csce750 — Analysis of Algorithms
Fall 2019 — Lecture Notes: Dynamic Programming

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with some supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

1 Introduction

CLRS 15

Dynamic programming is a technique for algorithm design based on decomposing a problem into *overlapping subproblems*.

Key idea: Express the solution using a recurrence that relates the solution to the solutions to subproblems.

Signs that your problem is a good fit for dynamic programming:

- **Optimal substructure:** An optimal solution to a problem contains optimal solutions to subproblems.
- **Overlapping subproblems:** The subproblems that generate the optimal substructure are dependent on each other.

Dynamic programming is recursion without repetition.

2 How to design a dynamic programming algorithm

Creating dynamic programming algorithms is not an automatable process, but there is a general pattern.

1. Identify a **family of subproblems**, in which the final answer is a special case.
2. **Write a recurrence** showing how the solutions to those subproblems are related to each other.
3. Find **base cases** for the recurrence.
4. Form the algorithm.
   (a) Fill in a **table of recurrence values** in an order that obeys the dependencies.
   (b) Extract the **final solution** from the table.

(CLRS 359 describes the same pattern in a slightly different way.)

3 Example: Rod cutting

Section 15.1 of the textbook describes a classic and useful example of dynamic programming called the **rod cutting** problem, including "top down" versus "bottom up" versions of the solution.

For the sake of time, we will not discuss rod cutting in detail in class, but you should read Section 15.1 carefully.

4 Example: Matrix multiplication ordering

Suppose we have n matrices $A_1, \ldots, A_n$ to multiply:

$$X = A_1 \cdot A_2 \cdot \cdots \cdot A_n$$

Because matrix multiplication is associative, we can choose the order in which we perform the multiplications.

$$A(BC) = (AB)C$$

The simple algorithm for multiplying an $m \times n$ matrix with an $n \times p$ matrix takes mnp scalar multiplications.

Which order should we use to compute X as quickly as possible?

5 Order matters

Suppose A is 5×20, B is 20×2, and C is 2×10.

- $(AB)C$:
  - $A \cdot B$ takes $5 \cdot 20 \cdot 2 = 200$ multiplications.
  - $(AB) \cdot C$ takes $5 \cdot 2 \cdot 10 = 100$ multiplications.
- $A(BC)$:
  - $B \cdot C$ takes $20 \cdot 2 \cdot 10 = 400$ multiplications.
  - $A \cdot (BC)$ takes $5 \cdot 20 \cdot 10 = 1000$ multiplications.

Therefore: The *order* in which we perform the multiplications can make a big difference.

Problem ('optimal parenthezation'): Given a sequence of matrix sizes

$$p_0, p_1, \ldots, p_n,$$

in which matrix i has size $p_{i-1} \times p_i$, find an ordering for the matrix multiplications that minimizes the total number of scalar multiplications needed.

6 Matrix chain multiplication: Family of subproblems

Idea 1: Given indices i and j, define

$$A_{i..j} = A_i \cdot A_{i+1} \cdot \cdots \cdot A_j.$$

The final answer we seek is $A_{1..n}$.

Idea 2: Let $m[i, j]$ denote the smallest number of multiplications needed to compute $A_{i..j}$.

7 Matrix chain multiplication: Recurrence

Suppose we want to compute $A_{i..j}$. Consider the **last** multiplication in the optimal sequence:

$$A_{i..j} = A_{i..k} \cdot A_{k+1..j}$$

Idea 3: Both $A_{i..k}$ and $A_{k+1..j}$ must both be optimal sequences for their respective subproblems.

Idea 4: If we know i, j, and k, the total number of matrix multiplications to compute $A_{i..j}$ is:

$$m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

8 Matrix chain multiplication: Recurrence (2)

Question: What value should we choose for k?

Answer: Try them all, and choose the best one.

$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_k p_j)$$

9 Matrix chain multiplication: Base case

If $i = j$, then it's easy because there is only one matrix:

$$m[i,j] = 0 \qquad \text{if } i = j$$

10 Matrix chain multiplication: Algorithm

```
MATRIXCHAINMULT(p[0,...,n])
  m = new n × n array
  for i = 1,...,n do
    m[i,i] = 0
  end for
  for l = 2,...,n do
    for i = 1,...,n − l + 1 do
      j = i + l − 1
      m[i,j] = ∞
      for k = i,...,j − 1 do
        m[i,j] = min(m[i,j], m[i,k] + m[k+1,j] + p[i−1]p[k]p[j])
      end for
    end for
  end for
  return m
```

11 Matrix chain multiplication: Solution extraction

To extract the solution, we need to know, for each $m[i,j]$, *which value of* k led to the smallest total.

$$s[i,j] = \operatorname*{argmin}_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_k p_j)$$

We can use this information to extract the complete solution recursively:

```
PRINTOPTIMALPARENTHEZATION(s, i, j)
  if i = j then
    print "A"_i
  else
    print "("
    PRINTOPTIMALPARENTHEZATION(s, i, s[i, j])
    PRINTOPTIMALPARENTHEZATION(s, s[i, j] + 1, j)
    print ")"
  end if
```

12 Matrix chain multiplication: Algorithm (with extraction data)

```
MATRIXCHAINMULT(p[0, ..., n])
  m = new n × n array
  s = new n × n array
  for i = 1, ..., n do
    m[i, i] = 0
  end for
  for l = 2, ..., n do
    for i = 1, ..., n − l + 1 do
      j = i + l − 1
      m[i, j] = ∞
      for k = i, ..., j − 1 do
        q = m[i, k] + m[k + 1, j] + p[i − 1]p[k]p[j]
        if q < m[i, j] then
          m[i, j] = q
          s[i, j] = k
        end if
      end for
    end for
  end for
  return m and s
```

13 Greedy algorithms

CLRS 16

We won't cover greedy algorithms directly (though we will see a few examples later). However, Chapter 16 has some interesting insight into when a greedy approach might work correctly:

> *[B]eneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.*