Lecture 7: Dynamic Programming I

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

September 28, 2023

Overview

- Dynamic Programming
  - General Paradigm
  - Simple example: Fibonacci
- Weighted Interval Scheduling
  - Solution with Dynamic Programming
  - Principles of Dynamic Programming
- Subset-Sum & Knapsack
  - Subset-Sum
  - Knapsack
- Acknowledgements

General Paradigm

- Sometimes, when trying a divide and conquer approach, we are only able to divide in a way which makes us perform "exhaustive search"

Looks like it is going to be a bad divide and conquer

Dynamic Programming.

General Paradigm

- Sometimes, when trying a divide and conquer approach, we are only able to divide in a way which makes us perform "exhaustive search"

  Looks like it is going to be a bad divide and conquer

- However, in several situations, it turns out that a *small set* of particular subproblems appear *several times* in our recurrence

Dynamic Programming.

General Paradigm

- Sometimes, when trying a divide and conquer approach, we are only able to divide in a way which makes us perform "exhaustive search"

  Looks like it is going to be a bad divide and conquer

- However, in several situations, it turns out that a *small set* of particular subproblems appear *several times* in our recurrence
- Instead of recomputing the subproblems, we can:
  1. solve them once
  2. save them to memory (memoization)
  3. and if we need them again, we already precomputed them! (savings)

Dynamic Programming.

Fibonacci Sequence

- Fibonacci sequence

$$F(n) = F(n-1) + F(n-2)$$

with $F(0) = F(1) = 1$

Fibonacci Sequence

- Fibonacci sequence

$$F(n) = F(n-1) + F(n-2)$$

  with $F(0) = F(1) = 1$
- Exponential recursion tree (see board)

Looks like we can't compute this!

Fibonacci Sequence

- Fibonacci sequence

$$F(n) = F(n-1) + F(n-2)$$

  with $F(0) = F(1) = 1$

- Exponential recursion tree (see board)

Looks like we can't compute this!

- Wait a second, many subproblems are the same!

Can compute everything with *much smaller* subtree!

Fibonacci Sequence

- Fibonacci sequence

$$F(n) = F(n-1) + F(n-2)$$

  with $F(0) = F(1) = 1$
- Exponential recursion tree (see board)

Looks like we can't compute this!

- Wait a second, many subproblems are the same!

Can compute everything with *much smaller* subtree!

- Essence of Dynamic Programming.

Fibonacci Sequence

- Fibonacci sequence

$$F(n) = F(n-1) + F(n-2)$$

with $F(0) = F(1) = 1$

- Exponential recursion tree (see board)

Looks like we can't compute this!

- Wait a second, many subproblems are the same!

Can compute everything with *much smaller* subtree!

- Essence of Dynamic Programming.
- **Remark on output size:** note here that word RAM is no longer appropriate, as the input can be given with $O(\log n)$ bits (say by giving n, $F(0) = F(1) = 1$, which takes $O(\log n)$ bits). But output size is $\exp(n)$, which takes $O(n)$ bits (which in this case is exponential time).

Weighted interval scheduling

- **Input:** n intervals with weights, denoted $[(s_1, f_1), w_1], \ldots, [(s_n, f_n), w_n]$
- **Output:** subset of non-overlapping intervals of *maximum* weight
- **Model:** Word RAM

Weighted interval scheduling

- **Input:** n intervals with weights, denoted $[(s_1, f_1), w_1], \ldots, [(s_n, f_n), w_n]$
- **Output:** subset of non-overlapping intervals of *maximum* weight
- Why does greedy not work?

Weighted interval scheduling

- **Input:** n intervals with weights, denoted $[(s_1, f_1), w_1], \ldots, [(s_n, f_n), w_n]$
- **Output:** subset of non-overlapping intervals of *maximum* weight

- Let's try a recursive approach.
  - Sort items by finishing time, so can assume $f_1 \leq f_2 \leq \cdots \leq f_n$
  - For each interval j, let $p(j)$ be largest index $i < j$ such that $f_i < s_j$.

