CS 3510 Algorithms 2/20/2024

Lecture 9: Dynamic Programming

Lecturer: Abrahim Ladha *Scribe(s): Saigautam Bonam, Tejas Pradeep*

1 Dynamic Programming

Dynamic Programming is often done in two ways either top down (with a recurrence and memoization) or bottom up (with iteration). For this class, we will emphasize bottom up. In either case, we solve small subproblems and store them. This will often cause a space time tradeoff. The difference between Divide-and-Conquer algorithms and Dynamic Programming algorithms is that the subproblems in Dynamic Programming overlap, so it's inefficient to recompute all of them. Here's our first DP example:

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

Figure 1: Fibonacci with little space, but large time complexity.

As an example consider the Fibonacci algorithm above. The algorithm takes exponential time with respect to the input. However we can make this faster by using some space and storing intermediate results. Consider using an array and using the former two elements to calculate the current element like in the following algorithm.

```
def fib2(n):
    if n = 0
        return 0

    initialize dp as table of size n + 1 with 0s
    dp[0] = 0
    dp[1] = 1

    for i in 2..(n+1)
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

Figure 2: Fibonacci with some space, but small time complexity.

Essentially every dynamic programming solution involves a memory structure, giving a base case on the memory structure, and filling up that memory structure using a *recurrence* (in this case $dp[i] = dp[i-1] + dp[i-2]$).

2 Example 1

Problem: Suppose you have n stair steps. You can leap one, two, or three steps at a time. What is the number of combinations, or the number of ways, that you could reach step n?

Solution: Create an array $T[0 \ldots n]$ where $T[i]$ represents the number of ways to reach step i.

Our base cases are $T[0] = 1, T[1] = 1$ and $T[2] = 2$. To reach step n, there was some last step. You either jumped 1, 2, or 3 steps, and you jumped from 1, 2, or 3 steps away. That means that the number of ways to go from 0 to n is the sum of the number of ways to go from 0 to $n-1$, 0 to $n-2$, and 0 to $n-3$. This gives us our recurrence of

$$T[i] = T[i-1] + T[i-2] + T[i-3]$$

Implementation: (Pseudocode)

```
def numways(k):
    initialize dp as table of size n + 1 with 0s
    dp[0] = 0
    dp[1] = 1
    dp[2] = 2

    for i in 3..(n+1):
        dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]

    return dp[k]
```

Figure 3: Numways algorithm.

The results are as follows:

indices	0	1	2	3	4	5	6	7	8	9
results	1	1	2	4	7	13	24	44	81	149

Runtime: The runtime of a dynamic progrmaming algorithm is the size of the table multiplied by the work done to calculate each cell. In this case that is $\mathcal{O}(n) \times \mathcal{O}(1) = \mathcal{O}(n)$

3 Example 2

Suppose you had two operations:

- add one
- multiply by two

How many operations does it take to get from 0 to some k.

The results are as follows.

indices	0	1	2	3	4	5	6	7	8	9
results	0	1	2	3	3	4	4	5	4	5

It's not monotonic, and it's actually a little hard. Note that for 8 it's a power of two so it takes fewer operations than either of its neighbors. Trivially you could find this in exponential time by trying all possibilities. Let's write our recurrence.

$dp[0] = 0$ because for 0, you need zero operations. $dp[1] = 1$ because for 1, you add one to zero. Now we'll try to multiply by two as much as possible so we'll add 1 if the element is odd and multiply by two if it is even. We get the following overall recurrence.

$$d(i) = \left\{ \begin{array}{ll} d(i-1)+1, & \text{if } i \text{ is odd} \\ d(i/2)+1, & \text{if } i \text{ is even} \end{array} \right\} \tag{1}$$

This might require a proof, but it's simple enough you can likely see why it is what it is. Let's actually write out the algorithm.

```
def minop(k):
    initialize dp as table of size k + 1 with 0s
    dp[0] = 0
    dp[1] = 1

    for i in 2..(k + 1)
        dp[i] = dp[i - 1] + 1
        if i is even
            dp[i] = min(dp[i], dp[i / 2] + 1)

    return dp[k]
```

Figure 4: Minop algorithm.

Minimum is a fairly common tool in dynamic programming problems. Usually you choose between two possible recurrences, and the minimum results in your answer for that element in the table. You could try some greedy approach, but it gets hard to prove if that is optimal.

4 Example 3

Problem: House robber. Given an array houses with values $H = [h_1 \dots h_n]$, you want to rob them, but you can't rob two adjacent houses or alarms will go off. What is the maximum amount you can steal? Here's an example: $[2, 7, 9, 3, 1]$ would yield $2 + 9 + 1 = 12$.

Solution: Let's define our array $T[0 \dots n]$ with $T[i]$ =maximum we can steal from houses $0 \dots i$. The base cases are that $T[0] = H[0], T[1] = \max(H[0], H[1])$.

Now let's develop the recurrence. At the next house visited, we can choose to rob or not rob a house. We take the maximum of both scenarios. If we rob the house, than we could not have robbed the previous house, and if we didn't rob the house, then we could have robbed the previous house. This gives us the recurrence

$$T[i] = \max(T[i-2] + H[i], T[i-1])$$

