
Dynamic programming is one of the most useful algorithmic techniques. We use an example problem, longest increasing subsequence (LIS), to illustrate the idea.

Given a list of $n$ numbers (i.e. sequence, denoted as $A$), we define subsequence as a sub-list of the numbers (with the original order). For example, let the sequence being 4; 2; 7; 5; 3; 5; 9; 6. A subsequence is 2, 5, 4, 6. We call a subsequence an increasing subsequence if this subsequence consists of numbers in the increasing order. Example of increasing subsequence of the above example is 4, 7, 9. The LIS problem is to find the length of LIS. We first study how to compute the length of LIS. In this example, the length of LIS is 4.

Let us define $L(i)$ as the length of a LIS that ends at position $i$ (position means index of the array). Then, the length of the LIS of the whole list is simply $\max_{i=1}^n L(i)$.

To see how to compute $L(i)$, we consider a LIS ending at $i$. Its previous number should be some at some $j < i$, where $A[j] < A[i]$. But which one is $j$ if there are multiple such $j$? Let $j$ be the position immediately preceding $i$ in the LIS ending at $i$. Then we know $L(i) = L(j) + 1$. This is because since the LIS uses $j$-th number, the it must take the LIS ending at $j$ (otherwise it contradicts the assumption of the LIS ending at $i$). Since we do not know which $j$ to use, we take the best: $L(i) = \max_{j<i, A[j]<A[i]} L(j) + 1$.

Now to compute a $L(i)$, we can use the previous recursion: it will then recursively compute $L(j)$. This works, but with a great cost. The problem is, many subproblems $L(j)$ are computed multiple times. To see how serious the problem is, we consider the worst case for computing $L(n)$. Let $S(i)$ be the number of recursion calls (i.e. nodes in recursion tree) for computing $L(i)$. Then, in the worst case,

$$S(n) = 1 + \sum_{i=1}^{n-1} S(i)$$

We observe $S(1) = 1, S(2) = 2, S(3) = 4$. Then we guess: $S(i) = 2^{i-1}$, and this is easily provable using induction. So, $S(n) = 2^{n-1}$, which implies that the recursion based algorithm runs in exponential time. Now we apply the dynamic programming that will lead to a polynomial time algorithm. The key idea is to memorize the subproblem solutions so that we do not re-compute them, using a bottom-up approach.

```
1: for i = 1 to n do
2:   L[i] = 1
3:   for j = 1 to i - 1 do
5:       L[i] = 1 + L[j]
6:     end if
7: end for
8: end for
9: return $\max_{i=1}^n L[i]$.
```

Note: when computing $L(i)$, $L(j)$ has already been computed. And for each $L(i)$, it gets updated just once. Running time: there are $n$ iterations, each iteration taking at most $O(n)$ time. So the total time is $O(n^2)$.

The only remaining issue is that we only know the LIS’s length, but what is LIS itself? That is not so hard. We suppose the LIS ending at node $p$ (i.e. we know the last letter of the LIS, $A[p]$). To find the one before $A[p]$, we perform trace back. In other words, we check which position $j$ that leading to the current value at $L[p]$; then we move to this $j$. We do this recursively (each time figure out its immediate previous node). I suggest you to work out a few examples to better understand this.

Lecture 15: Dynamic Programming, continued

In this lecture, we continue our discussion of dynamic programming. We discussed two problems: 0-1 Knapsack and the longest common subsequence (LCS). The LCS problem is well explained in the textbook, and I will skip it here. Read the section in the textbook if you have questions about the LCS problem.
Recall that in the 0-1 Knapsack problem, we have \( n \) items, each weight \( w_i \) (which is an integer) pound and worth \( v_i \) dollars. The goal is find a subset of items as valuable as possible but no more than \( W \) pounds (here \( W \) is a given integer). For each item, we must either take or not take and thus is called 0-1 Knapsack.

Suppose the optimal solution contains \( k \) items, \( b_1, b_2, \ldots, b_k \). Suppose we remove \( b_k \), then we have the following property: the remaining items, \( b_1, b_2, \ldots, b_{k-1} \) must the most valuable items under the constraints that total weight is no more than \( W - w_{b_k} \). Otherwise, we have a contradiction that \( b_1, b_2, \ldots, b_k \) are the most valuable items with no more than \( W \) total weight.

Now we define \( M[i, w] \) as the highest value (in dollar) when we choose items from 1 to \( i \), and total weight is no more than \( w \). Clearly, the solution to our Knapsack problem is simply \( M[n, W] \). It is easy to see that \( M[0, w] = 0 \), and \( M[i, 0] = 0 \) for all \( i \). When \( i \geq 1 \) and \( w > 0 \), we have: \( M[i, w] = \max(M[i-1, w-w_i] + v_i, M[i-1, w]) \) if \( w_i < w \), and otherwise \( M[i, w] = M[i-1, w] \). This is because we either take item \( i \) or not. If we take item \( i \), then the remaining capacity of Knapsack is reduced by the weight of item \( i \). If not, the capacity remains unchanged. In either case, we should pack as valuable as we can using the remaining items.

```plaintext
1: for w = 0 to W do
2:    M[0, w] = 0
3: end for
4: for i = 1 to n do
5:    M[i, 0] = 0
6:    for w = 1 to W do
7:        if \( w_i \leq w \) then
8:            M[i, w] = \max(M[i-1, w-w_i] + v_i, M[i-1, w])
9:        else
10:           M[i, w] = M[i-1, w]
11:       end if
12:    end for
13: end for
14: return M[n, W].
```

Running time: we have \( O(nW) \) cells in \( M \) array, each taking \( O(1) \) to compute. So the total time is \( O(nW) \). Is this a polynomial-time algorithm? Not quite but it works well when \( W \) is relatively small.