Dynamic Programming

July 5, 2018
Dynamic Programming

- Dynamic programming is a method for solving optimization problems.
- Compute the solutions to the subproblems once and store the solutions, so that they can be reused repeatedly later
  - Top-down DP: recursion + memoization
  - Bottom-up DP: iteratively store subproblems to a table
- Order the computations in a way that you avoid recalculate duplicate work.
- Trade space for time
Dynamic Programming vs Divide and Conquer

- Divide and Conquer
  - partition the problem into subproblems
  - solve the subproblems
  - combine the solutions to solve the original one

- In divide and conquer, the subproblems are usually independent, they did not call the same subsubproblems

- In dynamic programming, every subproblem is computed exactly once, and stored in a table for future use
Fibonacci Numbers

- Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, \ldots
- $F(0) = 0$
- $F(1) = 1$ (It has two base cases)
- $F(n) = F(n - 1) + F(n - 2)$, for $n \geq 2$
- Closed form of Fibonacci numbers

$$\alpha = \frac{1 + \sqrt{5}}{2}$$
$$\beta = \frac{1 - \sqrt{5}}{2}$$
$$F_n = \frac{\alpha^n - \beta^n}{\sqrt{5}}$$
Too Much Recursion

// Compute the n-th Fibonacci number
// Bad algorithm
public static void fib( int n ) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  else
    return fib( n-1 ) + fib( n-2 );
}

- Time complexity $O(2^n)$
Top-down DP (recursion + memoization)

```java
HashMap<Integer, Integer> map = new HashMap<>();
int fib(int n) {
    if(map.containsKey(n)){
        return map.get(n);
    }
    int res = 0;
    if(n > 0 && n <= 2) res = 1;
    else if(n > 2) res = fib(n - 1) + fib(n - 2);
    map.put(n, res);
    return res;
}
```

- $fib(k)$ only recurses the first time it is called
- Memoize and reuse solutions to subproblems to solve problem
- Subproblems: $fib(1)$, $fib(2)$, $..., fib(n)$
- Time complexity $O(n)$
int fib(int n) {
    int f[] = new int[n + 1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++){
        f[i] = f[i - 1] + f[i - 2];
    }
    return f[n];
}

- Time complexity $O(n)$
- When compute the nth number, the array already stores the previous two numbers. (Topological sort of subproblem dependency)
- To save space, we can also store the previous two numbers instead of the whole array
Steps to Solve DP

- Define subproblems (smaller version of the main problem)
- Relate subproblems solutions (usually recurrence)
- Recurse and memoize (or build up table bottom-up)
- Solve the original problem
Subproblems of Strings (Sequences)

- Suffixes $x[i:]$
- Prefixes $x[:i]$
- Substrings $x[i:j]$
Given sequence \( X = \{x_1, x_2, \ldots, x_m\} \), \( Y = \{y_1, y_2, \ldots, y_k\} \)

\( Y \) is a subsequence of \( X \) if there exists a strictly increasing sequence \( i_1, i_2, \ldots, i_k \) of indices of \( X \) such that \( x_{i_j} = y_j \).

For example, \( Y = \{B, C, D, B\} \) is a subsequence of \( X = \{A, E, B, C, B, D, A, B\} \).

Note that we do not assume the elements of \( Y \) are consecutive elements of \( X \).

In LCS problem, we are given two sequences \( X = \{x_1, x_2, \ldots, x_m\} \) and \( Y = \{y_1, y_2, \ldots, y_n\} \), and wish to find a maximum length common subsequence of \( X \) and \( Y \).

Brute force: \( O(2^m) \) UPDATED!
Longest Common Subsequence

- Find subproblems: prefixes or suffixes
- Analyze optimal substructure
  - Let $Z = \{z_1, z_2, \ldots, z_k\}$ be any LCS of $X$ and $Y$
  - If last characters of both sequences match: $x_m = y_n$, then
    $z_k = x_m = y_n$, and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$
  - If $x_m \neq y_n$, and $z_k \neq x_m$, then $Z$ is an LCS of $X_{m-1}$ and $Y$
  - If $x_m \neq y_n$, and $z_k \neq y_n$, then $Z$ is an LCS of $Y_{n-1}$ and $X$
- Recurrence
  - Let $c[i, j]$ be the length of the LCS of $X_i$ and $Y_j$
  - $c[i, j] = 0$, if $i = 0$ or $j = 0$
  - $c[i, j] = c[i - 1, j - 1] + 1$, if $i, j > 0$ and $x_i = y_j$
  - $\max\{c[i - 1, j], c[i, j - 1]\}$, if $i, j > 0$ and $x_i \neq y_j$
int lcs( char[] X, char[] Y, int m, int n ) {
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

