EECS 376: Discussion 3

Dynamic Programming

John Xie
johnxie@umich.edu

Agenda

- **Class Logistics**
- Dynamic Programming

Logistics

- HW3 due NEXT Wednesday, Sept. 20 at 8:00 PM technically! (10:00 PM without penalty)
- Go to office hours if you are having issues with homework
- If you don’t want to leave your room, post questions on Piazza or join virtual office hours

Agenda

- Class Logistics
- **Dynamic Programming**

Dynamic Programming

- Solves problems that are basically recursive problem. It is like the opposite of Divide-And-Conquer algorithms.

- Use *memoization* to optimize solving problems with
  - Many *overlapping subproblems*. (You need to use the solution to some subproblems multiple times)
  - An *optimal substructure.* (You can find the optimal solution to a problem using the optimal solution to a subproblem)

- DP is efficient because it trades off memory for time. O(1) lookup for previous solutions

Ex: Fibonacci Numbers

Repeated, overlapping subproblems

- Fib(4)
  = Fib(3) + Fib(2)
  = Fib(2) + Fib(1) + Fib(1) + Fib(0)
  = Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0)

Recurrence Relation:

- Fib(n) = Fib(n-1) + Fib(n-2) for n >= 2

Base Cases:
F(1) = 1
F(0) = 0

Bottom-Up vs Top-Down

Bottom-Up:

- Iterative (Usually done with a for-loop)
- Start with the smallest possible sub-problem and build upwards
- Almost always faster in practice
- Does not run the risk of overflowing the stack
- Easier IMO

Top-Down:

- Recursive (Usually done with recursive calls)
- Start at the main problem and then break down into smaller subproblems
- Only calculate subproblems that are actually used
- Harder IMO

Ex: Fibonacci Numbers

```
// bottom up - iterative
int fib(int n){
    vector<int> memo(n+1); // memo[i] = ith fibonacci number;
    memo[0] = 0;
    memo[1] = 1;

    for(int i = 2; i<=n; i++)
        memo[i] = memo[i-1] + memo[i-2];

    return memo[n];
}
```

Ex: Fibonacci Numbers

```
// top-down recursive
int fib(int n){
    vector<int> memo(n+1, -1); // memo[i] = ith fibonacci number;
    return topdown(n, memo);
}

int topdown(int n, vector<int>& memo){
    if(n <= 1) return n;
    if(memo[n] != -1) return memo[n];

    return memo[n] = topdown(n-1, memo) + topdown(n-2, memo);
}
```

0-1 Knapsack Problem

Definition 1.1 (0-1 Knapsack problem)**.** You are given two n-length arrays containing positive integer weights $W = (w_1, w_2, \ldots, w_n)$ and values $V = (v_1, v_2, \ldots, v_n)$ of n items (where the i^{th} item has weight $w_i \in \mathbb{Z}^+$ and value $v_i \in \mathbb{Z}^+$),[1] and a knapsack with weight capacity $C \in \mathbb{N}$.

You must pick a subset of items $S \subseteq \{1, 2, \ldots, n\}$ (there are no copies or fractional items), such that the total weight of the chosen items is less than or equal to the weight capacity: $\sum_{i \in S} w_i \leq C$. What is the maximum total value you can obtain from objects in your knapsack, $\sum_{i \in S} v_i$?

0-1 Knapsack Recurrence Relation

In order to solve this, we define the subproblem $K(i, C)$ to be the maximum value we can obtain by considering only the first i objects and a knapsack with weight capacity equal to C. There are three cases:

1. $i = 0$ or $C = 0$: In this case, clearly $K(i, C) = 0$ because there are either no items or no space in the knapsack.

2. $w_i > C$: In this case, you cannot fit the i^{th} item in the knapsack, so your set of items in the knapsack must consist only of the first $i - 1$ items. That is, $K(i, C) = K(i - 1, C)$

3. If neither of the above cases hold, then you can either have the i^{th} in the knapsack, or not, and the maximum value will be the larger of the two values you get from these two options.

$$K(i, C) = \max\{K(i-1, C-w_i) + v_i, K(i-1, C)\}$$

0-1 Knapsack Bottom-Up

Input: Integers n, C, arrays W, V, and memo table DP.
Output: The maximum total value of objects the Knapsack can hold

function KNAPSACK(n, C, W, V)
    $DP[n][C] \leftarrow -1$ ▷ Initialize values in lookup-table to -1
    for $i = 0 : n$ **do**
        $DP[i][0] = 0$
    for $j = 0 : C$ **do**
        $DP[0][j] = 0$
    for $i = 1 : n$ **do**
        for $j = 1 : C$ **do**
            if $W[i] > j$ **then**
                $DP[i][j] = DP[i-1][j]$
            else
                $DP[i][j] = \max(DP[i-1][j - W[i]] + V[i], DP[i-1][j])$
    return $DP[n][C]$

