Dynamic Programming

Algorithm design strategies seen thus far:

- Brute force — exhaustively consider through all possibilities
- Divide and conquer — examples: binary search, mergesort, quicksort, Karatsuba
- Greedy algorithm — examples: Prim's algorithm for minimum spanning tree, Dijkstra's algorithm for single source shortest path
- Dynamic programming — this lecture

1 Definitions

Dynamic Programming is an algorithmic paradigm: It is used when the solution can be recursively described in terms of solutions to subproblems (optimal substructure).

Memoization: Algorithm finds solutions to subproblems and stores them in memory for later use. More efficient than brute-force methods, which solve the same subproblems over and over again.

Typically solved bottom-up, building a table of solved subproblems that are used to solve larger ones. A big difference from the divide and conquer paradigm is that unlike divide and conquer the subproblems overlap.

The word programming is historical, and was chosen by Richard Bellman to describe a program in the sense of a schedule.

We study a number of examples to see Dynamic programming in action.

2 Example 1 — Fibonacci series

The Fibonacci series can be recursively expressed as:

$$F_n = F_{n-1} + F_{n-2}$$

with $F_0 = 0$, and $F_1 = 1$.

$$F_2 = F_1 + F_0, \quad F_3 = F_1 + F_2, \text{ and so on}$$

```
int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

The naive recursive solution has exponential in n run time — $O(c^n)$ where c is a constant. The exponential run time happens because of repeatedly solving the same subproblems.

Dynamic programming solution with memoization — Top down approach

Idea: Cache (memoize) computed results. For any new computation, check cache to see if it holds the result, or else recurse.

```
vector<int> A(n+1, -1); // For memoization
A[0] = 0;
A[1] = 1;

int fibonacci(int n) {
    if (A[n] != -1) // Result in cache
        return A[n];
    else
        A[n] = fibonacci(n-1) + fibonacci(n-2); // Cache result
    return A[n];
}
```

Dynamic programming solution with tabulation — Bottom up approach
Idea: Build a table of partial results from the bottom up.

```
int fibonacci(int n) {
    vector<int> A(n+1);
    A[0] = 0;
    A[1] = 1;
    for (int i = 2; i <= n; i++) {
        A[i] = A[i-1] + A[i-2];
    }
    return A[n];
}
```

Time: $O(n)$ Space: $O(n)$ Turns out you don't need a whole array. All you need is the last two numbers (two variables). So the space required is only $O(1)$.

3 Example 2 — Rod cutting problem

You are given a rod of size $n > 0$, it can be cut into any number of pieces k ($k \leq n$). Price for each piece of size i is represented as $p(i)$ and maximum revenue from a rod of size i is $r(i)$ (could be split into multiple pieces). Find maximum $r(n)$ for the rod of size n.

For example, here's an array of prices where the array index + 1 indicates rod length:

$$p = \{1, 5, 8, 9, 10, 17, 17, 20\}$$

The price of a rod of length 2 is 5.
Example: What is the best price for a rod of length 3?

- Break into 3 pieces of length 1: \$3
- Break into 1 piece of length 2 and 1 piece of length 1: \$6
- Don't break the rod: \$8

Best price: don't cut the rod.
Example for rod of length 4: breaking up into 2 and 2 yields \$10.
Let's build some intuition:

- Length 1: only possibility price \$1
- Length 2: \$2 (cut) or \$5 (uncut) $\rightarrow$ uncut is better
- Length 3: possibilities (0,3), (1,2) $\rightarrow$ max(8,6) = 8
- Length 4: (0,4), (1,3), (2,2) $\rightarrow$ max(9,9,10) = 10

General algorithm:

```cpp
int rodCutting(const vector<int>& P, int n) {
    vector<int> maxP(n+1, 0);
    for (int i = 1; i <= n; i++) {
        int maxVal = P[i-1];
        for (int j = 1; j <= i/2; j++) {
            maxVal = max(maxVal, maxP[j] + maxP[i-j]);
        }
        maxP[i] = maxVal;
    }
    return maxP[n];
}
```

Complexity: $O(n^2)$

Example 3 — Max sum of non adjacent elements

Given an array of positive integers, find the maximum sum of non adjacent elements.

Example: [75, 105, 120, 75, 90, 135] Ans: $330 = 75 + 120 + 135$

General approach:

```cpp
int maxSumNonAdj(const vector<int>& in) {
    int n = in.size();
    vector<int> maxSum(n);
    maxSum[0] = in[0];
    maxSum[1] = max(in[0], in[1]);
    for (int i = 2; i < n; i++) {
        maxSum[i] = max(maxSum[i-1], maxSum[i-2] + in[i]);
    }
    return maxSum[n-1];
}
```

4 Example 4 — Levenshtein Distance (Edit distance)

The Levenshtein distance between two words is the minimum number of single character edits (insertions, deletions, or substitutions) required to change one word to another. Each operation has unit cost.

Example: kitten $\rightarrow$ sitting = 3 edits

General formula:

- If min(i,j)=0, dist[i][j] = max(i,j)
- If X[i] == Y[j], dist[i][j] = dist[i-1][j-1]
- Else dist[i][j] = 1 + min(dist[i-1][j], dist[i][j-1], dist[i-1][j-1])

```cpp
int editDistance(const string& X, const string& Y) {
    int m = X.size(), n = Y.size();
    vector<vector<int>> dist(m+1, vector<int>(n+1));

    for (int i = 0; i <= m; i++) dist[i][0] = i;
    for (int j = 0; j <= n; j++) dist[0][j] = j;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i-1] == Y[j-1])
                dist[i][j] = dist[i-1][j-1];
            else
                dist[i][j] = 1 + min({dist[i-1][j], dist[i][j-1], dist[i-1][j-1]});
        }
    }
    return dist[m][n];
}
```

Complexity: $O(mn)$

5 Example 5 — Knapsack problem (1/0)

Given items with values V_i and weights W_i, capacity W, maximize value without exceeding W.

```cpp
int knapsack(const vector<int>& V, const vector<int>& W, int capacity) {
    int n = V.size();
    vector<vector<int>> dp(n+1, vector<int>(capacity+1, 0));
    for (int i = 1; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            if (W[i-1] <= w)
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - W[i-1]] + V[i-1]);
            else
                dp[i][w] = dp[i-1][w];
        }
    }
    return dp[n][capacity];
}
```

Complexity: $O(nW)$

6 More Examples

See: `https://www.techiedelight.com/Category/dynamic-programming/`