CS 341: Algorithms

Lecture 7: Dynamic programming

Éric Schost

based on lecture notes by many other CS341 instructors

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

Fall 2024

Goals

This module: the dynamic programming paradigm through examples

- weighted interval scheduling, knapsack, 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.

Warmup example: Fibonacci numbers

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

A better algorithm

Observations

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

A better algorithm

Observations

- to compute F_n, we 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. $\quad T[n] = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2)$
3. **return** $T[n]$

A better algorithm

Observations

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

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

Observations

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

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

A better algorithm

Observations

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

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 $\boldsymbol{\Theta(n)}$ additions

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

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 always easy to rewrite iteratively

Recipe

- **Identify subproblems** and (typically) store their solutions in an array.
  Need to know:
  - dimensions of the array
  - what precisely the array stores
  - where the answer will be found
- **Establish recurrence**
  - how do small subproblems contribute to the solution of a larger one?
- **Find the base case(s)**
- **Specify the order of computation**
- **Recovery of the solution**
  - traceback strategy to determine the final solution

Weighted interval scheduling

The 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

Sketch of the algorithm

Basic idea: either we choose I_n or not.

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

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

Goal:

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

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 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_j's.

- for a given j, find where s_j would be in $[f_1, \ldots, f_n]$
- precisely: p_j is the last index i such that $f_i \leq s_j$
- binary search, so $O(n \log(n))$ total.

Note: still OK if repeated f_i's

Main procedure

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

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

$$\boldsymbol{M[j] = \max(M[j-1], M[p_j] + w_j)}$$

Runtime: $\boldsymbol{\Theta(n \log(n))}$ (sorting, p_j's) and $\boldsymbol{\Theta(n)}$ (finding the $M[j]$'s)

Exercise

recover the optimum set, not only $M[n]$, for extra $\Theta(n)$

0/1 knapsack

The 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 $\sum_{i \in S} v_i$

The 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

The 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

Set $\boldsymbol{O[w,i]}$:= maximum value achievable using a knapsack of capacity w, items $1,\dots,i$

Want: $\boldsymbol{O[W,n]}$

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 (requires $w_n \le W$)
- $\boldsymbol{O[W, n-1]}$, if we don't choose n

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 $i = 1, \ldots, n$
    for $w = 1, \ldots, W$
        if $w_i > w$
            $O[w, i] \leftarrow O[w, i-1]$
        else
            $O[w, i] \leftarrow \max(v_i + O[w - w_i, i-1], O[w, i-1])$

Runtime $\Theta(nW)$.

Discussion

1. **Runtime.** This is called a **pseudo-polynomial** algorithm
   - in our word RAM model, we have been assuming all v_i's, w_i's and W 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

2. Exercise

recover the optimum subset

A related problem

Subset sum: given positive integers $a_1, \ldots, a_n$ and integer K, find if there is $S \subseteq \{1, \ldots, n\}$ with

$$\sum_{i \in S} a_i = K$$

Option 1: write a "new" algorithm

- very much like knapsack
- pseudo polynomial runtime $\Theta(nK)$

Option 2: use the knapsack algorithm with

- $w_1, \ldots, w_n = a_1, \ldots, a_n$
- $v_1, \ldots, v_n = a_1, \ldots, a_n$
- $W = K$