Weighted interval scheduling

- **Input:** n intervals with weights, denoted $[(s_1, f_1), w_1], \ldots, [(s_n, f_n), w_n]$
- **Output:** subset of non-overlapping intervals of *maximum* weight

- Let's try a recursive approach.
  - Sort items by finishing time, so can assume $f_1 \leq f_2 \leq \cdots \leq f_n$
  - For each interval j, let $p(j)$ be largest index $i < j$ such that $f_i < s_j$.
- Given optimal solution S, two possibilities: either $n \in S$ or $n \notin S$.

Weighted interval scheduling

- **Input:** n intervals with weights, denoted $[(s_1, f_1), w_1], \ldots, [(s_n, f_n), w_n]$
- **Output:** subset of non-overlapping intervals of *maximum* weight

- Let's try a recursive approach.
  - Sort items by finishing time, so can assume $f_1 \leq f_2 \leq \cdots \leq f_n$
  - For each interval j, let $p(j)$ be largest index $i < j$ such that $f_i < s_j$.
- Given optimal solution S, two possibilities: either $n \in S$ or $n \notin S$.
- Letting $\mathrm{weight}(j)$ be the weight of optimal solution to problem $[(s_1, f_1), w_1], \ldots, [(s_j, f_j), w_j]$, we have

$$\mathrm{weight}(n) = \max\{w_n + \mathrm{weight}(p(n)), \mathrm{weight}(n-1)\}$$

Weighted interval scheduling

- **Input:** n intervals with weights, denoted $[(s_1, f_1), w_1], \ldots, [(s_n, f_n), w_n]$
- **Output:** subset of non-overlapping intervals of *maximum* weight

- Let's try a recursive approach.
  - Sort items by finishing time, so can assume $f_1 \leq f_2 \leq \cdots \leq f_n$
  - For each interval j, let $p(j)$ be largest index $i < j$ such that $f_i < s_j$.
- Given optimal solution S, two possibilities: either $n \in S$ or $n \notin S$.
- Letting $\mathrm{weight}(j)$ be the weight of optimal solution to problem $[(s_1, f_1), w_1], \ldots, [(s_j, f_j), w_j]$, we have

$$\mathrm{weight}(n) = \max\{w_n + \mathrm{weight}(p(n)), \mathrm{weight}(n-1)\}$$

Looks like a bad divide and conquer.
Imagine if $p(j) = j - 2$ for each j!

Weighted interval scheduling

- **Input:** n intervals with weights, denoted $[(s_1, f_1), w_1], \ldots, [(s_n, f_n), w_n]$
- **Output:** subset of non-overlapping intervals of *maximum* weight

- Let's try a recursive approach.
  - Sort items by finishing time, so can assume $f_1 \leq f_2 \leq \cdots \leq f_n$
  - For each interval j, let $p(j)$ be largest index $i < j$ such that $f_i < s_j$.
- Given optimal solution S, two possibilities: either $n \in S$ or $n \notin S$.
- Letting $\mathrm{weight}(j)$ be the weight of optimal solution to problem $[(s_1, f_1), w_1], \ldots, [(s_j, f_j), w_j]$, we have

$$\mathrm{weight}(n) = \max\{w_n + \mathrm{weight}(p(n)), \mathrm{weight}(n-1)\}$$

- How can we solve such recurrences efficiently?

Dynamic Programming: Memoization

- Note that although the recurrence might be bad from a divide and conquer point of view, we only need to solve n different subproblems!

Many repetitions in recursion tree!

Dynamic Programming: Memoization

- Note that although the recurrence might be bad from a divide and conquer point of view, we only need to solve n different subproblems!

Many repetitions in recursion tree!

- If we can store the solution to a subproblem when we encounter it, we *don't need to solve it again*! (*Memoization*)

Dynamic Programming: Memoization

- Note that although the recurrence might be bad from a divide and conquer point of view, we only need to solve n different subproblems!

  Many repetitions in recursion tree!

