Dynamic Programming CH 6

Frank

October 20, 2021

1 Longest Increasing Subsequence (LIS)

- Sequence $A = a_1, a_2, \ldots, a_n$

  Subsequence of $A : a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

- For example, consider sequence

$$6, 7, 4, 2, 5, 3, 9, 8, 5$$

  A subsequence might be $7, 5, 3, 9, 5$ (Does not have to be adjacent)

- ***Input***: Sequence A of numbers
- ***Output***: A LIS of A

1.1 Solution Attempt

- Approaching a problem with dynamic program: Work backwards, pretend that someone has given us the optimal solution. Look at the solution and analyze it backwards to see how the solution is made up of solutions to smaller problems (Optimal substructure/Principle of Optimality).

- Suppose S is a LIS of A.

  $S = S'a_i$ for some i where a_i, element of A is where S ends (last element of sequence S).

  S' is a

  1. longest
  2. increasing subsequence of A
  3. that [has the condition where it] ends in some position j of A
  4. s.t. $j < i$ and $a_j < a_i$

  by "cut-and-paste" argument (If there was a S'' with above conditions 2-4 and is longer than S', then we would get that S is not LIS of A, which is a contradiction. So our S must have been a longer LIS in the first place, so we "cut-and-paste" that longer LIS in place of S and we have the above is true again, that S' is the LIS with conditions 2-4 in place). Note that 2-4 are true by trivial reasons, since S' is a part of S that does not include the last element of S.

  Note then that our optimal solution is an "extension" (may be a shortening) of an optimal solution to a smaller problem (a sequence that does not include a_i or any elements after that, thus a **substructure**). This means we have uncovered the **Optimal Substructure** (A necessary component of a DP algorithm), which is S' in this case.

1.2 An Alternate Problem (Subproblem)

- Find the length of a LIS of A, then **retrofit** the problem to finding a LIS of A.

  Note this is like dijkstra's algorithm, where we find the weight of the shortest paths, then use that to determine our actual shortest path.

- ***Input***: Sequence A of numbers
- ***Output***: Length of a LIS of A

1.3 Subproblem to Solve

$(\star)L(i)$ is the length of LIS of A ending at position i (similar to S).

$$(\dagger)L(i) = \begin{cases} 1 + max\{L(j) : j < i \cap a_j < a_i\} & \text{if } \exists j < i \text{ s.t. } a_j < a_i \\ 1 & \text{otherwise} \end{cases}$$

Where $max(\emptyset) = 0$, and if there is no element before i, then we would only have a_i, which has length 1.

Note that $(\star)$ is the definition of the subproblem that we want to solve, and $(\dagger)$ is the recursive formula to compute $(\star)$, which we know is right because of our previous analysis that led to the optimal substructure of the LIS.

Formally, we would use induction to prove that $(\dagger)$ leads to $(\star)$. And in the induction step, we would use our optimal substructure to prove the correctness. We **do not** need to prove using induction, just argue by optimal substructure.

Another important aspect of DP is that we have to try all substructures out and pick out the best (in this case, we pick out the max length of a LIS that ends strictly before i).

1.4 Algorithm-Length

```
1: procedure LIS-Length(A)
2:     for (i = 1 → n) do
3:         L(i) ← 1                                  ▷ Initial assumption
4:         for (j = 1 → i − 1) do                    ▷ Search for max in substructure
5:             if (A[j] < A[i] and L[i] < L[j] + 1) then
6:                 L[i] = L[j] + 1                   ▷ Update assumption
7:     m = argmax(L)                                 ▷ Find index of element with max length
8:     return L[m]
```

Note that we initially consider that $L(i)$ for any i is 1, but we search in our substructure for the maximum as per the process in $(\dagger)$. We are working in the **opposite direction** (bottom of the recursion). By line 5, if we find that the length of LIS until i is shorter than the length of LIS until j plus 1 (for extending subsequence until j), then we extend $L(i)$ by 1 in line 6. We finally return the maximum $L(i)$ for $1 \leq i \leq n$.

1.5 Running Time-Length

$O(n^2)$ from double nested for loop.

1.6 Final Algorithm (One way out of many)

```
1:  procedure LIS(A)
2:      for (i = 1 → n) do
3:          L(i) ← 1                                   ▷ Initial assumption
4:          pre(i) = −1                                ▷ Predecessor: Non-existing index
5:          for (j = 1 → i − 1) do                     ▷ Search for max in substructure
6:              if (A[j] < A[i] and L[i] < L[j] + 1) then
7:                  L[i] = L[j] + 1                    ▷ Update assumption
8:                  pre(i) ← j                         ▷ Update predecessor
9:      m = argmax(L)                                  ▷ Find index of element with max length
10:     i ← m                                          ▷ L[m] is length of LIS of A
11:     while (i ≠ −1) do
12:         S[L[i]] = A[i]
13:         i ← pre(i)                                 ▷ Go to predecessor
14:     return S
```

Another way would be to not use $pre(i)$, After assigning the last element m in sequence S, simply look backwards for an element with length $L[m] - 1$ and so on.

2 Ingredients of DP Algorithm

1. Define subproblems to solve, which we will call ($\star$)
2. Give a recursive formula that computes ($\star$), which we will call ($\dagger$), also known as **Bellman equation**
3. Solve original problem using subproblems

3 Similarities between DP and Greedy Algorithm and DC problems

Greedy algorithms also usually exhibit the same characteristic that optimal solutions are extended from smaller substructures. However, in greey algorithms we usually go one way from a substructure whereas in DP, we have to determine which subproblem to pick in order to extend the subsolution to optimal

Usually to solve a DP problem, we have to change the problem (dynamic) whereas in DC problems we do not. Also, in DC problems, we shrink the size of problem by a constant factor whereas in DP we can shrink by a constant amount (linear recursion depth).

4 Problem without Optimal Substructure

Suppose to want to go from A to B, the cheapest flight is from $A \to C \to D \to B$. Maybe there is another flight from $A \to E \to D \to B$ which is not the cheapest flight from A to B, but might be the cheapest flight from A to D. Then if we consider the optimal flight from A to B, the substructure of this flight is no longer optimal (We have a cheaper flight from A to D that is not part of the original optimal route). In other words, the cheapest flight from A to B may not be an extension of the cheapest flight from A to D.