CS 341: Algorithms
Lec 11: Dynamic Programming

Armin Jamshidpey   Collin Roberts

Based on lecture notes by Éric Schost

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2025

Goals

This module: the dynamic programming paradigm through examples

- interval scheduling, longest increasing subsequence, longest common subsequence, etc

Computational model:

- word RAM
- assume all weights, values, capacities, deadlines, etc, fit in a word

What about the name?

- **programming** as in **decision making**
- **dynamic** because it sounds cool.

A slow recursive algorithm

Def: Fibonacci numbers

- $F_0 = 0$, $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

Fib(n)

1. **if** $n = 0$ **return** 0
2. **if** $n = 1$ **return** 1
3. **return** $\text{Fib}(n-1) + \text{Fib}(n-2)$

Assuming we count additions **at unit cost**, runtime is

$$T(0) = T(1) = 0, \quad T(n) = T(n-1) + T(n-2) + 1$$

This gives $T(n) = F(n+1) - 1$, so $\boldsymbol{T(n) \in \Theta(\varphi^n)}$, $\varphi = (1+\sqrt{5})/2$.

A better algorithm

Observations

- to compute F_n, we only need the values of $F_0, \ldots, F_{n-1}$
- the algorithm recomputes them many, many times

Improved recursive algorithm

let $T = [0, 1, \bullet, \bullet, \ldots]$ be a global array
Fib(n)
1. **if** $T[n] = \bullet$
2. $\quad T[n] = \text{Fib}(n-1) + \text{Fib}(n-2)$
3. **return** $T[n]$

A better algorithm

Iterative version

Fib(n)

1. let $T = [0, 1, \bullet, \bullet, \ldots]$
2. **for** $i = 2, \ldots, n$
3. $\quad T[i] = T[i-1] + T[i-2]$
4. **return** $T[n]$

A better algorithm

Iterative version (enhanced, not always feasible)

Fib(n)
1. $(u, v) \leftarrow (0, 1)$
2. **for** $i = 2, \ldots, n$
3. $\quad (u, v) \leftarrow (v, u + v)$
4. **return** v

All these improved versions use $\boldsymbol{O(n)}$ additions

Main feature: solve "subproblems" bottom up, and store solutions if needed.

A Recipe for Designing a D. P. Algorithm

1. **Identify the subproblem**
   Typically the computation of solutions of the subproblems will make it natural to retain the solutions in an array.
   - Need to 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
2. **Establish DP-recurrence**
   Specify how a subproblem contributes to the solution of a larger subproblem. How does the value in a cell of the array depend on the values of other cells in the array?
3. **Set values for the base cases**
4. **Specify the order of computation**
   The algorithm must clearly state the order of computation for the cells.
5. **Recovery of the solution (if needed)**
   Keep track of the subproblems that provided the best solutions. Use a traceback strategy to determine the full solution.

Dynamic programming

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 one (or more) loops

Dynamic programming vs divide-and-conquer

- dynamic programming usually deals with all input sizes $1, \ldots, n$
- DAC may not solve "subproblems"
- DAC algorithms not easy to rewrite iteratively

The Interval scheduling Problem

Input:

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- each interval has a weight $\boldsymbol{w_i}$

Output:

- a choice T of intervals that **do not overlap** and **maximizes** $\sum_{i \in T} w_i$
- greedy algorithm in the case $w_i = 1$

Example: A car rental company has the following requests for a given day:

- $I_1 = [2, 8]$, $w_1 = 6$
- $I_2 = [2, 4]$, $w_2 = 2$
- $I_3 = [5, 6]$, $w_3 = 1$
- $I_4 = [7, 9]$, $w_4 = 2$

Answer is $T = [I_1]$, $W = 6$

Sketch of the algorithm

Basic idea: either we choose I_n or not.

- then the optimum $\boldsymbol{O(I_1, \ldots, I_n)}$ is the max of two values:
- $\boldsymbol{w_n + O(I_{m_1}, \ldots, I_{m_s})}$, if we choose I_n, where $I_{m_1}, \ldots, I_{m_s}$ are the intervals that do not overlap with I_n
- $\boldsymbol{O(I_1, \ldots, I_{n-1})}$, if we don't choose I_n

In general, we don't know what $I_{m_1}, \ldots, I_{m_s}$ look like.