- If we can store the solution to a subproblem when we encounter it, we *don't need to solve it again*! (*Memoization*)
- With this at hand, we note that we only need to compute the subproblems $\mathrm{weight}(j)$, for $0 \le j \le n$.
  Moreover, if have solutions to $\mathrm{weight}(k)$ for all $k < j$, we can obtain $\mathrm{weight}(j)$ by the recursion:

  $$\mathrm{weight}(j) = \max\{w_j + \mathrm{weight}(p(j)), \mathrm{weight}(j-1)\}$$

  which takes $O(1)$ time to compute, when we have the values $\mathrm{weight}(j-1)$ and $\mathrm{weight}(p(j))$

Dynamic Programming: Memoization

- Note that although the recurrence might be bad from a divide and conquer point of view, we only need to solve n different subproblems!

  Many repetitions in recursion tree!

- If we can store the solution to a subproblem when we encounter it, we *don't need to solve it again*! (*Memoization*)
- With this at hand, we note that we only need to compute the subproblems $\mathrm{weight}(j)$, for $0 \leq j \leq n$.
  Moreover, if have solutions to $\mathrm{weight}(k)$ for all $k < j$, we can obtain $\mathrm{weight}(j)$ by the recursion:

$$\mathrm{weight}(j) = \max\{w_j + \mathrm{weight}(p(j)), \mathrm{weight}(j-1)\}$$

  which takes $O(1)$ time to compute, when we have the values $\mathrm{weight}(j-1)$ and $\mathrm{weight}(p(j))$
- Thus, running time is $O(n \log n)$, as we spent $O(n \log n)$ to sort the intervals and then it takes $O(n)$ time to compute all values of $\mathrm{weight}(j)$, for $0 \leq j \leq n$.

Principles of Dynamic Programming

- Reduce our problem to a simple recurrence relation
- **Important:** this recurrence relation should only have *small number of subproblems* appearing in its recursion tree!
- *Memoization*: compute from *bottom-up*, storing answers to subproblems in memory.
- Return final answer!

- Dynamic Programming
  - General Paradigm
  - Simple example: Fibonacci
- Weighted Interval Scheduling
  - Solution with Dynamic Programming
  - Principles of Dynamic Programming
- Subset-Sum & Knapsack
  - Subset-Sum
  - Knapsack
- Acknowledgements

Subset-Sum

- **Input:** n non-negative weights, denoted $w_1, \ldots, w_n$, and a bound W
- **Output:** subset $S \subseteq [n]$ such that
  1. $\sum_{i \in S} w_i \leq W$
  2. $\sum_{i \in S} w_i \geq \sum_{i \in T} w_i$ (for all T satisfying 1)
- **Model:** Word RAM
- special case of 0-1 knapsack (values equal weights)

Subset-Sum

- **Input:** n non-negative weights, denoted $w_1, \ldots, w_n$, and a bound W
- **Output:** subset $S \subseteq [n]$ such that
  1. $\sum_{i \in S} w_i \leq W$
  2. $\sum_{i \in S} w_i \geq \sum_{i \in T} w_i$ (for all T satisfying 1)
- If we try the same approach as in previous problem, we run into trouble

Subproblems of $([n], W)$ are:

  1. $([n-1], W)$ (if we don't take weight w_n)
  2. $([n-1], W - w_n)$ (if we do take w_n)

Subset-Sum

- **Input:** n non-negative weights, denoted $w_1, \ldots, w_n$, and a bound W
- **Output:** subset $S \subseteq [n]$ such that
  1. $\sum_{i \in S} w_i \leq W$
  2. $\sum_{i \in S} w_i \geq \sum_{i \in T} w_i$ (for all T satisfying 1)
- If we try the same approach as in previous problem, we run into trouble

  Subproblems of $([n], W)$ are:
  1. $([n-1], W)$ (if we don't take weight w_n)
  2. $([n-1], W - w_n)$ (if we do take w_n)
- Account for all values that the total weight W can take!

