Dynamic Programming II

Ethan Arnold, Kevin Chen

CS 104C

Spring 2019

DP as Transitions

- We've talked about dynamic programming in terms of recursive functions.
- We can also view dynamic programming as a way to do *transitions* between states.
- Usually we visualize states as nodes in a graph, and there are directed edges representing which states depend on which other states.
- The states correspond to the values that the arguments to the recursive function might take, and each state has an associated answer that corresponds to the value of the recursive function.
- The directed edges in the state graph correspond to recursive calls inside the recursive function; these are the "transitions".

Maximum path sum

- **Problem:** Given an $M \times N$ matrix A of integers, what is the maximum sum on a path from the top left of the matrix to the bottom right, where each move is either down or right?
- What should we define as our state?
  - An index (i, j) into the matrix A, and an associated answer of the maximum path sum ending at the element $A_{i,j}$.
- What transitions should we define between states? (How can we change our state?)
  - Move right (check $(i, j-1)$)
  - Move down (check $(i-1, j)$)
- How do we code this?

Maximum path sum solution

```
int maxPathSum(int[][] mat) {
    int M = mat.length, N = mat[0].length;
    int[][] dp = new int[M][N];
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++) {
            dp[i][j] = Integer.MIN_VALUE;
            if (i == 0 && j == 0)
                dp[0][0] = mat[0][0];
            if (i > 0 && dp[i - 1][j] +
                mat[i][j] > dp[i][j])
                dp[i][j] = dp[i - 1][j] + mat[i][j];
            if (j > 0 && dp[i][j - 1] +
                mat[i][j] > dp[i][j])
                dp[i][j] = dp[i][j - 1] + mat[i][j];
        }
    return dp[M - 1][N - 1];
}
```

Maximum path sum solution

- Time complexity? $O(MN)$
- Space complexity? $O(MN)$ (could reduce to $O(\min\{M, N\})$)

Knapsack problem

- You are a jewel thief, and you want to steal the most valuable set of jewels possible from a set of n jewels.
- Each jewel has a value v_i and a weight w_i.
- You have a sack, which has a weight capacity C.
- What is the maximum value you can store in the sack?

Fractional Knapsack

- Say you also have a state-of-the-art laser cutter, and can cut the jewel to any real weight.
- The ratio of the jewel's value to it's weight does not change.
- What is the optimal strategy here?

Fractional knapsack

```
double fractionalKnapsack(List<Jewel> jewels,
                          double capacity) {
    Collections.sort(jewels, (a, b) -> {
        return Double.compare(
            a.value / a.weight,
            b.value / b.weight
        );    });
    Collections.reverse(jewels);
    double value = 0.0;
    for (Jewel jewel : jewels) {
        double taken = Math.min(capacity, jewel.weight);
        capacity -= taken;
        value += jewel.value * taken / jewel.weight;
    }
    return value;
}
```

Fractional knapsack

```
double fractionalKnapsack(List<Jewel> jewels,
                          double capacity) {
    Collections.sort(jewels, (a, b) -> {
        return Integer.compare(
            a.value * b.weight,
            b.value * a.weight
        );    });
    Collections.reverse(jewels);
    double value = 0.0;
    for (Jewel jewel : jewels) {
        double taken = Math.min(capacity, jewel.weight);
        capacity -= taken;
        value += taken * jewel.value / jewel.weight;
    }
    return value;
}
```

- Time Complexity? $O(n \log n)$ Space Complexity? $O(n)$

Infinite knapsack

- Now say we can't use our laser cutter, but we've stumbled on the motherload of jewels, and effectively have an infinite number of any jewel.
- Now what's the best strategy?

Infinite knapsack

- What is our state? The total capacity we've used so far
- What are the transitions?
  1. Take a jewel (try all of them, checking state $j - w_i$ for each i)
  2. Don't take a jewel (you can think of this as reserving some empty space in our knapsack)
- How do you update your state with each of these operations, using the answers from other states?
  1. For each jewel i: look at state $j - w_i$, and add v_i to the answer for that state
  2. Look at state $j - 1$, add don't add anything (we don't get any value by leaving empty space)

Infinite knapsack

```
int infKnapsack(List<Jewel> jewels, int maxCapacity) {
    int[] bestValue = new int[maxCapacity + 1];
    bestValue[0] = 0;
    for (int cap = 1; cap <= maxCapacity; cap++)
        bestValue[cap] = bestValue[cap - 1];
        for (Jewel jewel : jewels)
            if (jewel.weight <= cap)
                bestValue[cap] = Math.max(
                    bestValue[cap],
                    jewel.value +
                        bestValue[cap - jewel.weight]
                );
    return bestValue[maxCapacity];
}
```

- Time complexity? $O(nC)$
- Space complexity? $O(n + C)$

0 / 1 Knapsack

- This is the most common variant of knapsack. Each item can be taken at most once.
- How can we modify our previous solution to solve this?
- Add another dimension to our state. Using the first i jewels, what is the maximum value we can get with capacity j? (Our state is the pair (i, j).)
- How do we transition between states now?
  1. Don't take jewel i (check state $(i-1, j)$)
  2. Do take jewel i (check state $(i-1, j-w_i)$)

0/1 Knapsack

```
int knapsack(List<Jewel> jewels, int maxCapacity) {
int n = jewels.size();
int[][] bestValue = new int[n][maxCapacity + 1];
for (int cap = 1; cap <= maxCapacity; cap++)
    for (int i = 1; i < n; i++)
        jewel = jewels.get(i - 1);
        bestValue[i][cap] = bestValue[i - 1][cap];
        if (jewel.weight <= cap)
            bestValue[i][cap] = Math.max(
                bestValue[i][cap],
                jewel.value +
                    bestValue[i - 1][cap - jewel.weight]
            );
    return bestValue[n - 1][maxCapacity];
}
```

- Time Complexity? $O(nC)$
- Space Complexity? $O(nC)$

Bounded Knapsack

- Now in addition to price p_i and weight w_i, each jewel has a count c_i (all $c_i \leq c$).
- How can we modify our previous solution to solve this?
- Option 1: Just make c_i copies of each jewel.
- What is the runtime of this? $O(\sum_i c_i C) = O(ncC)$
- Can we do better?

Bounded Knapsack

- For each jewel (v_i, w_i, c_i), create the following jewels and solve 0 / 1 Knapsack:
  1. (v_i, w_i)
  2. $(2 \cdot v_i, 2 \cdot w_i)$
  3. $(4 \cdot v_i, 4 \cdot w_i)$
  4. $(8 \cdot v_i, 8 \cdot w_i)$
  5. $\dots$
  6. $(2^k \cdot v_i, 2^k \cdot w_i)$
  7. $((c_i - \sum 2^k)v_i, (c_i - \sum 2^k)w_i)$
- Any amount of a jewel can be represented as a sum of values from this list.
- How many items are in this list? $O(\log(c_i))$.
- What is the runtime of this? $O(\sum_i \log(c_i)C) = O(n \log(c)C)$