```
def houserobber(H):
    initialize dp as table of size n with 0s
    dp[0] = H[0]
    dp[1] = max(H[0], H[1])

    for i in 2..n
        dp[i] = max(dp[i - 2] + H[i], dp[i - 1])
    return dp[n - 1]
```

Figure 5: House Robber algorithm

For example, given $H = [2, 7, 9, 3, 1]$, $T = [2, 7, 11, 11, 12]$. The runtime is $\mathcal{O}(n)$ since we have a table of size n with $\mathcal{O}(1)$ work at each step.

5 Example 4

Problem: Given n, m, what is the number of paths through an $n \times m$ grid from (1, 1) to (n, m) if you can only go down and right? For example, given a 3×3 matrix, there are 6 paths. This can be solved without dynamic programming, just some combinatorics, but let's do it with DP.

Solution: Let's define array $T[1 \dots n][1 \dots m]$ where $T[i][j]$ =the number of paths from (1, 1) to (i, j). The base cases are that $T[1 \dots n][1]$ and $T[1][1 \dots n]$ equal 1. This is essentially the first row and the first column of the matrix.

Now working on the recurrence, for non-base case cells,

the number of ways to reach (i, j) is the sum of the number of ways to reach the cell above it and the cell to its left. This is because you can only come to (i, j) from $(i-1, j)$ or $(i, j-1)$.

So, the recurrence relation is:

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

Finally, $dp[n][m]$ will give the number of paths from $(1, 1)$ to (n, m).

```
def count_paths(n, m):
   initialize dp as a table of size (n+1)*(m+1) with 0s

    for i in 0...n:
        dp[i][0] = 1
    for j in 0...m:
        dp[0][j] = 1

    for i in 1...n:
        for j in 1..m:
            dp[i][j] = dp[i-1][j] + dp[i][j-1]

    return dp[n][m]
```

Figure 6: House Robber algorithm

The Time and Space complexity is $O(n * m)$

6 Example 5

Problem Similar question as example 4, Find the number of paths to reach cell (n, m). Additionally, the grid has bombs, denoted by an input *bombs* such that $bombs[i][j] = True$ denoted cell (i, j) has a bomb. Find number of paths from $(0, 0)$ to (n, m) such that no bombs are traversed.

Unlike example 4, this problem cannot be solved with combinatorics and requires dynamic programming.

Let's define array $T[1 \ldots n][1 \ldots m]$ where $T[i][j]$ =the number of paths from (1, 1) to (i, j).

Base Cases:

1. If the starting cell $(1, 1)$ has a bomb, then there are 0 paths.

$$dp[1][1] = \begin{cases} 0 & \text{if } bombs[1][1] = \text{True} \\ 1 & \text{otherwise} \end{cases}$$

2. For the first row and first column:

$$dp[i][1] = \begin{cases} 0 & \text{if } bombs[i][1] = \text{True} \\ dp[i-1][1] & \text{otherwise} \end{cases}$$

$$dp[1][j] = \begin{cases} 0 & \text{if } bombs[1][j] = \text{True} \\ dp[1][j-1] & \text{otherwise} \end{cases}$$

Recurrence Relation:
For all cells not on the first row or first column:

$$dp[i][j] = \begin{cases} 0 & \text{if } bombs[i][j] = \text{True} \\ dp[i-1][j] + dp[i][j-1] & \text{otherwise} \end{cases}$$

This means that if a cell has a bomb, no paths can go through it. Otherwise, the number of paths to a cell is the sum of the paths to the cell above it and the cell to its left.

The solution will be in $dp[n][m]$, which gives the number of paths from $(1, 1)$ to (n, m) without traversing any bombs.

```
def count_paths_with_bombs(n, m, bombs):
    initialize dp as a table of size (n+1)*(m+1) with 0s

    if bombs[0][0]:
        return 0

    dp[0][0] = 1

    for i in 1...n:
        if not bombs[i][0]:
            dp[i][0] = dp[i-1][0]

    for j in 1...m:
        if not bombs[0][j]:
            dp[0][j] = dp[0][j-1]

    for i in 1...n:
        for j in 1...m:
            if not bombs[i][j]:
                dp[i][j] = dp[i-1][j] + dp[i][j-1]

    return dp[n][m]
```

Figure 7: House Robber algorithm

The overall time complexity is $O(n \times m)$. This is because we're essentially visiting each cell of the grid exactly once. The space complexity is also $O(n \times m)$ due to the "dp" table.