Dynamic Programming

Arnav Gupta

April 9, 2024

Contents

1 Dynamic Programming

1.1 Recipe

1.1.1 Identify the Subproblem

Typically the computation of solutions of the subproblems will make it natural to retain the solutions in an array:

- must know dimensions of the array

- specify the precise meaning of the value in any cell of the array
- specify where the answer will be found in the array

1.1.2 Establish DP-recurrence

Specify how a subproblem contributes to the solution of a larger subproblem.

How the value in a cell of the array depends on the values of other cells in the array.

1.1.3 Set Values for the Base Case

Define the base values for the DP-recurrence.

1.1.4 Specify the Order of Computation

The algorithm must state the order of computation for the cells.

1.1.5 Recovery of the Solution

Keep track of the subproblems that provided the best solutions. Use a traceback strategy to determine the full solution.

1.2 Key Features

- solve problems through recursion
- use a small (polynomial) number of nested subproblems
- may have to store results for all subproblems
- can often be turned into 1+ loops

1.3 DP vs Divide-and-Conquer

- dynamic programming usually deals with all input sizes from 1 to n
- divide-and-conquer may not solve subproblems
- divide-and-conquer algorithms not easy to rewrite iteratively

2 Interval Scheduling

Input: n intervals of form $I_i = [s_i, f_i]$, each interval has weight w_i

Output: a choice T of intervals that do not overlap and maximizes total weight

Greedy algorithm works in the case where all weights are 1.

2.1 Sketch of the Algorithm

Basic Idea: choose I_n or not, then choose the max of

- $w_n + O(I_{m1}, \ldots, I_{ms})$ if we choose I_n where $I_{m1}, \ldots, I_{ms}$ are the intervals that do not overlap with I_n
- $O(I_1, \ldots, I_{n-1})$ if we don't choose I_n

Goal:

- find a way to ensure that $I_{m1}, \ldots, I_{ms}$ are of the form $I_1, \ldots, I_s$ for some $s < n$
- it then suffices to optimize over all $I_1, \ldots, I_j$ where $1 \leq j \leq n$

Assume $I_1, \ldots, I_n$ sorted by increasing end time: $f_i \leq f_{i+1}$.

Claim: for all j, the set of $I_k \leq I_j$ that do not overlap I_j is of the form $I_1, \ldots, I_{p_j}$ for some $0 \leq p_j \leq j$ where $p_j = 0$ if no such interval exists

The algorithm needs all p_i, and this can be found by comparing with finish times.

2.2 Finding pj's

Let A be a permutation of $[1, ..., n]$ such that $s_{A[1]} \leq s_{A[2]} \leq \cdots \leq s_{A[n]}$.

```
FindPj(A, s1, ..., sn, f1, ..., fn):
  f0 = -infinity
  i = 1
  for k in range(0, n):
      while i <= n and fk <= s[A[i]] < f[k+1]:
          pi = k
          i++
```

Runtime: $O(n \log(n))$ for sorting and $O(n)$ for loops

2.3 Main Procedure

Definition: M[i] is the maximal weight to get with $I_1, \ldots, I_i$

Recurrence: M[0] $= 0$ and for $i \geq 1$, $M[i] = \max(M[i-1], M[pi] + wi)$

Runtime: $O(n \log(n))$ for sorting twice and $O(n)$ for finding all $M[i]$

3 0/1 Knapsack Problem

Input: items from 1 to n with weights w_i and values v_i, along with a capacity W

Output: a subset of the items S that has total weight less than W and maximizes total value

Basic idea: either choose item n or not, then the optimum is the max of:

- $v_n + O[W - w_n, n-1]$ if we choose n
- $O[W, n-1]$ if we don't choose n

Initial conditions: $O[0, i] = 0$ for any i and $O[w, 0] = 0$ for any w

```
01KnapSack(v1, ..., vn, w1, ..., wn, W):
  initialize an array O[0..n, 0..W] with O(0, j) = 0 and O(w, 0) = 0
  for i in range(1, n):
      for w in range(1, W):
          if wi > w:
              O[w, i] = O[w, i-1]
          else:
              O[w, i] = max(vi + O[w - wi, i-1], O[w, i - 1])
```

Runtime: $\Theta(nW)$ which is pseudo-polynomial

3.1 Pseudo-Polynomial Algorithms

In our word RAM model, we assume all v_i and w_i fit in a word, so the input size is $\Theta(n)$ words, but the runtime also depends on the values of the inputs.

01-knapsack is NP-complete.