Dynamic Programming

Topics

1. Introduction
2. Problems & Solutions
3. Important References

Introduction

Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions – ideally, using a memory-based data structure. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space.

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not guarantee an optimal solution, because picking locally optimal choices may result in a bad global solution, but it is often faster to calculate. Fortunately, some greedy algorithms are proven to lead to the optimal solution.

Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:

1. **Overlapping Subproblems**
2. **Optimal Substructure**

(A) Overlapping Subproblems

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point in storing the solutions if they are not needed again. For example, *Binary Search* doesn't have common subproblems. However, if we take an example of the following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.
Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21…

```
// simple recursive program for finding nth Fibonacci number
int fib(int n) {
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

There are two different ways to store the values so that these values can be reused.

1. **Memoization (Top Down):** The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

```c
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];
// Function to initialize NIL values in lookup table
void _initialize() {
    for (int i = 0; i < MAX; i++)
        lookup[i] = NIL;
}

// Function for nth Fibonacci number
int fib(int n) {
    if (lookup[n] == NIL) {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n-1) + fib(n-2);
    }
    return lookup[n];
}

int main () {
    int n = 40;
    _initialize();
    printf("Fibonacci number is %d\n", fib(n));
    return 0;
}
```

2. **Tabulation (Bottom Up):** The tabulated program for a given problem builds a table in ***bottom up*** fashion and returns the last entry from table.

```c
#include<stdio.h>
int fib(int n) {
    int f[n+1];
    f[0] = 0;   f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];

    return f[n];
}

int main () {
    int n = 40;
    printf("Fibonacci number is %d\n",fib(n));
    return 0;
}
```

Both **Tabulated** and **Memoized** methods store the solutions of subproblems. In Memoized version, table is filled on demand while in tabulated version, starting from the first entry, all entries are filled one by one. Unlike the tabulated version, all entries of the lookup table are not necessarily filled in memoized version.

(B) Optimal Substructure

A given problem has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Consider finding a shortest path for travelling between two cities by car. Such an example is likely to exhibit optimal substructure. That is, if the shortest route from Seattle to Los Angeles passes through Portland and then Sacramento, then the shortest route from Portland to Los Angeles must pass through Sacramento too. That is, the problem of how to get from Portland to Los Angeles is nested inside the problem of how to get from Seattle to Los Angeles.

As an example of a problem that is unlikely to exhibit optimal substructure, consider the problem of finding the cheapest airline ticket from Buenos Aires to Kiev. Even if that ticket involves stops in Miami and then London, we can't conclude that the cheapest ticket from Miami to Kiev stops in London, because the price at which an airline sells a multi-flight trip is usually not the sum of the prices at which it would sell the individual flights in the trip.

Definition

A slightly more formal definition of optimal substructure can be given as follows:
Let a "problem" be a collection of "alternatives", and let each alternative have an associated cost, **c(a)**. The task is to find a set of alternatives that *minimizes* c(a). Suppose that the alternatives can be partitioned into subsets, where each subset has its own cost function, and each alternative belongs to only one subset. The minima of each of these cost functions can be found, as can the minima of the global cost function, restricted to the same subsets. If these minima match for each subset, then it's almost obvious that a global minimum can be picked, not out of the full set of alternatives, but only out of the set that consists of the minima of the smaller local cost functions we have defined. If minimizing the local functions is a problem of "lower order", and (specifically) if, after a finite number of these reductions, the problem becomes trivial, then the problem has an optimal substructure.

Problems & Solutions

1. Fibonacci Series and its application:
    a. Count ways to reach the n^{th} stair if one or two steps can be taken at once
    Solution:
    DP State: dp[i] = Number of ways to reach the i^{th} stair.

```
dp[1] = 1
dp[2] = 2
dp[i] = dp[i-1] + dp[i-2]; { ∀i : 3 <= i <= n }
```

    Ans: dp[n]

    b. Domino problem:
        i. Number of ways to fill $2 \times N$ sized rectangle with dominos of size 1×2 [Dominos can be rotated to 2×1]
        Solution:
        DP State: dp[i] = No. of ways to fill $2 \times i$ with 2×1 sized dominos.

