Dynamic Programming

Xiyuan Yang

2025.10.05

MIT 6.006 Introduction to algorithms, focusing on **dynamic programming**.

Contents

§1. Introduction

MIT 6.006 Introduction to algorithms.

This section will focus on **Dynamic Programming**.

§2. Recursive Algorithms

§2.1. SRTBOT

1. Subproblem definition
2. Relate subproblem solutions recursively
3. Topological order on sub-problems (*to* subproblem textbf{DAG}, for dependencies for all the sub problems.)
4. Base cases of relation

5. Original problem solution via subproblem(s)
6. Time analysis

Example (Merge Sort).
1. Define sub problems as a function accepting parameters!
2. Sometimes, the huge problem itself is one kind of sub problems.
3. Key: Finding the related relation between different sub problems.
4. Base Case: The initial & final statement for recursion

Memorization: The computation of sub problems may in multiple times!
- e.g. Fibonacci Problems
- In simple sub-problem decomposition, $F(k)$ has been computed several times when solving $F(k+i), i > 0$.
- We need to use memorization to avoid repeated computation.

Recordings (Save memory and time).
- 空间换时间的算法思想，通过预处理把计算问题变成查询问题。
- 同样，也可以节省算法的空间复杂度。

Recordings (Important for finding relations).
- 动态规划问题的关键在于找到合适的子结构问题
- 从这个子结构出发，构建不同参数子规模的连接，可以从顶向下走递归，也可以自底向上走分治（本质还是递归的问题），中间可以做 mem 存储来节省时间复杂度。
- 使用 DAG 建模复杂问题。
  - 找到最优复杂度的问题本质可以看做是找 DAG 中的最短路问题！

§2.2. Reusing Subproblem Solutions

- Draw subproblem dependencies as a **DAG**
- How to solve them?
  - **Top down**: record subproblem solutions in a memo and re-use(**recursion + memoization**)
  - Bottom up: solve subproblems in **topological sort order** (usually via loops)
- For Fibonacci, n + 1 subproblems (vertices) and < 2n dependencies (edges)
- Time to compute is then O(n) additions

A subtlety is that Fibonacci numbers grow to $\Theta(n)$ bits long, potentially $\gg$ word size w. This means the number of bits needed to store the n-th Fibonacci number is proportional to n. When n is large, this number can be much bigger than the standard word size of your computer's CPU (e.g., 32 or 64 bits). Each addition costs $O(\lceil \frac{n}{w} \rceil)$ time, so total cost is:

$$O\left(n\left\lceil\frac{n}{w}\right\rceil\right) = O\left(n + \frac{n^2}{w}\right)$$

time.

§2.3. DAG Simulations

Recall for DAG problems:

Problem 2.3.1 (SSSP Problems for DAG).
Given a graph $G = (V, E)$ and a starting point $u \in V$. We need to **compute $f(u, i), \forall i \in V$.** We define $f(i, i) = 0$.

For an edge $u \to v$ with weight $w(u, v)$, we can find a shorter path using **relaxation**:

$$\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + w(u, v))$$

对于 DAG 来说，因为其独特的无环拓扑结构，使用拓扑排序来确定唯一的处理顺序，然后每个节点按照拓扑排序的顺序处理，最终保证每个节点处理的使用前驱节点已经处理过了。

Recordings (DFS and DP).

- 把问题分解成子问题之后再使用 DP 解决，本质上就是带记忆的 DFS Search。
- 递归函数本质上也可以看做是对一个 DAG 的 DFS 的过程。
- 在 DAG 建模的问题中，可以保证最终的出度为 0 的点只能是问题最后的结果，而入度为 0 的点是已知的结果。(比如拆分下来最小的结果)
  - 递归的过程本质上可以建模成对依赖关系的反向图的 DFS Search 过程，从大问题开始不断 搜索知道遇到出度（原图入度）为 0 的点。
- 说到底还是建模问题，要建模出结构良好的子结构问题。

§2.4. Example

Example (Bowling Example).

