Dynamic Programming (DP) Tutorial

This tutorial explains essential dynamic programming problems and techniques. Each section includes explanations and C code implementations.

Knapsack Problem

The knapsack problem involves selecting items with given weights and values to maximize total value without exceeding a fixed capacity. Key variants include:

- **0/1 Knapsack**: Items are indivisible (taken or skipped). Solved via dynamic programming (DP) with a 2D table where rows represent items and columns represent capacities. The recurrence is:
  `dp[i][w] = max(value[i] + dp[i-1][w-weight[i]], dp[i-1][w])`.
- **Fractional Knapsack**: Items can be split. Solved greedily by selecting items with the highest value-to-weight ratio first.
- **Unbounded Knapsack**: Items can be selected multiple times. Uses a 1D DP array with `dp[w] = max(dp[w], value[i] + dp[w-weight[i]])`.

Subset Sum Problem

Given a set of non-negative integers and a target sum, determine if any subset sums to the target. Solved using DP where:

- `dp[i][s] = true` if sum `s` can be formed using the first `i` elements.
- Recurrence: `dp[i][s] = dp[i-1][s] OR dp[i-1][s-arr[i]]`.
  Optimized via 1D DP to track achievable sums.

Matrix Chain Multiplication

Minimize scalar multiplications when multiplying a chain of matrices. Key steps:

1. Define `dp[i][j]` as the minimum cost to multiply matrices from `i` to `j`.
2. Recurrence:
   `dp[i][j] = min(dp[i][k] + dp[k+1][j] + dims[i-1]*dims[k]*dims[j])` for all `k` in `[i, j-1]`.
3. Use a table to store costs and track optimal parenthesization.

Longest Increasing Subsequence (LIS)

Find the longest subsequence where elements are strictly increasing. Approaches:

- **DP solution**: For each element, check all previous elements:
  `dp[i] = max(dp[i], dp[j] + 1)` where `j < i` and `arr[j] < arr[i]`. Time: $O(n^2)$.

- **Optimized approach**: Use a tail array to track the smallest tail of increasing subsequences of length `i+1`. Time: O(n log n).

Longest Common Subsequence (LCS)

Find the longest subsequence common to two strings. DP approach:

- `dp[i][j]` = LCS of `str1[0..i-1]` and `str2[0..j-1]`.
- Recurrence:
  `dp[i][j] = dp[i-1][j-1] + 1` if `str1[i-1] == str2[j-1]`,
  else `max(dp[i-1][j], dp[i][j-1])`.

DP on Trees

Solve tree-based problems by traversing subtrees and combining results:

- **Post-order traversal**: Process children before the root.
- **State definition**: `dp[u][state]` represents the solution for the subtree rooted at node `u` under a given state (e.g., inclusion in a set).
- **Examples**: Maximum independent set, tree diameter.

DP on Grids

Solve 2D grid problems using DP:

- **State**: `dp[i][j]` represents the solution for cell `(i, j)`.
- **Transitions**: Based on allowed moves (e.g., right/down).
- **Applications**: Path counting, minimum cost path, and obstacle avoidance.

Digit DP

Count numbers in a range satisfying digit-based constraints:

- **State**: `(position, tight, sum, ...)`.
- **Tight constraint**: Whether current digits match the upper-bound prefix.
- **Memoization**: Store states to avoid recomputation.

Bitmask DP

Solve subset-based problems using bitmask integers:

- **Bitmask**: An integer where the `i`-th bit indicates inclusion of the `i`-th element.
- **State**: `dp[mask][params]` for subsets represented by `mask`.
- **Applications**: Traveling Salesman Problem (TSP), assignment problems.

Each technique leverages optimal substructure and overlapping subproblems, with state design tailored to problem constraints.

1. 0/1 Knapsack Problem

Given weights and values of items, maximize total value under a weight limit.

C Code:

```c
int knapsack(int W, int wt[], int val[], int n) {
    int dp[n+1][W+1];
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (wt[i-1] <= w)
                dp[i][w] = (val[i-1] + dp[i-1][w-wt[i-1]] > dp[i-1][w]) ?
                            val[i-1] + dp[i-1][w-wt[i-1]] : dp[i-1][w];
            else
                dp[i][w] = dp[i-1][w];
        }
    }
    return dp[n][W];
}
```

---

2. Subset Sum

Determine if a subset of numbers adds up to a given sum.

C Code:

```c
int subsetSum(int arr[], int n, int sum) {
    int dp[n+1][sum+1];
    for (int i = 0; i <= n; i++)
        dp[i][0] = 1;
    for (int j = 1; j <= sum; j++)
        dp[0][j] = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (arr[i-1] <= j)
                dp[i][j] = dp[i-1][j] || dp[i-1][j - arr[i-1]];
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    return dp[n][sum];
}
```

---

3. Matrix Chain Multiplication

Find the most efficient way to multiply a sequence of matrices.

