Problem 1 (20 points). Let $S_1 = AGCCTG$ and $S_2 = AGCT$ be two strings over alphabet $\Sigma = \{A, C, G, T\}$. Draw the dynamic programming table for computing the edit distance between S_1 and S_2.

Solution.

		A	G	C	T
	0	1	2	3	4
A	1	0	1	2	3
G	2	1	0	1	2
C	3	2	1	0	1
C	4	3	2	1	1
T	5	4	3	2	1
G	6	5	4	3	2

Problem 2 (20 points). A subsequence is *palindromic* if it is the same whether read left to right or right to left. Given a string A of length n, design a dynamic programming algorithm to find the longest palindromic subsequence of A. Your algorithm should run in $O(n^2)$ time. For example, the longest palindromic subsequence for string $A = ACTCGCATA$ is $ATCGCTA$.

Solution. Define $c[i, j]$ as the length of the *longest palindromic subsequences* within substring $A[i \cdots j]$. For $i + 1 \le j$, we have the *recursion* formula below (we define $\delta(A[i] = A[j]) = 1$ if $A[i] = A[j]$ and $\delta(A[i] = A[j]) = 0$ if $A[i] \neq A[j]$):

$$c[i, j] = \max \begin{cases} c[i+1, j-1] + 2 \cdot \delta(A[i] = A[j]) \\ c[i+1, j] \\ c[i, j-1] \end{cases}$$

The *pseudo-code*, which includes three steps, is as follows.

Initialization:

$c[i, i-1] = 0$, for all $1 \le i \le n$

$c[i, i] = 1$, for all $1 \le i \le n$

Iteration:

for $d = 1 \cdots n-1$:

  for $i = 1 \cdots n-d$:

    $j = i + d$

    $c[i, j] = \max\{c[i+1, j-1] + 2 \cdot \delta(A[i] = A[j]), c[i+1, j], c[i, j-1]\}$

  endfor

endfor

Termination: $c[1, n]$ gives length of the *longest palindromic subsequences* in A. (The corresponding indices can be fetched through introducing backtracing pointers.)

Running time: The initialization of $c[i, j]$ takes $O(n)$ time. The iteration step takes $O(n^2)$ time. The total running time of the algorithm is $O(n^2)$.

Problem 3 (20 points). Let $A[1 \cdots n]$ be an array with n integers (possibly with negative ones). Design a dynamic programming algorithm to compute indices i and j such that $\sum_{k: i \le k \le j} A[k]$ is maximized. Your algorithm should run in $O(n)$ time.

Solution. Let $S(i,j) = \sum_{k:i\leq k\leq j} A[k]$. Define $c[j] = \max_{i:1\leq i\leq j} S(i,j)$, i.e., $c[j]$ equals to the maximum sum of the consecutive elements in A that ends at $A[j]$. We have the *recursion* formula below:

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

The *pseudo-code*, which includes three steps, is as follows.

Initialization: $c[0] = -\infty$

Iteration:

```
for j = 1···n :
  if c[j−1] > 0
    c[j] = c[j−1] + A[j]
  else
    c[j] = A[j]
  endif
endfor
```

Termination: $\max_{j:1\leq j\leq n} c[j]$ gives the largest sum. (The corresponding indices can be fetched through introducing backtracing pointers.)

Running time: The iteration step takes $O(n)$ time, which dominates the overall running time.

Problem 4 (20 points). Let $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$ be two strings. Design a dynamic programming algorithm to compute the *longest common substrings* between A and B, i.e., to find the *largest* k and indices i and j such that $A[i\cdots i+k-1] = B[j\cdots j+k-1]$. Your algorithm should run in $O(mn)$ time.

Solution. Define $c[i,j]$ be the length of the longest common *suffix* of strings $A[1\cdots i]$ and $B[1\cdots j]$, i.e., $c[i,j]$ equals to the largest integer k such that $A[i-k+1\cdots i] = B[j-k+1\cdots j]$.

Recursion:

$$c[i,j] = \min \begin{cases} c[i-1,j-1]+1 & \text{if } A[i] = B[j] \\ 0 & \text{otherwise} \end{cases}$$

The dynamic programming algorithm is given below:

Initialization:

$c[i,0] = 0$, for $0 \leq i \leq n$
$c[0,j] = 0$, for $0 \leq j \leq m$

Iteration:

```
for i ← 1 to n do
  for j ← 1 to m do
    if (A[i−1] = B[j−1]) then
      c[i, j] ← c[i−1, j−1] + 1
    else
      c[i, j] ← 0
    end if
  end for
end for
```

Termination: compute $k^* = \max_{i,j} c[i,j]$ and $(i^*, j^*) = \arg\max_{i,j} c[i,j]$.

Running time: It takes constant time to compute each entry of table $c[\cdot,\cdot]$. Therefore, the total running time is $O(mn)$.

Problem 5 (20 points). Let $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$ be two strings. Design a dynamic programming algorithm to compute the *longest common subsequence* between A and B, i.e., to find the *largest* k and indices $1 \le i_1 < i_2 < \cdots < i_k \le n$ and $1 \le j_1 < j_2 < \cdots < j_k \le m$ such that $A[i_1] = B[j_1], A[i_2] = B[j_2], \cdots, A[i_k] = B[j_k]$. Your algorithm should run in $O(mn)$ time.

Solution. Define $LCS[i,j]$ as the length of the *longest common subsequences* between $a_1a_2\cdots a_i$ and $b_1b_2\cdots b_j$. Then we have the *recursion* formula below:

$$LCS[i,j] = \begin{cases} LCS[i-1,j-1]+1 & \text{if} \quad a_i = b_j \\ max\{LCS[i-1,j], LCS[i,j-1]\} & \text{if} \quad a_i \neq b_j \end{cases}$$

The *pseudo-code*, which includes three steps, is as follows.

Initialization:

$LCS[i,0] = 0$, for all i in $1,2,\ldots n$

$LCS[0,j] = 0$, for all j in $1,2,\ldots m$

Iteration:

```
for i = 1···n :
  for j = 1···m :
    if a_i = b_j: LCS[i,j] = LCS[i−1,j−1]+1
    else: LCS[i,j] = max{LCS[i−1,j], LCS[i,j−1]}
  endfor
endfor
```

Termination: $LCS[n,m]$ gives the length of the *longest common subsequences* between A and B. (The corresponding indices can be fetched through introducing backtracing pointers.)

Running time: The initialization of $LCS[i,j]$ takes $O(m+n)$ time. The iteration step have two embedded *for* loops, and therefore takes $O(mn)$ time. The total running time of the algorithm is $O(mn)$.