```
dp[1] = 1
dp[2] = 2
dp[i] = dp[i-1] + dp[i-2]; { ∀i : 3 <= i <= n }
```

        Ans: dp[n]

        ii. Number of ways to fill $5 \times N$ sized rectangle with dominos of size 1×5 [Dominos can be rotated to 5×1]
        Solution:
        DP State: dp[i] = No. of ways to fill $5 \times i$ with 5×1 sized dominos.

```
dp[1] = dp[2] = dp[3] = dp[4] = 1
dp[5] = 2
dp[i] = dp[i-1] + dp[i-5]; { ∀i : 6 <= i <= n }
```

        Ans: dp[n]

        iii. Number of ways to fill $K \times N$ sized rectangle with dominos of size $1 \times K$ [Dominos can be rotated to $K \times 1$]
        Solution:
        DP State: dp[i] = No. of ways to fill $K \times i$ with $K \times 1$ sized dominos.

```
dp[i] = 1; { ∀i : 1 <= i < K }
dp[i] = 2; i = K
dp[i] = dp[i-1] + dp[i-K]; { ∀i : K+1 <= i <= n }
```

        Ans: dp[n]

2. Compute ${}^{n}C_{r}$ for given values of `n` and `r` (Number of ways to ***choose*** `r` items from `n` items)
   Solution:
   DP State: `dp[i][j]` = Number of ways to choose j items from i items.
   There are two possibilities:
   a. **Including i^{th} item : `dp[i-1][j-1]`**
   b. **Excluding i^{th} item : `dp[i-1][j]`**
      So we can write `dp[i][j]` as following:

```
dp[0][j] = 0; { ∀j : 1 <= j <= r }
dp[i][0] = dp[i][i] = 1, { ∀i : 0 <= i <= n }
for 2 <= i <= n
    for 1 <= j <= min(r, i-1)
        dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
```

   Ans: `dp[n][r]`

3. **Integer Knapsack:** A thief robbing a store finds n items; the i^{th} item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take?
   Solution:
   v : Array of *values* ; `v[i]` : denotes the value of i^{th} item
   w: Array of *weights* ; `w[i]` : denotes the weight of i^{th} item
   DP State: `dp[i][j]` = Max profit after processing i items with knapsack size j.

```
dp[0][j] = 0; { ∀j : 1 <= j <= W }
dp[i][0] = 0; { ∀i : 1 <= i <= n }

for 1 <= i <= n:  //Assume that item array indexing starts from 1
    for 1 <= j <= W:
        if(j-w[i] <  0):
            dp[i][j] = dp[i-1][j]
        else:
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])
```

   Ans: `dp[n][W]`

4. **Longest Increasing Sequence:** Given an array of integers, find the length of the longest increasing subsequence.
   Solution:
   DP State: `dp[i]` = Length of longest increasing subsequence with i^{th} element as the last element of the sequence.

```
for 0 <= i <= n-1:
    dp[i] = 0
    for 0 <= j < i:
        if (a[j] < a[i]):
            dp[i] = max(dp[i], dp[j])
    dp[i] += 1 // Increasing length by one for the current element
```

   Ans: `max(dp[i]: for 0 <= i <= n-1)`

5. **Maximum contiguous subarray sum:** Given an array of integers, find the subarray with maximum sum, report the sum.
   Solution:
   DP State: `dp[i]` = Max sum after processing first *i* elements of the array with the subarray ending at the i^{th} element.

```
dp[0] = a[0]
dp[i] = max(dp[i-1] + a[i], a[i]); { ∀i : 1 <= i <= n-1 }
```

   Ans: `max(dp[i]: for 0 <= i <= n-1)`

6. **Maximum non-contiguous subsequence sum:** Given an array of integers, find maximum sum of array elements, such that no two chosen elements are at adjacent position of the array.
   Solution:
   DP State: `dp[i]` = Max sum after processing first *i* elements.