C Code:

```c
int matrixChain(int p[], int n) {
    int dp[n][n];
    for (int i = 1; i < n; i++)
        dp[i][i] = 0;

    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            dp[i][j] = 1e9;
            for (int k = i; k < j; k++) {
                int q = dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < dp[i][j])
                    dp[i][j] = q;
            }
        }
    }
    return dp[1][n-1];
}
```

---

4. Longest Increasing Subsequence (LIS)

Find the length of the longest increasing subsequence.

C Code:

```c
int lis(int arr[], int n) {
    int dp[n];
    int max = 0;
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j] && dp[i] < dp[j] + 1)
                dp[i] = dp[j] + 1;
        }
        if (dp[i] > max)
            max = dp[i];
    }
    return max;
}
```

---

5. Longest Common Subsequence (LCS)

Find the longest subsequence common to two strings.

C Code:

```c
int lcs(char* X, char* Y, int m, int n) {
    int dp[m+1][n+1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
                dp[i][j] = (dp[i-1][j] > dp[i][j-1]) ? dp[i-1][j] : dp[i][j-
1];
        }
    }
    return dp[m][n];
}
```

---

6. DP on Trees

Use DFS + DP to compute values in a tree, such as maximum sum from root to leaf.

C Code:

```c
#define MAX 100
int dp[MAX];

void dfs(int node, int parent, int tree[MAX][MAX], int value[]) {
    dp[node] = value[node];
    for (int i = 0; i < MAX; i++) {
        if (tree[node][i] && i != parent) {
            dfs(i, node, tree, value);
            if (dp[node] < value[node] + dp[i])
                dp[node] = value[node] + dp[i];
        }
    }
}
```

---

7. DP on Grids

Count the number of paths in an m x n grid.

C Code:

```c
int countPaths(int m, int n) {
    int dp[m][n];
    for (int i = 0; i < m; i++)
        dp[i][0] = 1;
    for (int j = 0; j < n; j++)
        dp[0][j] = 1;

    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }
    return dp[m-1][n-1];
}
```

---

8. Digit DP

Solve number problems digit by digit (e.g., count numbers with constraints).

C Code (Count numbers <= N with sum of digits <= K):

```c
int dp[20][200];

int digitDP(char* num, int pos, int tight, int sum) {
    if (pos == strlen(num)) return (sum >= 0);
    if (dp[pos][sum] != -1 && tight == 0) return dp[pos][sum];

    int limit = tight ? num[pos] - '0' : 9;
    int res = 0;
    for (int d = 0; d <= limit; d++) {
        res += digitDP(num, pos + 1, tight && (d == limit), sum - d);
    }

    if (!tight) dp[pos][sum] = res;
    return res;
}
```

---

9. Bitmask DP

Use bitmask to represent states (e.g., all cities visited in TSP).

C Code (TSP example):

```c
#define INF 1000000000
int tsp(int mask, int pos, int n, int dist[20][20], int dp[1<<20][20]) {
    if (mask == (1 << n) - 1) return dist[pos][0];
```

```
    if (dp[mask][pos] != -1) return dp[mask][pos];

    int ans = INF;
    for (int city = 0; city < n; city++) {
        if ((mask & (1 << city)) == 0) {
            int newAns = dist[pos][city] + tsp(mask | (1 << city), city, n,
dist, dp);
            if (newAns < ans) ans = newAns;
        }
    }
    return dp[mask][pos] = ans;
}
```

This concludes the Dynamic Programming tutorial in C. These patterns and problems are key to mastering competitive programming and technical interviews. Let me know if you'd like a PDF version or added visualizations!

Knapsack
https://www.youtube.com/watch?v=hagBB17_hvg
https://www.youtube.com/watch?v=cJ21moQpofY
https://www.youtube.com/watch?v=xOlhR_2QCXY
Subset Sum
https://www.youtube.com/watch?v=KBEFyzpUUEw
Matrix Chain Multiplication
https://www.youtube.com/watch?v=O_G2hVZvNBg
Longest Increasing Subsequence
https://www.youtube.com/watch?v=CE2b_-XfVDk
Longest Common Subsequence
https://www.youtube.com/watch?v=NnD96abizww
DP on Trees
https://www.youtube.com/watch?v=8cQryxznvwk
https://www.youtube.com/playlist?list=PLb3g_Z8nEv1j_BC-fmZWHFe6jmU_zv-8s
https://www.youtube.com/playlist?list=PL_z_8CaSLPWfxJPz2-YKqL9gXWdgrhvdn
https://www.youtube.com/watch?v=qZ5zayHSH2g
DP on Grids
https://www.youtube.com/watch?v=1hSt4sUH_gs
https://www.youtube.com/watch?v=-B8hWstTp20
https://www.youtube.com/watch?v=cthI6e5KGgg
https://www.youtube.com/watch?v=12o3Lw-Wt_k

Digit DP

https://www.youtube.com/watch?v=cthI6e5KGgg

https://www.youtube.com/watch?v=8cQryxznvwk

Bitmask DP

https://www.youtube.com/watch?v=0ABttC0Mw78

https://www.youtube.com/playlist?list=PLi0ZM-RCX5nuaQHFtKxEzbUtLgeHPf_R-