Subset-Sum

- **Input:** n non-negative weights, denoted $w_1, \dots, w_n$, and a bound W
- **Output:** subset $S \subseteq [n]$ such that
  1. $\sum_{i \in S} w_i \leq W$
  2. $\sum_{i \in S} w_i \geq \sum_{i \in T} w_i$ (for all T satisfying 1)
- If we try the same approach as in previous problem, we run into trouble

  Subproblems of $([n], W)$ are:
  1. $([n-1], W)$ (if we don't take weight w_n)
  2. $([n-1], W - w_n)$ (if we do take w_n)
- Account for all values that the total weight W can take!
- Subproblems: all pairs of the form $([j], \omega)$, where $j \in [n]$ and $0 \leq \omega \leq W$

Subset-Sum

- **Input:** n non-negative weights, denoted $w_1, \ldots, w_n$, and a bound W
- **Output:** subset $S \subseteq [n]$ such that
  1. $\sum_{i \in S} w_i \leq W$
  2. $\sum_{i \in S} w_i \geq \sum_{i \in T} w_i$ (for all T satisfying 1)
- If we try the same approach as in previous problem, we run into trouble

  Subproblems of $([n], W)$ are:
  1. $([n-1], W)$ (if we don't take weight w_n)
  2. $([n-1], W - w_n)$ (if we do take w_n)
- Account for all values that the total weight W can take!
- Subproblems: all pairs of the form $([j], \omega)$, where $j \in [n]$ and $0 \leq \omega \leq W$
- So DP will build up a table of all values of $\mathrm{weight}([j], \omega)$ and use recurrence:

$$\mathrm{weight}([j], \omega) = \max\{\mathrm{weight}([j-1], \omega), w_j + \mathrm{weight}([j-1], \omega - w_j)\}$$

Analysis of DP algorithm

- Number of subproblems: $O(n \cdot W)$
- Time to compute solution to supproblem, given table of "smaller" subproblems: $O(1)$
- Total running time: $O(n \cdot W)$
- Correctness follows from recursion

Analysis of DP algorithm

- Number of subproblems: $O(n \cdot W)$
- Time to compute solution to supproblem, given table of "smaller" subproblems: $O(1)$
- Total running time: $O(n \cdot W)$
- Correctness follows from recursion
- This algorithm is called *pseudo-polynomial*, since its running time is polynomial in n and W (the largest integer involved in defining the problem)

Pseudo-polynomial good when low numbers, bad when big numbers.

0-1 Knapsack

- **Input:** n items, each with a prescribed value and weight, given by $(v_1, w_1), \ldots, (v_n, w_n)$, as well as a maximum load W
- **Output:** a subset of the items $S \subseteq [n]$ such that:
  1. $\sum_{k \in S} w_i \leq W$ (respect max load)
  2. $\sum_{k \in S} v_i \geq \sum_{i \in T} v_i$ for any other set T that respects max load
- **Model:** Word RAM

0-1 Knapsack

- **Input:** n items, each with a prescribed value and weight, given by $(v_1, w_1), \ldots, (v_n, w_n)$, as well as a maximum load W
- **Output:** a subset of the items $S \subseteq [n]$ such that:
  1. $\sum_{k \in S} w_i \leq W$ (respect max load)
  2. $\sum_{k \in S} v_i \geq \sum_{i \in T} v_i$ for any other set T that respects max load
- Same solution as Subset Sum: the recurrence now becomes

$$\text{value}([j], \omega) = \max\{\text{value}([j-1], \omega), v_j + \text{value}([j-1], \omega - w_j)\}$$

Acknowledgement

- Based on prof. Lau's lecture 11 notes

  https://cs.uwaterloo.ca/~lapchi/cs341/notes/L11.pdf
- Based on [Kleinberg Tardos 2006, Chapter 6]

References I

Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford. (2009)
Introduction to Algorithms, third edition.
MIT Press

Kleinberg, Jon and Tardos, Eva (2006)
Algorithm Design.
Addison Wesley