CS2223: Algorithms
Dynamic Programming

1 When can we use Dynamic Programming and when is it useful

Dynamic programming can be used to obtain an algorithm to an optimization problem when it exhibits **optimal substructure** property – the optimal solution to a problem consists of optimal solutions to sub-problems.

For example, consider that we need to find the shortest path between two nodes in a graph. This exhibits optimal substructure property – suppose the shortest path between nodes (i, j) passes through a node k. The path (i, k) must be the shortest path from i to k.

A problem that does not exhibit optimal substructure property is finding the longest path between two nodes in a graph. Suppose we need to find the longest path between nodes i and j. Let the longest path pass through node k. The path from i to k which appears in the optimal solution might not be the longest path from i to k (because we need no node repeats in a path).

Dynamic programming is useful when the number of sub-problems to be solved is small – typically polynomial in input size. A recursive algorithm might visit the same sub-problem over and over again, in which case we say that the optimization problem has **overlapping subproblems**. Dynamic programming is efficient in such a case, because it keeps track of solutions for all sub-problems and hence does not solve the same sub-problem more than once.

2 Longest Increasing Subsequence

Given a sequence $A =< a_1, a_2, \ldots, a_n >$, a subsequence is $< a_{i_1}, a_{i_2}, \ldots, a_{i_k} >$, such that $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. Subsequence is increasing if $ai_1 < a_{i_2} < \ldots < a_{i_k}$. You need to find the longest increasing subsequence in a given sequence.

Example: longest increasing subsequence of $< 5, 2, 8, 1, 3, 6, 9, 7 >$ is $< 2, 3, 6, 9 >$.

Solution: We use an array $X[1..n]$. Let x_j denote the longest increasing subsequence of $< a_1, a_2, \ldots, a_j >$ that includes the element a_j. x_j can be computed as follows.

$$x_j = \begin{cases} 1 & \text{if } a_k > a_j, \forall 1 \leq k < j \\ max_{1 \leq k < j, a_k < a_j}\{x_k + 1\} & \text{otherwise} \end{cases}$$

After computing all x_js, we can obtain the length of the longest increasing subsequence by examining all the x_js and finding the largest one. If we also want to find the longest increasing subsequence, in addition to the x_js, store the previous element in the longest increasing subsequence (i.e., the k which gives the longest increasing subsequence for j).

For example, the x_js stored are as follows.
$a_j = 5\ 2\ 8\ 1\ 3\ 6\ 9\ 7$
$x_j = 1\ 1\ 2\ 1\ 2\ 3\ 4\ 4$

Finding x_j can be done in one scan over all the previous x_ks. Once all the x_js are found, we need to do one final scan. Therefore the complexity is given by $1 + 2 + \ldots + (n-1) + n = \Theta(n^2)$.

3 Edit Distance between Two Strings

Given two strings $x[1..m]$ and $y[1..n]$, what is the distance between them? (or) What is the cost of transforming string x to string y, when the operations allowed are – insert a character (cost=1), delete a character (cost=1), replace a character in x with another character in y (cost=1), match a character in x with the same character in y (cost=0).

Note that the problem exhibits optimal substructure property.

If the optimum cost of transforming x to y performs insert $y[n]$ at the last step, then the optimum solution is the (optimum cost of matching $x[1..m]$ and $y[1..(n-1)]$) + 1.

Similarly if the optimum cost of transforming x to y performs delete $x[m]$ at the last step, then the optimum solution is the (optimum cost of matching $x[1..(m-1)]$ and $y[1..n]$) + 1.

Similarly if the optimum cost of transforming x to y performs replacing character $x[m]$ with character $y[n]$ at the last step, then the optimum solution is the (optimum cost of matching $x[1..(m-1)]$ and $y[1..(n-1)]$) + 1.

Similarly if the optimum cost of transforming x to y performs matching character $x[m]$ with character $y[n]$ at the last step, then the optimum solution is the (optimum cost of matching $x[1..(m-1)]$ and $y[1..(n-1)]$) + 0.

Solution: We use a matrix $A[(0..m)*(0..n)]$, where $A[i,j]$ denotes the optimum cost for transforming $x[1..i]$ to $y[1..j]$, and can be computed as follows.

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

Complexity: $A[i,j]$ can be computed in $\Theta(1)$. Therefore the complexity is $\Theta(mn)$.

4 Longest Common Subsequence

Given two strings $X = x[1..m]$ and $Y = y[1..n]$, find the longest subsequence that occurs in both strings.

Note that this problem exhibits the optimal substructure property. Let $Z = z[1..k]$ be a longest common subsequence between X and Y. The following is true.

- if $x[m] = y[n]$, then $z[k] = x[m] = y[n]$ and $Z[1..(k-1)]$ is an LCS of $X[1..(m-1)]$ and $Y[1..(n-1)]$.
- if $x[m] \neq y[n]$, then $z[k] \neq x[m]$ implies Z is LCS of $x[1..(m-1)]$ and Y.
- if $x[m] \neq y[n]$, then $z[k] \neq y[n]$ implies Z is LCS of X and $y[1..(n-1)]$.

We can therefore solve the LCS problem as follows. Define $A[0..m, 0..n]$. $A[i, j]$ denotes the length of the LCS of $x[0..i]$ and $y[0..j]$, and is defined as follows.

$$A[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ A[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ max(A[i-1, j], A[1, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

5 0-1 Knapsack Problem

A thief robbing a store finds n items. Item i is worth v_i and weighs w_i. He has a a knapsack that can hold atmost W weight. Which items must he take?

This problem exhibits the optimal substructure property. If the optimal solution includes item n, then all items in the optimal solution except item n will be the optimal solution for the knapsack problem where we consider items $1..(n-1)$ and weight of the knapsack is atmost $W - w_n$. If the optimal solution does not include item n, then this optimal solution is also the optimal solution for the knapsack problem where we consider items $1..(n-1)$ and weight of the knapsack is atmost W.

We define $A[0..n, 0..W]$, where $A[i, j]$ denotes using items $1, 2, \ldots i$ and where the knapsack can hold atmost weight j. This is computed as follows.

$$A[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ max(A[i-1, j], A[i-1, j-w_i]) & \text{otherwise} \end{cases}$$

6 Matrix Multiplication

You are given n matrices, say $M_1, M_2, \ldots, M_n$ where the size of M_i is given by $s_i * s_{i+1}$. Note the following – matrices A and B can be multiplied if size of A is $s_1 * s_2$, and the size of B is $s_2 * s_3$. The size of the result is $s_1 * s_3$, and the complexity of multiplication is $s_1 * s_2 * s_3$.

You need to find the optimal order in which these matrices should be multiplied.

Note that this problem exhibits optimal substructure property – if the optimal way of multiplying these n matrices is $(M_1 * M_2 * \ldots * M_i) * (M_{i+1} *$

$M_{i+2} * \ldots * M_n)$, then it must in turn contain the optimal way of multiplying $(M_1 * M_2 * \ldots * M_i)$ and the optimal way of multiplying $(M_{i+1} * M_{i+2} * \ldots * M_n)$.

We define $A[1..n, 1..n]$, where $A[i, j]$ denotes the cost of the best way of multiplying matrices $A[i, i+1, \ldots j]$. Note that $A[i, j]$ is not defined when $j < i$. We compute as follows.

$$A[i, j] = \begin{cases} \text{undefined} & \text{if } i < j \\ 0 & \text{if } i = j \\ min_{1 \leq k < j}\{A[i, k] + A[k+1, j] + s_i * s_{k+1} * s_{j+1}\} & \text{otherwise} \end{cases}$$