Dynamic Programming

Hoa T. Vu *

1 Introduction

The main ideas behind dynamic programming are as follows. Dynamic programming is basically smart recursion. One may think of dynamic programming as recursion with memoization.

2 First example: Fibonacci sequence

The Fibonacci sequence is defined as follows. $F_0 = 0, F_1 = 1$. For $i > 1$, $F_i = F_{i-1} + F_{i-2}$. So the sequence goes like this 0, 1, 1, 2, 3, 5, 8, 13, ...

This suggests a recursive algorithm

Algorithm 1: Recursive Fibonacci

Function *Fibonacci(i)*
  if $i = 0$ **then** return 0
  else if $i = 1$ **then** return 1
  else Return $Fibonacci(i-1) + Fibonacci(i-2)$

The running time of this algorithms is

$$T(n) = T(n-1) + T(n-2) + O(1) \leq 2T(n-1) + c$$

Exercise: Use recursion tree method to show that the running time is $\Omega(2^n)$.

Why is the above algorithm wasteful? That's because there is a lot of repeated computation.

*San Diego State University, hvu2@sdsu.edu

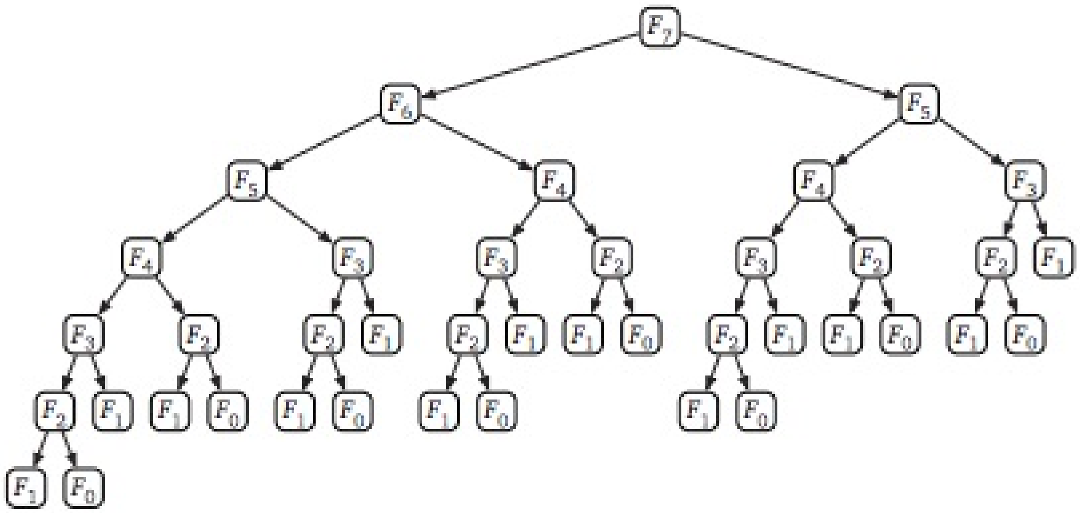


Figure 3.1. The recursion tree for computing F_7; arrows represent recursive calls.

So, we try to avoid repeated computation by checking if the computation has been done before.

Algorithm 2: Dynamic Programming (top down) for computing the nth Fibonacci number

---

Initialize $F[1 \dots n]$ where each entry is NULL.
Function *ComputeFibonacci(i)*
    if $i = 0$ **then** return 0
    else if $i = 1$ **then** return 1
    else **if** $F[i-1] \neq NULL$ **then** ComputeFibonacci($i-1$)
    if $F[i-2] \neq NULL$ **then** ComputeFibonacci($i-2$)
    Return $F[i-1] + F[i-2]$
Call ComputeFibonacci(n).

The above approach is call top-down. We can also fill the array F bottom up deliberately.

Algorithm 3: Dynamic Programming (bottom up) for computing the nth Fibonacci number

---

Initialize $F[1 \dots n]$ where each entry is NULL.
F[0] = 0, F[1] = 1.
for $i = 2, 3, \dots, n$ **do**
    $F[i] = F[i-1] + F[i-2]$.
end for

The running time is $O(n)$ which is a great improvement over naive recursion.

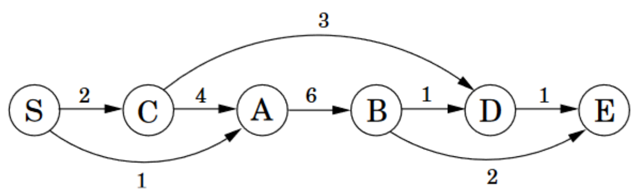
3 Shortest path in Directed Acyclic Graphs

We will talk more about graphs and related terminology later. For now, think of a graph as a set of vertices/nodes V and a set of edges E that connecting them. A directed graph is a graph where each edge has a direction. Often we use $n = |V|$ and $m = |E|$.