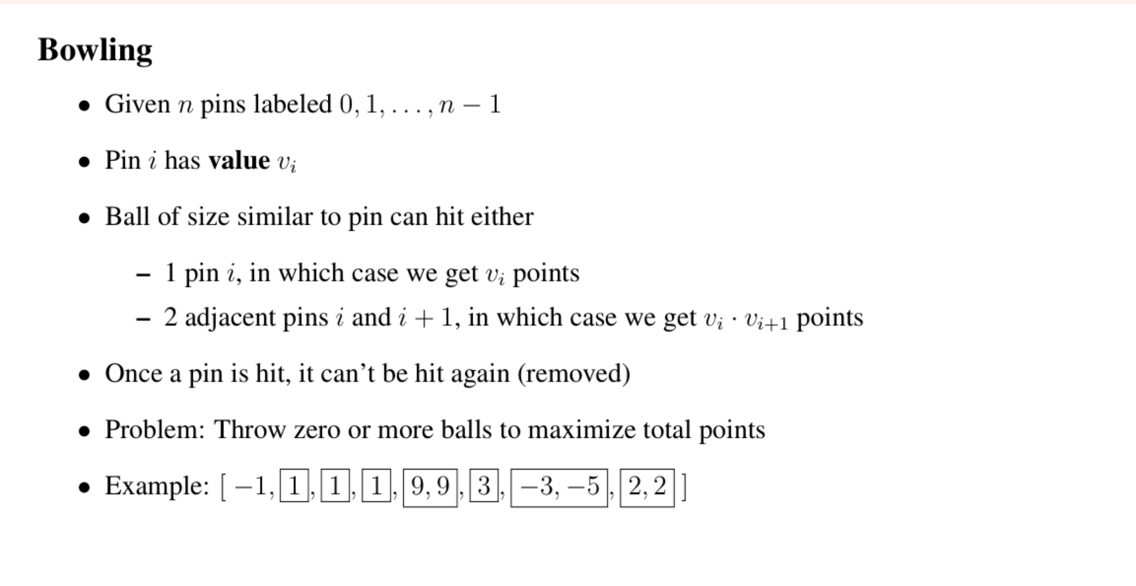

Figure 1: Bowling Problems Demo

Sub problems design:

IF the input is a sequence:

- Prefixes `x[:i]`, $O(n)$

- suffixes `x[i:]`, $O(n)$
- substrings `x[i:j]`

§2.4.1. Solution with suffixes

- Sub problem: $B(i) \to$ [i:]
- We need to compute B(0)
- relate:

$$B(i) = \max\{B(i+1), v_i + B(i+1), v_i \cdot v_i + 1 + B(i+2)\}$$

- Classical Bottom DP from bottom-up!

§2.5. Conclusion for relate subproblem solution

- The general approach we're following to define a relation on subproblem solutions:
  - Identify a question about a subproblem solution that, if you knew the answer to, would reduce to "smaller" subproblem(s)
    - In case of bowling, the question is "how do we bowl the first couple of pins?"
  - Then locally brute-force the question by trying all possible answers, and taking the best
    - In case of bowling, we take the max because the problem asks to maximize
  - Alternatively, we can think of correctly guessing the answer to the question, and directly recurring; but then we actually check all possible guesses, and return the "best"
- The key for efficiency is for the question to have a small (polynomial) number of possible answers, so brute forcing is not too expensive
- Often (but not always) the nonrecursive work to compute the relation is equal to the number of answers we're trying

§3. Dynamic Programming Sub-Problems: LCS & LIS & Coins

Recordings (Why Sub problems?).
- Recursion and Reuse
- When sub problems overlap
- careful brute force. (Or clever brute force)

§3.1. Longest Common Subsequence (LCS)

Problem 3.1.1 (Basic LCS).
Given two strings A and B, find a longest (**not necessarily contiguous**) subsequence of A that is also a subsequence of B.

Define sub problems for multiple inputs: using a matrix, which is the **product of multiple sub problem spaces**.

For example, in this case, we define two sub-problems which are the suffixes for string A and suffixes for string B, the final sub problem space is the doc product of two independent sub-problem spaces. Of course, we can define matrix with higher dimensions for more complex problems.

Thus, define:

$$L(i,j) = \text{LCS}(A[i:], B[j:]), 0 \le i \le |A|, 0 \le j \le |B|$$

For `A[|A|:]`, it means an empty string, it is easy to construct the initial statement.

