Dynamic Programming

Elements of Dynamic Programming

Spring 2022

Contents

- Overview DP
- Elements of DP

Overview

The key idea of dynamic programming:

Avoid recomputing the same thing again!

Dynamic Programming

- Careful brute force“
- guessing + recursion + memoization
- Dividing into reasonable #subproblems whose solutions relate
- time=#subproblems x time/subproblem
- (treating recursive calls as O(1)
- (usually mainly guessing)

- count each subproblem only once; after first time, costs O(1) via memoization

5 Easy Steps to Dynamic Programming

1. define subproblems — count # subproblems
2. guess (part of solution) — count # choices
3. relate subproblem solutions — compute time/subproblem
4. recurse + memoize — time = time/subproblem · # subproblems
   OR build DP table bottom-up
   check subproblems acyclic/topological order
5. solve original problem: = a subproblem
   OR by combining subproblem solutions — $\Longrightarrow$ extra time

Examples:	Fibonacci
subprobs:	F_k for $1 \le k \le n$
# subprobs:	n
guess:	nothing
# choices:	1
recurrence:	$F_k = F_{k-1} + F_{k-2}$
time/subpr:	$\Theta(1)$
topo. order:	for $k = 1, \ldots, n$
total time:	$\Theta(n)$
orig. prob.:	F_n
extra time:	$\Theta(1)$

Elements of dynamic programming

When should we look for a dynamic programming solution to a problem?

Two key ingredients?

- Optimal sub-structure
- Overlapping sub-problems

Elements of dynamic programming

Optimal sub-structure

A problem exhibits optimal sub-structure, if an optimal solution to a problem contains within it optimal solutions to sub-problems.

an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.

Optimal substructure(contd.)

Example of optimal substructure:

$A_1 A_2 \ldots A_n$

$n=4$

$$m[1,4]=\min\begin{cases} m[1,1]+m[2,4]+p_0p_1p_5 \\ m[1,2]+m[3,4]+p_0p_2p_5 \\ m[1,3]+m[4,4]+p_0p_3p_5 \end{cases}$$

Space of Sub-Problem

- To characterize the space of subproblems, a good rule of thumb says to try to **keep the space as simple as possible** and then expand it as necessary.

Dynamic Programming

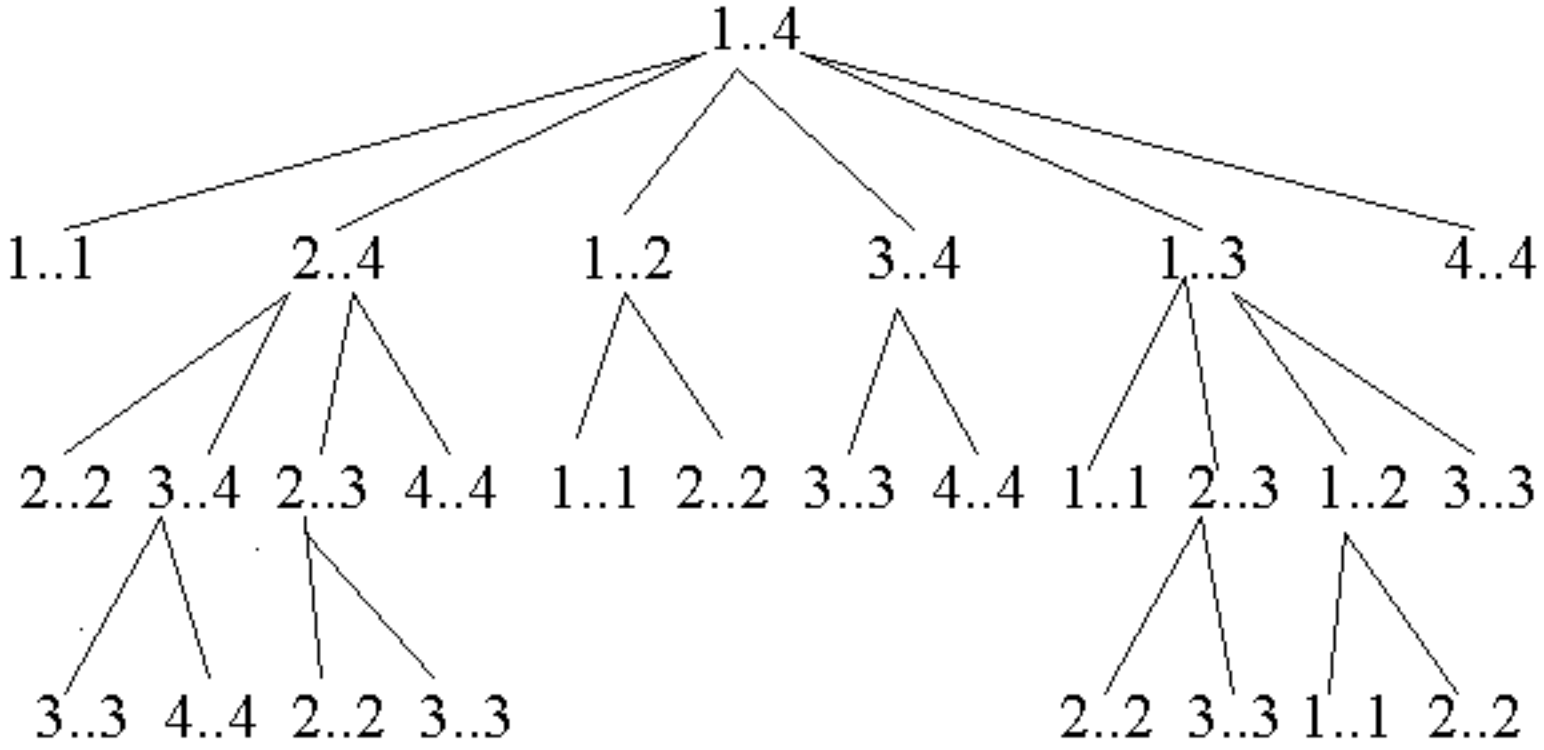
- Dynamic programming often uses optimal substructure in a bottom-up fashion.
- That is, we first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem.
- Finding an optimal solution to the problem entails making a choice among subproblems as to which we will use in solving the problem.
- The cost of the problem solution is usually the subproblem costs plus a cost that is directly attributable to the choice itself.