We consider a directed graph in topological order. For simplicity, assume we have vertices $v_1, v_2, \dots, v_n$ and directed edges in the graph only go from v_i to v_j where $i < j$. Note that this graph has no cycle (why?).

We are also given an array ℓ where $\ell[v_i, v_j]$ is the length of edges $v_i \to v_j$. Furthermore, $incoming(v_i)$ is the list of all vertices that have an edge going toward v_i.

For example, see the figure below.



The goal is to compute $dist[1 \dots n]$ where $dist[i]$ is the length of the shortest path from v_1 to v_i.

- Clearly, $dist[1] = 0$.
- Consider any node v_i. The shortest path from v_1 to v_i (denoted by $v_1 \rightsquigarrow v_i$) must go through some v_i predecessor v_j (ie, $j < i$ where there is an edge $v_j \to v_i$) and then then go from v_j to v_i using the edge $v_j \to v_i$ (it is possible that $v_j = v_1$ if there is an edge from v_1 to v_i).
- If $v_1 \rightsquigarrow v_i$ goes through v_j before going from v_j to v_i using the edge $v_j \to v_i$, then the length of that path would be the length of the shortest path $v_1 \rightsquigarrow v_j$ plus the length of the edge $v_j \to v_i$.
- Thus,
$$dist[v_i] = \min_{j<i: v_j \to v_i \in E} (dist[v_j] + \ell(v_j, v_i)).$$

The formal algorithm is as follows.

Algorithm 4: Dynamic programming for shortest path in directed acyclic graphs

1 Initialize $dist[1 \dots n]$ where each entry is ∞.
2 Initialize $Z[1 \dots n]$ where each entry is NULL. This array is used to keep track of the actual path (we will usually omit this for future examples).
3 $dist[1] = 0$.
4 **for** $i = 2, 3, \dots, n$ **do**
5     $min = \infty$
6   **for** $v_j \in incoming(v_i)$ **do**
7   **if** $dist[v_j] + \ell(v_j, v_i) < min$ **then** $min = dist[v_j] + \ell(v_j, v_i)$ and $Z[v_i] = v_j$;
8   **end for**
9 **end for**

Lines 3, 4 executes $O(|V|)$ times. Lines 5, 6 are executed $O(|E|)$ times (ie, once for each edge). Hence, the running time is $O(|V| + |E|)$.

4 Longest Increasing Subsequence

Given an array $A[1 \dots n]$ what is the longest increasing subsequence (LIS) in A?

Motivation: we may want to test how sorted a database is. If the length of the LIS is close to n, then A is nearly sorted.

Let us create a graph where node i corresponds to $A[i]$. If $A[i] \leq A[j]$ and $i \leq j$, then create an edge $i \rightarrow j$ with length 1. Otherwise, $\ell(i, j) = NULL$. Note that this graph is also directed acyclic.

Clearly, an increasing subsequence corresponds to a directed path. Hence, the goal is is to find the longest directed path in the graph we created.

Let $L[i]$ be the length of the longest path ending at vertex i plus 1 (ie, this corresponds to the length of the longest increasing subsequence ending at $A[i]$). Again,

- $L[1] = 1$.
- Consider any node i. The longest path ending at i must go through some v_i predecessor v_j (ie, $j < i$ where there is an edge $v_j \rightarrow v_i$) and then then go from v_j to v_i using the edge $v_j \rightarrow v_i$. If i has no predecessor, then the longest path ending at i is just i itself.
- Thus,

$$L[i] = \begin{cases} \max_{j<i}(L[j] + 1) & \text{if there is some } j \text{ s.t } A[j] \leq A[j] \\ 1 & \text{otherwise} \end{cases}$$

Algorithm 5: Dynamic programming for longest increasing subsequence

Initialize $L[1 \ldots n]$ where each entry is 1.
$L[1] = 1$.
for $i = 2, 3, \ldots, n$ **do**
    $max = 1$
    for $j = 1, 2, \ldots, i - 1$ **do**
        if $A[j] \leq A[i]$ *and* $L[j] + 1 > max$ **then**
            $max = L[j] + 1$
        end if
    end for
end for

5 Common theme

There are a few things that we should consider when designing a dynamic programming algorithm.