Then, for the state transition equation:

We have:

$$\max(L(i,j+1), L(i+1,j)) \le L(i,j) \le \min(L(i,j+1), L(i+1,j)) + 1$$

if $A(i) == B(j)$:
- We can prove that `L(i,j) = L(i+1, j+1) + 1`

else:
- at least one of `A[i]` and `B[j]` are not in the LCS.
- Thus, transform this problem into smaller sub problems.
- $\max(L(i,j+1), L(i+1,j))$

Recordings (Finding state transition equation).
- 问题的关键在于新加入的字母,这些字母往往会在状态转移方程中出现或者作为分支的判定条件
- 要思考在什么状态下，该问题可以被修改成为更小的子问题进行运算

§3.2. Longest Increasing Subsequence (LIS)

Problem 3.2.1 (LIS problems).
Given a string A, find a longest (not necessarily contiguous) subsequence of A that strictly increases (lexicographically)

Still using the suffixes:

$$L(i) = \text{LIS}(A[i:])$$

- final statement, we need to solve `L(0)`
- initial statement, `L(length_A) = 0` for it is an empty string.
- transition:
  - if i is in the longest sequence:
    - `L(i) = L(i+1) + 1`
  - if not:
    - `L(i) = L(i+1)`

However, it is not easy to find the "if" statement, thus, we need to **change the definition of sub problems**

We define: $x(i)$ = length of longest increasing subsequence of suffix `A[i :]` that includes `A[i]`. Then, we solve it again:

- final statement: result is the maximum value of $\{x(i) | i \in \{1, 2, 3, ..., n\}\}$
- initial statement: `x(length_A)` $= 0$
- transition:
  - if `s[i]` $<$ `s[i+1]`:
    - `x(i) = x(i+1) + 1`
  - else:
    - it is still difficult
    - $x(i) = \max\{1 + x(j) \mid i < j < |A|, A[j] > A[i]\} \cup \{1\}$
    - We need to traverse the processed string, which concat thr added `s[i]` into the current strings.

§3.3. Alternating Coin Game

Problem 3.3.1 (Alternating coin games).
Given sequence of n coins of value $v_0, v_1, ..., v_{n-1}$
- Two players ("me" and "you") take turns
- In a turn, take first or last coin among remaining coins
- My goal is to maximize total value of my taken coins, where I go first

The structure of the sub-problems are quite simple: we just need to define $L(i, j)$ as the optimal total value I can get for the substring of `[i..j]`$(i \leq j)$

- final statement: `L[0, n-1]`
- initial statement: `L[i,i]`

However, it is hard to write the transition!

Thus, we will change to another solution:

$x(i, j, p)$ = maximum total value I can take when player $p \in \{\text{me}, \text{you}\}$ starts from coins of values $v_i ... v_j$.
- me is p = 0
- you is p = 1

Recordings (Adding a new dimension).
We can add a new dimension when solving complex dp tasks.

- final statement: $x(0, n-1, 0)$
- initial statement:
  - $x(i, i, 0) = s[i]$
  - $x(i, i, 1) = 0$
- Then, the transition is:

$$L(i, j, 0) = \max(L(i, j-1, 1) + a[j], L(i+1, j, 1) + a[i])$$

$$L(i, j, 1) = \min(L(i, j-1, 0) + a[j], L(i+1, j, 0) + a[i])$$

- We must assume the component is clever enough.

Thus, we finish this problem in $O(n^2)$ running time.

§4. SubProblems Constraints and Expansions

§4.1. Bellman-Ford Expansion SSSP

§4.1.1. DAG Shortest Path

It is time when we solve the DAG problems using relaxation!

- Define sub problems:

$\delta_k(s, v)$ means the weight of shortest path from s to v using at most k edges.

- We want to solve:

$$\delta_{|E|}(s, v)$$

- What we have:

$$\delta_0(s, v) = 0, \forall v \in V$$

$$\delta_i(s, s) = 0, \forall i \in \{0, 1, 2, ..., |V|\}$$

- Status transform

$$\delta(s, v) = \min\{\delta(s, u) + w(u, v) \mid u \in \mathrm{Adj}^-(v)\} \cup \{\delta_{k-1}(s, u)\}$$

Recordings (When to use DP?).