int lcs_length(char[] X, char[] Y){
    return lcs(X, Y, X.length, Y.length);
}

This is a correct solution but it is very time consuming
Bottom-up DP

```java
int lcs( char[] X, char[] Y, int m, int n ){
    int dp[][] = new int[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] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[m][n];
}

int lcs_length(char[] X, char[] Y){
    return lcs(X, Y, X.length, Y.length);
}
```
### Example

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x_i$</th>
<th>$y_j$</th>
<th>$B$</th>
<th>$D$</th>
<th>$C$</th>
<th>$A$</th>
<th>$B$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**CLRS 15.8**
Why DP works?

- **Optimal substructure**
  - subproblems are just “smaller versions” of the main problem
  - for example, finding the LCS of two strings could be reduced to the problem of finding the LCS of two shorter strings (prefixes)
  - but the recursion is still expensive

- **Overlapping subproblems**
  - the same subproblem is encountered many times
  - we can solve each subproblem once and memoize the result
How similar are two strings?

Minimum number of edit operations to transform one string into the other string
  - Insertion
  - Deletion
  - Substitution

Example
  - cat to bat, output 1. Replace c with b
  - sunday to saturday, output 3. Insert a, insert t, replace n with r

Spell correction, computational biology, speech recognition, information extraction
Give two strings: $X_m = x_1x_2...x_m$, $Y_n = y_1y_2...y_n$

Define $D[i, j]$ as the edit distance between $X_i$ and $Y_j$

The edit distance between $X$ and $Y$ is thus $D[m, n]$

Subproblems: $D[i, j], \ i \leq m, j \leq n$

Recurrence:
- If $X_i = Y_j$, $D[i, j] = D[i - 1, j - 1]$
- If $X_i \neq Y_j$, $D[i, j] = \min(D[i - 1][j], D[i][j - 1], D[i - 1][j - 1]) + 1$
Bottom-up DP

```java
int editDistance(String str1, String str2, int m, int n) {
    int dp[][] = new int[m+1][n+1];
    for (int i=0; i<=m; i++) {
        for (int j=0; j<=n; j++) {
            // st1 is empty, insert all characters of str2
            if (i==0)
                dp[i][j] = j;
            else if (j==0)
                dp[i][j] = i;
            // If last characters are same
            else if (str1.charAt(i-1) == str2.charAt(j-1))
                dp[i][j] = dp[i-1][j-1];
            else // If last character are different
                dp[i][j] = 1 + min(dp[i][j-1], // Insert
                                    dp[i-1][j], // Remove
                                    dp[i-1][j-1]); // Replace
        }
    }
    return dp[m][n];
}
```
0-1 Knapsack Problem

- Given N items, pack the knapsack to get the maximum total value
- Each item has some weight \( w \) and some value \( v \)
- Total weight that we can carry is no more than some fixed number \( W \)
- 0-1 property, you can not break an item, either pick it or don’t

Maximize  \[
\sum_{k=1}^{N} x_k v_k
\]

Subject to:  \[
\sum_{k=1}^{N} x_k w_k \leq W, \ x_k \in \{0, 1\}
\]
0-1 Knapsack Problem

- **Original problem:** knapsack(N, W)
- **Subproblem:** knapsack(i, j), \( i \leq N, j \leq W \)
- **Recurrence:**
  - if \( wt[N-1] > W \): return knapsack(wt, val, W, N-1)
  - else return the max of:
    - knapsack(wt, val, W, N-1)
    - \( val[N - 1] + knapsack(wt, val, W - wt[N-1], N-1) \)
int knapsack(int wt[], int val[], int W, int N){
    if (N == 0 || W == 0)
        return 0;
    // If weight of the nth item is more than the capacity, then
    // this item cannot be included in the optimal solution
    if (wt[N-1] > W)
        return knapsack(wt, val, W, N-1);
    // Return the maximum of two cases:
    // nth item included or not included
    else return max( val[N-1] + knapsack(wt, val, W-wt[N-1], N-1),
                    knapsack(wt, val, W-wt[N-1], N-1) );
}
int knapsack(int wt[], int val[], int W, int N){
    int dp[][ ] = new int[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] = max(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];
}