- Define the dynamic programming table.
- What are the border (base) cases?
- How to fill the table?

6 Edit distance

We are given 2 input strings A and B of length n and m respectively. What is the minimum number of insertions, deletions, and replacements to transform A into B?

Applications: Comparing documents or DNA sequences.

Example 1. Consider $A = SNOWY$ and $B = SUNNY$. Think of this process as placing gap into 2 strings to align them together.

For example, the below corresponds to 1) insert U between S and N, 2) Replace O with N, and 3) Delete W. There are 3 edits in total.

$$\begin{array}{c|c|c|c|c|c} \mathrm{S} & \_ & \mathrm{N} & \mathrm{O} & \mathrm{W} & \mathrm{Y} \\ \mathrm{S} & \mathrm{U} & \mathrm{N} & \mathrm{N} & \_ & \mathrm{Y} \end{array}$$

Example 2. Let's consider the same 2 strings, but a different set of edits. The below corresponds to 1) insert S in the beginning 2) Replace S with U, and 3) Delete O, 4) Delete W, 5) insert N. There are 5 edits in total.

$$\begin{array}{c|c|c|c|c|c|c} \_ & \mathrm{S} & \mathrm{N} & \mathrm{O} & \mathrm{W} & \_ & \mathrm{Y} \\ \mathrm{S} & \mathrm{U} & \mathrm{N} & \_ & \_ & \mathrm{N} & \mathrm{Y} \end{array}$$

The question is to place the gaps that correspond to the smallest number of edits which is the edit distance between A and B.

A DP approach. We define the dynamic programming table as follows.

$$ED[i,j] = \text{edit distance between } A[1 \ldots i] \text{ and } B[1 \ldots j].$$

Consider the optimal gap placing. There are 3 cases for the last column:

- Case 1: deleting $A[i]$ at the end

  $$\begin{array}{c|c|c|} \ldots & \ldots & A[i] \\ \ldots & \ldots & \_ \end{array}$$

  The cost would be $ED[i-1,j]+1$. This corresponds to first transform $A[1 \ldots i-1]$ to $B[1 \ldots j]$ and then delete $A[i]$ at the end.

- Case 2: inserting $B[j]$ at the end

  $$\begin{array}{c|c|c|} \ldots & \ldots & \_ \\ \ldots & \ldots & B[j] \end{array}$$

  The cost would be $ED[i,j-1]+1$. This corresponds to first transform $A[1 \ldots i]$ to $B[1 \ldots j-1]$ and then insert $B[j]$ at the end.

  $$\begin{array}{c|c|c|} \ldots & \ldots & A[i] \\ \ldots & \ldots & B[j] \end{array}$$

  The cost would be $ED[i-1,j-1] + \begin{cases} 1 & \text{if } A[i] \neq B[j] \\ 0 & \text{if } A[i] = B[j] \end{cases}$.

  This corresponds to first transform $A[1 \ldots i-1]$ to $B[1 \ldots j-1]$ and then replace $A[i]$ with $B[j]$ at the end if $A[i] \neq B[j]$.

Let $diff(i,j) = \begin{cases} 1 & \text{if } A[i] \neq B[j] \\ 0 & \text{if } A[i] = B[j] \end{cases}$.
We have the following recursive relationship

$$ED[i,j] = \min\{ED[i-1,j]+1, ED[i,j-1]+1, ED[i-1,j-1]+diff(i,j)\}.$$

Base case $ED[i,0] = i$ and $ED[0,j] = j$ (Why?).

Algorithm 6: Dynamic programming for Edit distance

For each $i = 0, 1, 2, \dots, n$: $ED[i,0] = i$
For each $j = 0, 1, 2, \dots, m$: $ED[0,j] = j$.
for $i = 1, 3, \dots, n$ **do**
    for $j = 1, 2, \dots, n$ **do**
        $ED[i,j] = \min\{ED[i-1,j] + 1, ED[i,j-1] + 1, ED[i-1,j-1] + diff(i,j)\}$.
    end for
end for
Return $ED[n,m]$.

7 Knapsack

Let us assume that we have n items (without repetition) each of which has a weight $w[i]$ and a value $v[i]$. We are allowed to carry at most W unit of weight in total. Assume weights and values are all integers, how to pick items to carry such that we maximize the value?

Applications: This arises in various problems where values correspond to utilities and B corresponds to some resource constraints.

The greedy algorithm that picks items with highest value/weight fails to find the best solution. For example, consider the input $W = [3, 2, 2], V = [1.65, 1, 1], B = 4$.