- 动态规划的状态转移方程经常出现 min max 等求最大最小值的组合
- 这是因为动态规划经常子问题定义的是一个最优化问题
- 而最优化问题的处理基本逻辑就是暴力枚举+取最值
  - 因此检查动态规划的正确性（尤其遇到最大最小值）可以看所有枚举情况是否被包含
  - 这也是为什么更复杂的动态规划需要分类讨论而不可以直接取最值，因为有时候并不是简单的枚举！

§4.2. APSP

For all pairs shortest path: **Floyd-Warshall**

- For simple SSSP:
  - $O(|V|^2\ |E|) = O(|V|^4)$

$d(u, v, k)$ = minimum weight of a path from u to v that only uses vertices from $\{1, 2, ..., k\} \cup \{u, v\}$

- What we want to solve: $d(u, v, |V|)$
- What we have:
  - $d(u, v, 0) = w(u, v)$ if $(u, v) \in E$
  - $d(u, v, 0) = w(u, v)$, otherwise
- transformation:
  - If k is in the shortest path: $d(u, v, k) = d(u, k, k - 1) + d(k, v, k - 1)$
  - If not: $d(u, v, k) = d(u, v, k - 1)$

$$d(u,v,k) = \min(d(u,v,k-1), d(k,v,k-1) + d(u,k,k-1))$$

- Time complexity: $O(|V|^3)$

§4.3. Arithmetic Parenthesization

Problem 4.3.1 (Arithmetic Parenthesization).

- 给定一个数字序列，两个数字之间存在加号或者乘号
- 你可以任意改变运算优先级（加括号）
- 求解：最终最大的输出值
- Allow negative numbers

Idea: find the operations from the root. Sub-Problems: using substring!

Define sub problems:

$x(i, j \text{ opt})$ = opt value with sub string from [i..j)

- $0 \le i, j \le n$ and opt $\in \{\min, \max\}$
- What we have:
  - $x(i, i, \text{opt}) = 0$
  - $x(i, i+1, \text{opt}) = a_i$
- Original Problems: $x(0, n, \max)$

$$x(i, j, \text{opt}) = \text{opt } \{x(i, k, \text{opt}') * kx(k, j, \text{opt}'')) \mid i < k < j; \text{opt}', \text{opt}'' \in \{\min, \max\}\}$$

§4.4. Piano Fingering

Problem 4.4.1 (Piano Fingering).

Piano Fingering

- Given sequence $t_0, t_1, \ldots, t_{n-1}$ of n **single** notes to play with right hand (will generalize to multiple notes and hands later)
- Performer has right-hand fingers $1, 2, \ldots, F$ ($F = 5$ for most humans)
- Given metric $d(t, f, t', f')$ of **difficulty** of transitioning from note t with finger f to note t' with finger f'
  - Typically a sum of penalties for various difficulties, e.g.:
  - $1 < f < f'$ and $t > t'$ is uncomfortable
  - Legato (smooth) play requires $t \ne t'$ (else infinite penalty)
  - Weak-finger rule: prefer to avoid $f' \in \{4, 5\}$
  - $\{f, f'\} = \{3, 4\}$ is annoying
- Goal: Assign fingers to notes to minimize total difficulty

Figure 2: Piano Fingering

$x(i, f)$ = minimum total difficulty for playing notes t_i, t_{i+1}, . . . , tn−1 starting with finger f on note t_i.

Define: $x(i, f)$ is $\min\{x(i+1, f') + d(t_i, f, t_{i+1}, f') \mid 1 \le f' \le F\}$

Time Complexity: $O(n \times F^2)$

§4.5. Guitar Fingering

Problem 4.5.1 (Guitar Fingering).

Guitar Fingering

- Up to S = number of strings different ways to play the same note
- Redefine "finger" to be tuple (finger playing note, string playing note)
- Throughout algorithm, F gets replaced by $F \cdot S$
- Running time is thus $\Theta(n \cdot F^2 \cdot S^2)$

Figure 3: Guitar Fingering

$F \to F \times S$

Time Complexity: $O(nF^2S^2)$

§4.5.1. Multiple Notes at Once

t_i is a set of notes to play at time i.

f_i is a mapping of placing notes with different fingers: $t_i \to \{1, 2, 3, ..., F\}$

For each fingering f_i, at most T^F choices, $T = \max_i |t_i|$

Time Complexity: $O(nT^{2F})$

§5. Pseudopolynomial

Definition 5.1 (Pseudopolynomial Time).

