1.
```
1: procedure MaxScore(Score[1..n], Wait[1..n])
2:     dp[1..n, False..True] is set to all 0
3:     for all k ∈ {n, ..., 1} do
4:         dp[k, False] ← max(dp[k + 1, False], dp[k + 1, True])
5:         dp[k, True] ← Score[k] + max(dp[k + Wait[k] + 1, False], dp[k + Wait[k] + 1, True])
6:     end for
7:     return max(dp[1, False], dp[1, True])
8: end procedure
```

This algorithm clearly runs in $O(n)$ time.

$dp[k, dance]$ represents the maximum score you can obtain if you start (potentially) dancing at song k, where you dance if $dance = \textsc{True}$ and you don't otherwise. $dp[k, -]$ is vacuously defined to be 0 for $k > n$. The maximum score you can obtain if you start (potentially) dancing at song k is obviously the maximum of the two choices, so the algorithm is correct.

2. We work backwards, computing which states the NFA must be in to accept after reading $w[i]$.

```
1: procedure CheckNFA(inDelta[1..k, 0..1, 1..k], A[1..k], w[1..n])
2:     dp[1..n+1, 1..k] is set to all False
3:     dp[n+1, 1..k] ← A[1..k]
4:     for all i ∈ {n, ..., 1} do
5:         for all j ∈ {1, ..., k} do
6:             for all j' ∈ {1, ..., k} do
7:                 if inDelta[j, w[i], j'] = True then
8:                     dp[i, j] ← dp[i, j] ∨ dp[i+1, j']
9:                 end if
10:            end for
11:        end for
12:    end for
13:    return dp[1, 1]
14: end procedure
```

This algorithm clearly runs in $O(k^2 n)$ time.

$dp[i, j]$ represents whether or not the substring $w[i..n]$ is accepted when we start the NFA in state j. Vacuously $w[n+1..n] = \varepsilon$, the empty string. If the NFA can transition from state j to j' after reading $w[i]$, the NFA can only accept if it accepts the substring $w[i+1..n]$ and starting at j', so this algorithm is correct.

3. (a) We give a $O(n^2)$ algorithm.

```
1: procedure PalindromicLength(w[1..n])
2:     PL[0] ← 0
3:     P ← ∅
4:     for j ← 1 to n do
5:         P' ← ∅
6:         for all i ∈ P do
7:             if i > 1 and w[i − 1] = w[j] then
8:                 P' ← P' ∪ {i − 1}
9:             end if
10:        end for
11:        if i > 1 and w[j − 1] = w[j] then
12:            P' ← P' ∪ {j − 1}
13:        end if
14:        P ← P' ∪ {j}
15:        PL[j] ← j
16:        for all i ∈ P do
17:            PL[j] ← min(PL[j], PL[i − 1] + 1)
18:        end for
19:    end for
20:    return PL[n]
21: end procedure
```

This algorithm operates by computing and storing an array $PL[0..n]$, where $PL[0] = 0$ and $PL[j] =$ PalindromicLength$(w[1..j])$. At each step j of the loop starting on line 4, we compute the set P_j of starting positions of all palindromes ending at j. This is computed by noticing that $w[i..j]$ is a palindrome if and only if $w[i+1..j-1]$ is a palindrome and $w[i] = w[j]$. During the j-th iteration, we use $O(|P_j| + |P_{j-1}|) = O(n)$ time, so the running time of this algorithm is indeed $O(n^2)$.

To elaborate on the computation of $PL[j]$, we have several palindromes $w[i..j]$ stored in P_j. For each palindrome, the length of the decomposition is the palindromic length of $w[1..i-1]$ plus 1, because $w[i..j]$ is indeed a palindrome. Thus taking the min here is the right thing to do. The maximum value of $PL[j]$ is clearly j because we can decompose $w[1..j]$ into j length 1 strings, which are vacuously palindromes.

This algorithm can be improved to $O(n \log n)$ by considering prefixes $Z[1..n]$ of the infinite Zimin word Z over the alphabet $\mathbb{Z}^+$, which is the limit of the sequence Z_i, where $Z_0 = \varepsilon$ and $Z_k = Z_{k-1}kZ_{k-1}$, but according to @623, this improvement is not necessary.

(b) We first notice that we can precompute all the palindromes of $w[1..n]$ in $O(n^2)$ time. This is done by considering all possible centers (the n formal symbols and the $n-1$ empty spaces in between) and expanding on the left and right for a worst case expansion time complexity of $O(n)$. We will store the results in an array $P[1..n, 1..n]$ such that $P[i,j] = \textsc{True}$ if and only if $w[i..j]$ is a palindrome and denote this subroutine as ComputePalindromes.

We also notice that the metapalindrome must consist of a palindrome that is in some sense "centered" on $w[1..n]$. Thus, we find the metapalindromic length as follows.

```
1:  procedure MetapalindromicLength(w[1..n])
2:      P[1..n, 1..n] ← ComputePalindromes(w[1..n])
3:      center ← (n + 1)/2
4:      if n ≡ 0 (mod 2) then
5:          MPL[0] ← 0                (ab̲cd)
6:      else
7:          MPL[0] ← 1                (abc̲de)
8:      end if
9:      for i ← 1 to ⌊n/2⌋ do
10:         Z ← ∅
11:         a ← ⌈center − i⌉
12:         b ← ⌊center + i⌋
13:         for j ← 0 to i − 1 do
14:             a′ ← ⌈center − j⌉
15:             b′ ← ⌊center + j⌋
16:             if P[a, a′ − 1] = True and P[b′ + 1, b] = True then
17:                 Z ← Z ∪ {2 + MPL[j]}
18:             end if
19:         end for
20:         if P[a, b] = True then
21:             Z ← Z ∪ {1}
22:         end if
23:         Z ← Z ∪ {b − a + 1}
24:         MPL[i] ← min(Z)
25:     end for
26:     return MPL[⌊n/2⌋]
27: end procedure
```

This algorithm is clearly $O(n^2)$ from our $O(n^2)$ precomputation and $O(n^2)$ loop on line 8 (note that $\min(Z)$ is a minimum of a set of size $O(n)$).

Our approach to the algorithm is as follows. On iteration i of the loop, we compute the set Z of possible metapalindrome lengths and take the minimum. We compute a and b to be the positions of the first and last character in $w[1..n]$ after our expanding i characters from the center (e.g. the space in between b and c in $abcd$ and b in abc). We then consider all the smaller expansions from the center, with bounds a' and b', which have an associated metapalindrome length stored in $MPL[j]$. We then split the string $w[a..b]$ into $w[a..b] = w[a..a'-1]w[a'..b']w[b'+1..b]$. If this is a metapalindrome decomposition, we add 2 to the length of the inside decomposition because we added 2 new palindromes. If $w[a..b]$ is a palindrome, then the metapalindrome length is clearly 1. The maximum also makes sense, because $11\cdots11$ is clearly a palindrome. Thus the algorithm is correct.