Let $T[i,j]$ be the maximum value that we can carry using j unit of weights from items $1, \dots, i$. The optimal solution has to either we pick item i or we don't.

If we pick item i, the largest total value we could get is $T[i-1, j - w[i]] + V[i]$. If we do not pick item i, the largest total value we could get is $T[i-1,j]$.

Algorithm 7: Dynamic programming for Knapsack without repetition

For each $i = 0, 1, 2, \dots, n$: $T[i,0] = 0$
For each $j = 0, 1, 2, \dots, B$: $T[0,j] = 0$.
for $i = 1, 3, \dots, n$ **do**
    for $j = 1, 2, \dots, B$ **do**
        if $W[i] \le j$ **then**
            $T[i,j] = \max\{T[i-1, j - W[i]] + V[i], T[i-1,j]\}$.
        else
            $T[i,j] = T[i-1,j]$
        end if
    end for
end for
Return $T[n,B]$.

It is easy to see that the running time is $O(nB)$.

8 Subset Sum

Consider a multiset A of n positive integers (represented as an array $A[1 \dots n]$). We want to output true if there is a subset of A that sum to a target integer T.

Convention: empty subset sums to 0.

Let $S[i, j] = true$ if and only if there is a subset of $A[1 \dots i]$ that sums to j. Suppose there is a subset of A that sum to j. There are 2 cases:

- Case 1: This subset contains $A[i]$ which means there must be a subset of $A[1 \dots i-1]$ that sum to $j - A[i]$. Therefore, if this is the case, then $S[i-1, j-A[i]] = true$.
- Case 1: This subset does not $A[i]$ which means there must be a subset of $A[1 \dots i-1]$ that sum to j. Therefore, if this is the case, then $S[i-1, j] = true$.

Therefore, $S[i, j] = true$ if and only if $S[i, j - A[i]] = true$ or $S[i-1, j] = true$.

Algorithm 8: Dynamic programming for subset sum

Corner cases:
$S[0, 0] = true$
$S[0, j] = false$ for all $j = 1, 2, \dots, T$
for $i = 1, 3, \dots, n$ **do**
    for $j = 1, 2, \dots, T$ **do**
        if $A[i] \leq j$ **then**
            $S[i, j] = S[i-1, j-A[i]] \vee S[i-1, j] = true$
        else
            $S[i, j] = S[i-1, j] = true$
        end if
    end for
end for
Return $T[n, T]$.

It is easy to see that the running time is $O(nT)$.

9 Longest Palindromic Subsequence

A palindrome is a string that is the same as its reverse, e.g., racecar, tenet.

Input: A string $A[1 \dots n]$. Output: length of the longest palindromic subsequence (LPS).

Example: ARACABERCKAR. The output should be 7 since the longest palindromic subsequence is RACECAR.

Let $T[i, j]$ be the length of the longest palindromic subsequence of $A[i \dots j]$.

Corner cases: $T[i, i] = 1$ and $T[i, j] = 0$ for $i > j$.

We have 3 cases:

- Case 1: the LPS of $A[i \dots j]$ contains both $A[i]$ and $A[j]$ (only possible if $A[i] = A[j]$), then $T[i, j] = 2 + T[i+1, j-1]$.
- Case 2: the LPS of $A[i \dots j]$ does not contain $A[i]$, then $T[i, j] = T[i+1, j]$.
- Case 3: the LPS of $A[i \dots j]$ does not contain $A[j]$, then $T[i, j] = T[i, j-1]$.

This problem is simpler if we use top-down DP. The algorithm is as follows.

Algorithm 9: Dynamic Programming (top down) for LPS

Initialize $T[1 \dots n][1 \dots n]$ where each entry is NULL.
Function $F(i, j)$
  if $i = j$ **then** $T[i, j] = 1$
  if $i > j$ **then** $T[i, j] = 0$
  if $i < j$ **then**
    If $T[i+1, j-1] = NULL$, Call $F(i+1, j-1)$. If $T[i, j-1] = NULL$, Call $F(i, j-1)$. If $T[i+1, j] = NULL$, Call $F(i+1, j)$. **if** $A[i] = A[j]$ **then**
      $T[i, j] = \max\{2 + T[i+1, j-1], T[i+1, j], T[i, j-1]\}$
    else
      $T[i, j] = \max\{T[i+1, j], T[i, j-1]\}$
    end if
  end if
Return $F[1, n]$.

The running time is $O(n^2)$ since each $T[i, j]$ is called at most once and the non-recursive work is $O(1)$.

Exercise: How to implement this bottom-up?