- 真多项式时间：$T = O(\mathrm{poly}(N))$，其中 N 代表着输入数据的编码长度。
  - 编码长度就是二进制字符串的长度
- 伪多项式时间：$T = O(\mathrm{poly}(U))$，其中 U 代表这个输入数据的最大值。
  - 例如经典的背包问题

§5.1. Rod Cutting

Problem 5.1.1 (Rod Cutting Problems).

- 给定一个长度为 L 的钢条，现在允许将钢条切割成若干长度
- 每一个长度的子钢条都对应一个价值
- 求最大的价值长度

经典的动态规划算法：

$$X(l) = \max\{v(p) + X(l-p) \mid p \in [1, l]\}$$

§5.2. SubSet Sum

Problem 5.2.1 (Subset Sum problem).
- 给定一个集合有 n 个数字
- 求解集合的子集，子集内所有元素的和为目标数字 target

动态规划本身并不复杂：

$x(i, t)$: any subset of $A[i:]$ sum to t

- What we have:
  - $x(i, 0)$ = True
  - $x(n, t)$ = False (t != 0)
- Transformation:

$$x(n, t) = x(n+1, t) \vee (t - a[n] \geq 0 \wedge x(n+1, t - a[n]))$$

Time Complexity: $O(nT)$

Is it polynomial?

- Input size: $n + 1$, not polynomial in input size!
- for w-bit word RAM model: $T \leq 2^w$ and $w \geq \log(n+1)$
- for the least cases: $w \approx n$
- Then the time complexity: $O(n2^n)$

Recordings (Why not polynomial?).
- 在这里很复杂的一点是算法的运行时间同时和两个变量决定
  - 输入的数据量
  - 给定的目标值的大小
- 因此，这不是一个单变量的多项式运行时间
- 而对于输入编码 w 来说，这个数字是指数级增长的！

§6. Conclusion

Main Features of Dynamic Programs

- Review of examples from lecture
- **Subproblems:**
  - **Prefix/suffixes:** Bowling, LCS, LIS, Floyd–Warshall, Rod Cutting (coincidentally, really Integer subproblems), Subset Sum
  - **Substrings:** Alternating Coin Game, Arithmetic Parenthesization
  - **Multiple sequences:** LCS
  - **Integers:** Fibonacci, Rod Cutting, Subset Sum
    - **Pseudopolynomial:** Fibonacci, Subset Sum
  - **Vertices:** DAG shortest paths, Bellman–Ford, Floyd–Warshall
- **Subproblem constraints/expansion:**
  - **Nonexpansive constraint:** LIS (include first item)
  - $\mathbf{2\times}$ **expansion:** Alternating Coin Game (who goes first?), Arithmetic Parenthesization (min/max)
  - $\mathbf{\Theta(1)\times}$ **expansion:** Piano Fingering (first finger assignment)
  - $\mathbf{\Theta(n)\times}$ **expansion:** Bellman–Ford (# edges)
- **Relation:**
  - **Branching** = # dependant subproblems in each subproblem
  - $\mathbf{\Theta(1)}$ **branching:** Fibonacci, Bowling, LCS, Alternating Coin Game, Floyd–Warshall, Subset Sum
  - $\mathbf{\Theta(\text{degree})}$ **branching** (source of $|E|$ in running time): DAG shortest paths, Bellman–Ford
  - $\mathbf{\Theta(n)}$ **branching:** LIS, Arithmetic Parenthesization, Rod Cutting
  - **Combine multiple solutions (not path in subproblem DAG):** Fibonacci, Floyd–Warshall, Arithmetic Parenthesization
- **Original problem:**
  - **Combine multiple subproblems:** DAG shortest paths, Bellman–Ford, Floyd–Warshall, LIS, Piano Fingering

Figure 4: Summarization for all DP problems