Disclaimer

The hints, explanations, and code snippets provided herein are offered solely for educational and illustrative purposes. They are not intended to be exhaustive, definitive, or officially sanctioned solutions.

The hints, explanations, and code snippets provided herein are based on the "Exam Q: Maximum sum increasing subsequence (DP)" code example on codeexpert for the course Data Structures and Algorithms (252-0002-00L).

While reasonable effort has been made to ensure accuracy and clarity, no guarantee is made as to their completeness, reliability, or fitness for a particular purpose.

Adel Gavranović, the author, disclaims any liability for direct or indirect consequences arising from the use or misuse of this information.

Users are encouraged to independently verify and adapt any content to suit their specific requirements and contexts.

Annotated Recursive Solution

```
int max_sum_recursive(const std::vector<int>& Seq, int start) {
  int n = Seq.size();

  // BASE CASE
  if (start == n) {
    return 0;
  }

  // RECURSIVE CASE
  int max_sum;                          // `Seq` contains the sequence
  int prev;                             // [*****************]

  if (start == 0){                      // [s***************]  CASE A
    max_sum = Seq[0];                   //  ^---start
    prev = Min; // (cf. line 5)
  } else {                              // [******ps*********]  CASE B
    max_sum = 0;                        //        ^^---start
    prev = Seq[start-1];                //        |----prev
  }

  for (int i = start; i<n; ++i){        // [******poooooooooo]  LOOP over o's
    if (prev < Seq[i]){                 // p < o   ==>   growing subsequence!
      int maybeNextMax = Seq[i] + max_sum_recursive(Seq, i+1);  // <-- rec. call
      // and now we consider the subproblem on this next |part| in `Seq`:
      //                                  [*...*o...oø|oooo|]
      max_sum = std::max(max_sum, maybeNextMax);
      // update `max_sum` if a higher sum has been found
    }
  }

  return max_sum;
}
```

Steps for Dynamic Programming Solution

```
int solve(const std::vector<int>& Seq){

  // [Definition of the subproblems / the DP table]


  // [Recursion: Computation of an entry:]




  // [Computation order (topological order)]

  // [Solution and Running Time]


}
```

Full Steps for Dynamic Programming Solution

```cpp
int solve(const std::vector<int>& Seq){

  // [Definition of the subproblems / the DP table]
  //   - Array size:   same size as the sequence in `arr`
  //   - Entry:        max_sum (acc. to specs) for [i, n),
  //                   where n = Seq.size().

  // [Recursion: Computation of an entry:]
  //   - A[i] = Seq[i]
  //     - by the fact that single number is valid increasing subsequence
  //   - maximum over all possible j, j>i: A[j] + Seq[i] where Seq[i] < Seq[j]
  //     - NOTE FOR LATER: so we're gonna have to loop over all j: i < j
  //       - seems inefficient but is better than recursion so ¯\_('_')_/¯

  // [Computation order (topological order)]
  //   - from right to left, i.e. A[i] depends on all A[j] for i < j

  // [Solution and Running Time]
  //   - extract the solution at the max value within the whole array A
  //   - runtime: O(n^2), because we perform c*n (c in [0,1]) operations
  //                      at each of the n entries, thus O(n*n).
}
```

Annotated Dynamic Programming Solution

```
int solve(const std::vector<int>& Seq){

  // Definition of the DP Table
  int n = Seq.size();
  std::vector<int> A(n);

  // Computation of "first" entries
  for (int i = 0; i < n; ++i){
    A[i] = Seq[i];                              // From the description we know
  }                                             // that a single element is a
                                                // valid increasing subsequence

  // Computation of all other entries
  for (int i = n-1; i >= 0; --i){               // Loop over all entries i
    for (int j = i+1; j < n; ++j){              //  Loop over all entries j: i < j
      if (Seq[i] < Seq[j]){                     //   if Seq[i] and Seg[j] are "in order"
        A[i] = std::max(A[i], Seq[i] + A[j]); //    update IF that would
      }                                         //    increase the max_sum
    }
  }

  // Return of the solution
  return *std::max_element(A.begin(), A.end());   // picks maximum element
  //              ^---returns iterator to greatest element in [begin, pastTheEnd)
  //     ^---dereferences that iterator to get the actual (int) value
}
```