Goal:

- find a way to ensure that $I_{m_1}, \ldots, I_{m_s}$ are of the form $I_1, \ldots, I_s$, for some $s < n$
  (and so on for all indices $< n$)
- then it suffices to optimize over all $I_1, \ldots, I_j$, $j = 1, \ldots, n$

The indices p_j

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

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

The algorithm will need the p_i's.

- if $-\infty \leq s_i < f_1$, $\boldsymbol{p_i = 0}$ — f_1= earliest finish time
- if $f_1 \leq s_i < f_2$, $\boldsymbol{p_i = 1}$
- ...

(we will write $f_0 = -\infty$)

Computing the p_j's

let A be a permutation of $[1, \ldots, n]$ such that

$$s_{A[1]} \leq s_{A[2]} \leq \cdots \leq s_{A[n]}$$

Exercise: make sure you know how to find such an A

```
FindPj(A, s_1, ..., s_n, f_1, ..., f_n)
    f_0 ← −∞
    i ← 1
    for k = 0, ..., n
        while i ≤ n and f_k ≤ s_{A[i]} < f_{k+1}
            p_i ← k
            i++
```

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

Main procedure

Definition: $\boldsymbol{M[i]}$ is the maximal weight we can get with intervals $I_1, \ldots, I_i$

Recurrence: $M[0] = 0$ and for $i \geq 1$

$$\boldsymbol{M[i] = \max(M[i-1], M[p_i] + w_i)}$$

Runtime: $\boldsymbol{O(n \log(n))}$ (sorting twice) and $\boldsymbol{O(n)}$ (finding the $M[i]$'s)

Exercise: recover the optimum set for an extra $O(n)$

The 0/1 Knapsack Problem

Input:

- items $1, \ldots, n$ with **weights** $w_1, \ldots, w_n$ and **values** $v_1, \ldots, v_n$
- a **capacity** W

Output:

- a choice of items $S \subset \{1, \ldots, n\}$
- that satisfies the constraint $\sum_{i \in S} w_i \leq W$
- and maximizes the value $\sum_{i \in S} v_i$

Example:

- $w_1 = 3, w_2 = 4, w_3 = 6, w_4 = 5$
- $v_1 = 2, v_2 = 3, v_3 = 1, v_4 = 5$
- $W = 8$
- optimum $S = \{1, 4\}$ with weight 8 and value 7

See also:

- fractional knapsack (items can be divided), solved with a greedy algorithm

Setting up the recurrence

Basic idea: either we choose item n or not.

- then the optimum $\boldsymbol{O[W,n]}$ is the max of two values:
- $\boldsymbol{v_n + O[W - w_n, n-1]}$, if we choose n (and $w_n \leq W$)
- $\boldsymbol{O[W, n-1]}$, if we don't choose n

$$O[w,i] := \text{maximum value achievable using a knapsack of capacity } w \text{ and items } 1, \ldots, i$$

Initial conditions

- $O[0,i] = 0$ for any i
- $O[w,0] = 0$ for any w

Algorithm

01KnapSack$(v_1, \ldots, v_n, w_1, \ldots, w_n, W)$

```
initialize an array O[0..W, 0..n]
with all O(0, j) = 0 and all O(w, 0) = 0
for i = 1, ..., n
    for w = 1, ..., W
        if w_i > w
            O[w, i] ← O[w, i − 1]
        else
            O[w, i] ← max(v_i + O[w − w_i, i − 1], O[w, i − 1])
```

Runtime $\Theta(nW)$.

Discussion

This is called a **pseudo-polynomial** algorithm

- in our word RAM model, we have been assuming all v_is and w_is fit in a word
- so input size is $\Theta(n)$ words
- but the runtime also depends on the **values** of the inputs

01-knapsack is **NP-complete**, so we don't really expect to do much better