0-1 Knapsack Top-Down

Input: Integers n, C, arrays W, V, and lookup-table DP with values initialized to -1. Again, note the 1-based indexing.
Output: The maximum total value of objects the Knapsack can hold

function $\textsc{Knapsack}(n, C, W, V, DP)$
    if $n = 0$ or $C = 0$ **then**
        return 0
    if $DP[n-1][C] = -1$ **then**
        $DP[n-1][C] \leftarrow \text{Knapsack}(n-1, C, W, V, DP)$
    if $W[n] > C$ **then**
        return $DP[n-1][C]$
    if $DP[n-1][C-W[n]] = -1$ **then**
        $DP[n-1][C-W[n]] \leftarrow \text{Knapsack}(n-1, C-W[n], W, V, DP)$
    return $\max(DP[n-1][C-W[n]] + V[n], DP[n-1][C])$

Worksheet Question #2c

2. Let $\#C(\ell)$ denote the number of binary strings with length ℓ that have no consecutive occurrences of a 1. For example $\#C(3) = 5$; we can list all binary strings of length 3 and determine by inspection that only the strings 000, 001, 010, 100, and 101 have no consecutive occurrences of a 1.

   (c) Show that $\#C(\ell) = \#C(\ell - 1) + \#C(\ell - 2)$ for $\ell \geq 2$.

Worksheet Question #2c

Solution: Observe that there for a string of length $\ell \geq 2$, it must either begin with a 1 or a 0. If it begins with a 1, then the next bit must be a 0, and the remaining string has length $\ell - 2$ and must have no consecutive occurrences of a 1. If it begins with a 0, then the next bit may be any bit, and the remaining string has length $\ell - 1$ and must have no consecutive occurrences of a 1.

As such, we have that $\#C(\ell) = \#C(\ell - 1) + \#C(\ell - 2)$ for $\ell \geq 2$.

If we want to compute this recursively, we only need to add base cases. Observe that $\#C$ is only defined for non-negative integers, so our base cases should be $\#C(0) = 1$ and $\#C(1) = 2$.

Worksheet Question #2c

Input: Unsigned integer n

function $\#C(n)$
    if $n = 0$ **then**
        return 1
    if $n = 1$ **then**
        return 2
    return $\#C(n-1) + \#C(n-2)$

Worksheet Question #2d

(d) Use the bottom-up-table approach to improve your recursive algorithm.

Worksheet Question #2d

(d) Use the bottom-up-table approach to improve your recursive algorithm.

Input: Nonnegative integer n

function $\#C(n)$
  $DP \leftarrow$ a table of n integers
  $DP[0] \leftarrow 1$
  $DP[1] \leftarrow 2$
  for $i = 2$ to n **do**
    $DP[i] \leftarrow DP[i-1] + DP[i-2]$
  return $DP[n]$

Worksheet Question #3a

3. Given an array of $n \geq 1$ *positive real numbers* (represented as constant size floating points), $A[1..n]$, we are interested in finding the smallest *product* of any subarray of A, i.e.,

$$\min\{A[i]A[i+1]\cdots A[j] : i \leq j \text{ are indices of } A\}.$$

   (a) Give a recurrence relation (including base cases), that is suitable for an $O(n)$ time dynamic programming solution to the smallest product problem. Briefly explain why your recurrence relation is correct.

   Hint: The longest increasing subsequence recurrence might give you some inspiration, but this recurrence should be simpler than that.

Worksheet Question #3a

Solution: For each $0 \le i \le n$, let $S(i)$ be the smallest product that *ends* at position i of A (it's convenient to define smallest product of empty array as 1). Then we see that

$$S(i) = \begin{cases} 1 & i = 0 \\ \min\{S(i-1) \cdot A[i], A[i]\} & i > 0 \end{cases}.$$

Indeed, the key observation is that, since the product needs to end at position i, we should only extend the previous product if it doesn't hurt to do so (i.e., when $S(i) \le 1$; this observation can actually lead to a simpler algorithm that does not require a table). Given this recurrence, we may compute the smallest product as $\min\{S(i) \mid 1 \le i \le n\}$.

Worksheet Question #3b

(b) Give pseudocode implementing an $O(n)$ time bottom-up dynamic programming solution to the smallest product problem.

Worksheet Question #3b

(b) Give pseudocode implementing an $O(n)$ time bottom-up dynamic programming solution to the smallest product problem.

---

function $\text{MinProd}(A[1..n])$

  allocate $S[0..n]$

  $S[0] \leftarrow 1$

  for $i = 1..n$ **do**

    $S[i] \leftarrow \min\{S[i-1] \cdot A[i], A[i]\}$

  return $\min\{S[i] \mid 1 \leq i \leq n\}$

Thank you for your time. Any questions?