j < i$.
Define ***LSPS_shape*** as a property of a **longest single peak subsequence**.
LSPS_shape[i] keeps track of the shape of the longest single peak subsequence ending at position i.
For the *LSPS_shape* of a longest single peak subsequence, it can be either "increasing" or "non-increasing"(i.e. decreasing or having a single peak in the middle).
By default, every number in A is a single peak subsequence with length 1 of *LSPS_shape* "increasing".

b.

$$L(i) = \begin{cases} 1, \text{ when } i = 1 \\ \max\{L[j] + 1 : 1 <= j < i \text{ and } (\textit{LSPS\_shape}[j] = \text{“increasing” or } a_j > a_i)\}, \text{ when } i > 1 \end{cases}$$

The length of a LSPS S depends on the shape of the LSPS S' ending at position j and the value comparison of a_j and a_i. The relation shows as follows:

Relation*LSPS_shape* ending at j	increasing	non-increasing
$a_j > a_i$ where $1 <= j < i$	L[j] + 1 *LSPS_shape* → non-increasing	L[j] + 1 *LSPS_shape* → non-increasing
$a_j < a_i$ where $1 <= j < i$	L[j] + 1 *LSPS_shape* → increasing	L[j] *LSPS_shape* → non-increasing

c.

```
LSPS(A):
    # initialize two parallel arrays
    L := an array of size n contain all 0s
    LSPS_shape := an array of size n contain all “increasing”s
    for i := 1 to n do
        L[i] := 1
        for j := 1 to i - 1 do
            # see the analysis of this condition in the above form
            if (LSPS_shape[j]= “increasing” or a_j > a_i) and L[j] + 1 > L[i]
           then
```

```
9                          L[i] = L[j] + 1
10                         # shape changes
11                         if a_j > a_i then
12                                LSPS_shape[i] := “non-increasing”
13      return L[argmax(L)]
```

Correctness:

(i) The recursive formula in step (b) correctly computes the subproblem in step (a).

Let L[i] be the length of a LSPS of A ends at position i. Let S be a LSPS of A ends at position i, thus S = S’a_i where S’ is the longest proper prefix of S.

Case 1: i = 1
The LSPS only contain one element. S’ is empty.
⇒ L[i] = L[1] = 1

Case 2: i > 1
By a straightforward cut-and-paste argument, S’ a LSPS of A ending at position j for some j < i and S’ has maximum length among all such subsequences satisfy one of the following two properties:

1. The LSPS is an increasing sequence
2. The LSPS is decreasing (may contain at most a peak) and $a_j < a_i$

⇒ L[i] = max{L[j] + 1: 1 <= j < i and (*LSPS_shape* is increasing or $a_j > a_i$)}

Therefore,

$$L(i) = \begin{cases} 1, \text{ when } i = 1 \\ \max\{L[j] + 1: 1 <= j < i \text{ and } \\ \quad (\mathit{LSPS\_shape}[j] = \text{“increasing” or } a_j > a_i)\}, \text{ when } i > 1 \end{cases}$$

(ii) why the computation in (c) yields a solution to the given problem.

Line 5-10 will keep upgrade L[i], so that L[i] will be max{L[j] + 1 : 1<= j< i and (*LSPS_shape* is increasing or $a_j > a_i$)}, more precisely:

Line 6 make sure 1 <= j < i; Line 7 make sure the new LSPS construct from previous one is valid and longer; Line 8 upgrade L[i]; Line 9-10 keep track the shape of the LSPS (increasing or nonincreasing).

Line 4, compute L[i] for all possible i

Line 11, return the maximum L[i] which is a solution to the given problem.

Running Time: $\Theta(n^2)$