```
dp[0] = a[0]
dp[1] = max(a[0], a[1])
dp[i] = max(dp[i-1], dp[i-2] + a[i], a[i]); {∀i : 2 <= i <= n-1}
```

   Ans: `max(dp[i]: for 0 <= i <= n-1)`

7. **Maze with blockers:**
   a. Given a 2D maze of size `N×N`, where few cells are blocked, find the number of ways to reach `[n,n]`, if you start from `[0,0]`, and you can only move rightwards or downwards, *i.e.*, from (i,j) you can go to $(i,j+1)$ or $(i+1,j)$.
      Solution:
      DP State: `dp[i][j]` = Number of ways to reach (i,j) starting from $(0,0)$.

```
dp[0][j] = 1; { ∀j : 0 <= j <= n }
dp[i][0] = 1; { ∀i : 0 <= i <= n }
/* If dp[0][j] (or dp[i][0]) is blocked for any i (or j),
everything will be 0 in that row/column for the remaining
cells */
for 1 <= i <= n:
    for 1 <= j <= n:
        if the cell (i,j) is blocked:
            dp[i][j] = 0
        else:
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
```

      Ans: `dp[n][n]`

   b. If there are no blocked cells, you can directly compute the number of ways as ${}^{2n}C_n$ – This is because you need $2n$ steps to reach `(n,n)` starting from `(0,0)` out of which you have to take n steps *rightwards* and n steps *downward*, so you need to chose n steps out of $2n$ steps.

c. Given a 2D maze where each cell has $\mathbf{A_{ij}}$ number of apples, find the maximum number of apples you can collect on your path to reach the bottom-right corner of the matrix, if you start from (0,0), and you can move only rightwards or downwards, *i.e.*, from (i,j) you can go to $(i,j+1)$ or $(i+1,j)$.

Solution:

DP State: dp[i][j] = Maximum number of apples we can collect in the interim path from (0,0) to (i,j).

```
dp[0][0] = A[0][0]
dp[0][j] = dp[0][j-1] + A[0][j]; { ∀j : 1 <= j <= n-1 }
dp[i][0] = dp[i-1][0] + A[i][0]; { ∀i : 1 <= i <= n-1 }
for 1 <= i <= n-1:
    for 1 <= j <= n-1:
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + A[i][j]
```

Ans: dp[n-1][n-1]

8. **Submatrix sum:** Given a 2D matrix, you have to answer queries of the form: $(i\ j\ k\ l)$, that means, you have to report the sum of the submatrix (i,j) to (k,l) where (i,j) denotes top-left corner of the submatrix and (k,l) denotes bottom-right corner of the submatrix.

   Solution:

   DP State: dp[i][j] = submatrix sum from (0,0) to (i,j)

```
dp[0][0] = A[0][0]
dp[0][j] = dp[0][j-1] + A[0][j]; { ∀j : 1 <= j <= n-1 }
dp[i][0] = dp[i-1][0] + A[i][0]; { ∀i : 1 <= i <= n-1 }
for 1 <= i <= n-1:
  for 1 <= j <= n-1:
    dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i-1][j-1] + A[i][j]
```

   For given query $(i\ j\ k\ l)$, the sum of the submatrix can be computed as:

```
Sum = dp[k][l] - dp[i-1][l] - dp[k][j-1] + dp[i-1][j-1]
```

Important References

1. **Dynamic Programming Practice Problems @ClemsonUniversity**
2. **Dynamic Programming @TopCoder**
3. **Dynamic Programming Lecture on Fibonacci Series & Shortest Paths @MIT-OCW**
4. **Dynamic Programming @Geeks4Geeks**
5. **Dynamic Programming Tutorial @HackerEarth**
6. **Dynamic Programming Blog by Karan2116 @Codeforces**