In matrix-chain multiplication, we determined optimal parenthesizations of subchains of $A_i A_{i+1} \cdots A_j$, and then we chose the matrix A_k at which to split the product. The cost attributable to the choice itself is the term $p_{i-1} p_k p_j$.

A Recursive Solution

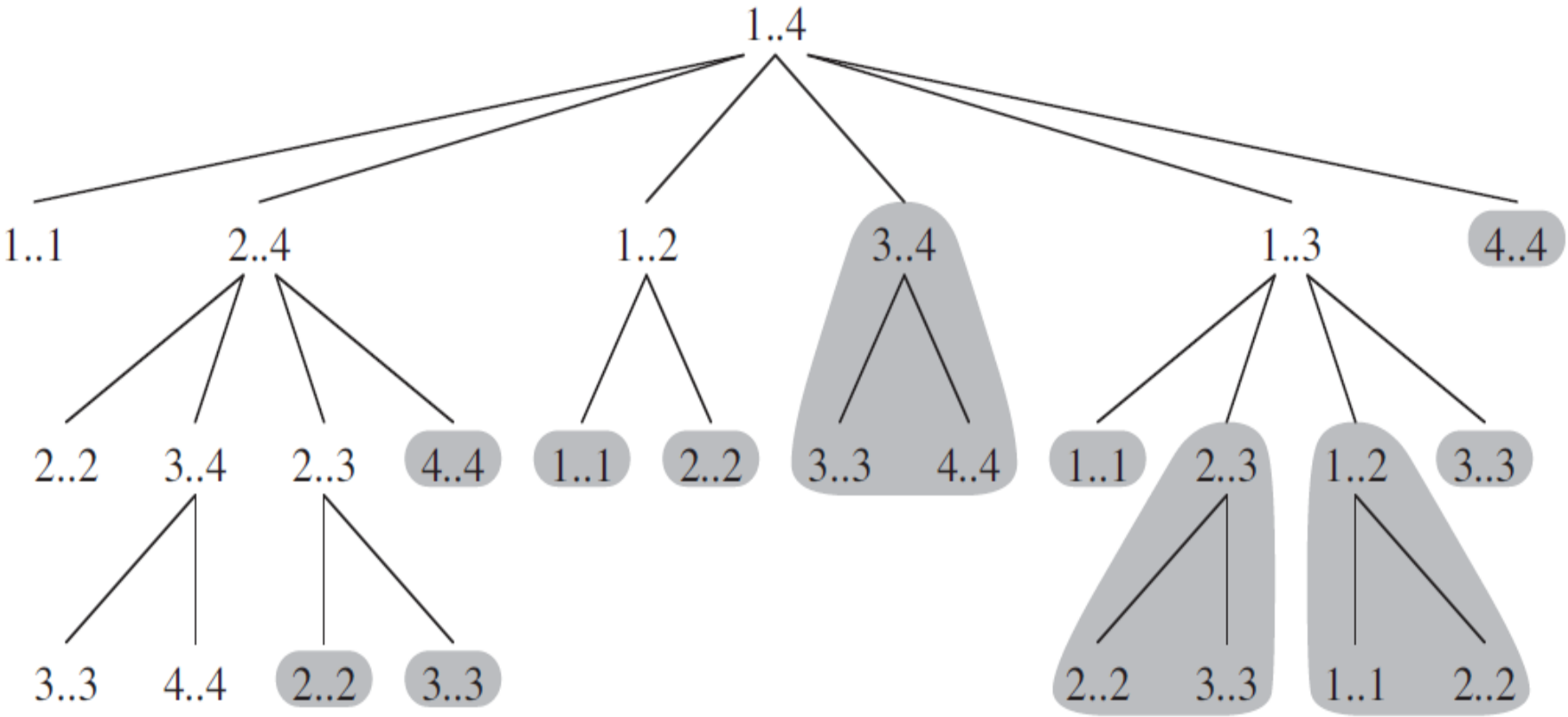
- Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i \ldots j}$.
- If $i = j$, then the sequence consists of just one matrix and no multiplications are involved. Otherwise, we have to split the sequence at some position.
- Let us assume that the optimal parenthesization splits the product $A_i A_{i+1} \ldots A_j$ between A_k and A_{k+1}, where $i \leq k < j$.
- Then, m[i, j] is equal to the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together.
- Since computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications we obtain

