CS375 Notes: Dynamic Programming

October 26, 2026

1 Dynamic Programming, Day 1

1.1 A recursive Fibonacci algorithm

Consider the recursive function

```
def F(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return F(n-1) + F(n-2)
```

The recursion tree branches into calls on smaller inputs. For example, a call to $F(n)$ makes calls to $F(n-1)$ and $F(n-2)$, and the same pattern repeats until reaching the base cases. Many subproblems are recomputed multiple times.

Let $T(n)$ be the running time of this recursive function. Its recurrence is

$$T(n) = T(n-1) + T(n-2) + \Theta(1),$$

with constant-time base cases. This grows exponentially, so the direct recursive algorithm becomes inefficient even for moderately large n.

1.2 Memoization / caching

Store the answer to each subproblem once it has been computed. Subsequent requests for that value can return the stored answer rather than recomputing it.

```
def F(n):
    A = [0, 1]
    if n <= 1:
        return A[n]
    while len(A) <= n:
        A.append(A[-1] + A[-2])
    return A[n]
```

Here, the array stores the Fibonacci values already computed. Each value is produced once, giving $O(n)$ time and $O(n)$ space.

2 Maximum-Weight Independent Set on a Path

2.1 Problem statement

Input: An array A of weights.

Output: The maximum possible sum of non-adjacent entries of A.

For example, let

$$A = [7, 2, 4, 8].$$

We want to choose entries so that no two selected entries are adjacent, while maximizing their total weight.

Let $\mathrm{OPT}(i)$ be the maximum sum obtainable from the first i entries of A. The key decision is whether to include the last entry under consideration:

- If we do not take $A[i]$, the best value is $\mathrm{OPT}(i-1)$.
- If we take $A[i]$, we cannot take its neighbor, so the best value is $A[i] + \mathrm{OPT}(i-2)$.

Thus, for the usual indexing convention,

$$\mathrm{OPT}(i) = \max\big(\mathrm{OPT}(i-1),\ A[i] + \mathrm{OPT}(i-2)\big).$$

The notes use the corresponding base cases $\mathrm{OPT}(0) = 0$ and $\mathrm{OPT}(1) = A[0]$ when indexing the array from zero.

2.2 Recursive formulation

```
def OPT(n):
    if n == 0:
        return 0
    if n == 1:
        return A[0]
    return max(OPT(n-1), A[n-1] + OPT(n-2))
```

This recurrence expresses the correct choice, but a direct recursive implementation repeats subproblems. Dynamic programming avoids that repeated work by filling in the answers in order.

2.3 Bottom-up dynamic programming

```
def OPT(n):
    # W has length n+1; initially all entries are zero.
    W = [0] * (n + 1)
    if n >= 1:
        W[1] = A[0]
    for i in range(2, n + 1):
        W[i] = max(A[i-1] + W[i-2], W[i-1])
    return W[n]
```

Each entry $W[i]$ stores the optimum for the first i array elements. Every entry is computed in constant time, so the running time is $O(n)$ and the table uses $O(n)$ space.

2.4 Recovering the actual set of entries

The table gives the optimal value, but we may also want to recover which entries achieve it. Starting from the end, compare the current table entry with the value obtained by skipping the current item.

```
# W is the completed dynamic-programming table.
output = []
i = n
while i > 0:
    if W[i] == W[i-1]:
        # An optimal solution can skip A[i-1].
        i -= 1
    else:
        # Take A[i-1], then skip its neighbor.
        output.append(A[i-1])
        i -= 2
return output
```

If the two values tie, skipping the current entry still permits an optimal solution. The reconstruction takes $O(n)$ time. Depending on the order in which items are appended, the resulting list may be in reverse order.

3 Rod Cutting

3.1 Problem statement

Given: A rod of integer length n and an array of prices $P[0], \ldots, P[n]$, where $P[i]$ is the selling price of a piece of length i.

Output: The best way to cut up the rod into integer-length pieces to maximize the sum of their prices.

For example, the notes use the price list

$$P = [0, 1, 3, 7, 7, \ldots]$$

and consider a rod of length 4. Some possible cuts and their revenues are

Pieces	Total price
$1+1+1+1$	4
$1+1+2$	5
$2+2$	6
$3+1$	8

The best option among these is the cut $3+1$, with revenue 8.

3.2 Why a greedy choice can fail

A tempting strategy is to choose a piece with the best price per unit length and recurse on the remaining rod. Consider

$$P = [0, 1, 8, 10]$$

and a rod of length 3. The length-2 piece has the best price per unit length, since $8/2 = 4$ while $10/3 < 4$. Thus this greedy rule first cuts off a piece of length 2, leaving a piece of length 1, for total revenue

$$P[2] + P[1] = 8 + 1 = 9.$$

But leaving the rod uncut gives revenue $P[3] = 10$, which is better. So the greedy choice already fails for $n = 3$.

3.3 Optimal substructure and recurrence

Let $\mathrm{OPT}(i)$ be the maximum revenue obtainable from a rod of length i. The base case is

$$\mathrm{OPT}(0) = 0.$$

For $i \geq 1$, choose the length j of the first piece. Its price is $P[j]$, and the remaining rod has length $i - j$. Therefore,

$$\boxed{\mathrm{OPT}(i) = \max_{1 \leq j \leq i} \big(P[j] + \mathrm{OPT}(i - j)\big).}$$

The recurrence tries every possible first cut and chooses the one with the largest total revenue. To see why it is correct, consider an optimal way to cut a rod of length i. Its first piece must have some length j with $1 \leq j \leq i$. That piece contributes $P[j]$, and the remaining pieces form a cutting of a rod of length $i - j$. The remaining cutting must itself be optimal: if it were not, we could replace it with a higher-revenue cutting of the same remaining length and improve the original solution, contradicting optimality. Hence the best revenue among solutions whose first piece has length j is exactly $P[j] + \mathrm{OPT}(i - j)$. Taking the maximum over all possible j therefore gives the optimal revenue for length i.

3.4 Bottom-up algorithm

```
def OPT(n):
    W = [0] * (n + 1)
    W[0] = 0
    for i in range(1, n + 1):
        W[i] = P[i]
        for j in range(1, i):
            W[i] = max(W[i], P[j] + W[i-j])
    return W[n]
```

For each length i, the algorithm considers all possible first-piece lengths. There are $O(i)$ candidates for length i, so the total running time is

$$\sum_{i=1}^{n} O(i) = O(n^2),$$

and the table takes $O(n)$ space. (Equivalently, the inner loop can include $j = i$ if the implementation consistently handles the remaining length of zero.)