Overlapping subproblems (contd.)



recursion tree for computation of RECURSIVE-MATRIX-CHAIN(p1,4)

Overlapping Sub-Problems



Overlapping Sub problems

When a recursive algorithm revisits the same problem, over and over again, we say that the optimization problem has overlapping sub problems.

Typically the total number of distinct sub problems, is polynomial in the input size.

Divide & Conquer approach is suitable when brand new problems are generated at each step of the recursion.

Overlapping Sub problems

Two ways of solving the same problem:

Top-down approach

This being based on recursion i.e. recursively solving the larger problem by breaking it down

Bottom-Up approach

This being based on dynamic programming. Starting with the smallest sub-problems, solving them and using their solutions

Overlapping Sub problems

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2        return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5        q = RECURSIVE-MATRIX-CHAIN(p, i, k)
               + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
               + p_{i−1} p_k p_j
6        if q < m[i, j]
7              m[i, j] = q
8  return m[i, j]
```

Overlapping Sub problems

Running time of recursive procedure, where T(n) is the running time of computing m[1,n] :

$$\begin{cases} T(1) \geq 1 \\ T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad for \;\; n > 1 \end{cases}$$

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

- **We can prove that $T(n) = \Omega(2^n)$ using substitution method.**

Overlapping Sub problems

$$T(1) \geq 1 = 2^0$$

$$T(n) \geq 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^{i} + n$$

$$= 2(2^{n-1} - 1) + n = (2^n - 2) + n \geq 2^{n-1}$$

Overlapping Sub problems

Top-down approach Vs Bottom-Up approach

Bottom-Up approach

More efficient because it takes advantage of the overlapping sub-problem property. There are only $\theta(n^2)$ different sub-problems. Each problem solved exactly once.

Top-down approach

Must repeatedly resolve each sub-problem, each time it reappears in the recursion.

Memoization

- A variation of Dynamic Programming and is TOP-DOWN
- Memoize the natural, but inefficient, recursive algorithm.
- Maintains an entry in the table for the solution to each sub-problem.
- Offers efficiency while maintaining top-down strategy.

Memoization

- A special character stored at each empty solution space.
- When the sub-problem is first encountered, it's solution is computed and stored.
- Each subsequent time the sub-problem is encountered, its solution is looked up.
- Time to search the sub-problem solutions?

Memoization

- This approach presupposes that the set of all possible sub-problem parameters is known and that the relation between table position and sub-problem is established.
- Another approach is to memoize by using hashing with the sub-problem parameters as keys.

MEMOIZATION

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

Conclusion

- We can solve the matrix-chain multiplication problem by either a **top-down, memoized dynamic-programming algorithm** or **a bottom-up dynamic programming algorithm** in $O(n^3)$ time.
- Both methods take advantage of the **overlapping-subproblems property.**
- There are only $\Theta(n^2)$ distinct subproblems in total, and either of these methods computes the solution to each subproblem only once.
- **Without memoization**, the natural **recursive algorithm runs in exponential time**, since solved subproblems are repeatedly solved.

Conclusion

- In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table.
- Moreover, for some problems we can exploit the regular pattern of table accesses in the dynamic programming algorithm to reduce time or space requirements even further.
- Alternatively, if some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Reference

- **Introduction to Algorithms**
  - Thomas H. Cormen
  - Chapter # 15