Dynamic Programming

Data Structures and Algorithms (094224)

Yuval Emek

Winter 2022/23

A general theme in algorithm design

- Focus on optimization problems
  - Looking for a feasible solution that minimizes/maximizes some objective function
  - E.g., shortest (s, t)-path, minimum spanning tree
- Common to many (optimization) problems:
  an (optimal) solution to the problem can be constructed from (optimal) solutions to smaller subproblems
- Property exploited by recursive (divide and conquer) algorithms
  - A top-down approach
  - Not clear in advance which subproblems will be solved
  - The same subproblem may be solved many times!
- Dynamic programming:
  - A bottom-up approach
  - Each subproblem is solved exactly once
  - Solutions are stored in a lookup table
    - Accessed in the process of solving larger subproblems
  - Trading space for time
    - Don't solve same subproblem many times, but solution has to be stored

The rod cutting problem

- Given a rod of length n
- Customers are willing to pay $p(i)$ for a (sub)rod of length $1 \leq i \leq n$
- Task: cut the rod into $1 \leq k \leq n$ subrods of lengths $\ell_1, \ldots, \ell_k \in \mathbb{Z}_{>0}$ whose total length is $\sum_{i=1}^{k} \ell_i = n$
- Objective: maximize (total) payoff $\sum_{i=1}^{k} p(\ell_i)$
- Brute force algorithm: try all possibilities to cut the rod and pick the one that maximizes the payoff
- How many ways are there to cut the rod?
  - 2^{n-1} if we consider the cut locations
  - $\approx \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$ (partition number) if we only care about subrod lengths
- A different approach is needed

Relating the problem to its subproblems

- Define $r(k)$ = optimal payoff that can be made from a length k rod
- Key observation:

$$r(n) = \begin{cases} 0, & n = 0 \\ \max_{1 \leq i \leq n} \{p(i) + r(n-i)\}, & n > 0 \end{cases}$$

  - Make the first cut at length i and cut the remaining rod optimally
- If we have already computed $r(0), r(1), \ldots, r(n-1)$, then computing $r(n)$ is straightforward
  - Theme: optimal solution to the problem from optimal solutions to its subproblems
    - A.k.a. *optimal substructure*
- A recursive equation for $r(n)$ although dynamic programming algorithms are not recursive!

Pseudocode

Compute the optimal payoff obtainable from a length n rod under prices p

`Cut_Rod`(n, p)

new array $r[0 \ldots n]$
$r[0] = 0$
for $j = 1, \ldots, n$ **do**
    $q = -\infty$
    for $i = 1, \ldots, j$ **do**
        $q = \max\{q, p(i) + r[j - i]\}$
    $r[j] = q$
return $r[n]$

Remarks

- Pseudocode essentially implements the recursive equation for $r(n)$
  - Almost a "template"
  - Correctness follows directly from the correctness of the recursive equation for $r(n)$
  - Typical for dynamic programming algorithms
- Constructing the optimal cutting scheme (rather than just its value):
  - Store the iteration i in which the maximum is realized (line 6) for each j
    - Enables tracking back the optimal solution
  - Generally: keep track of the subproblem(s) that realize the optimal value for each table entry

Run-time and space analysis

- Run-time:
  - n iterations of the outer for loop (lines 3–7)
  - $O(n)$ iterations of the inner for loop (lines 5–6)
  - $O(1)$ time in each inner loop iteration
  - $O(n^2)$ time in total
- Space:
  - $O(n)$

Recalling matrix multiplication

- Matrices $A \in \mathbb{R}^{p \times q}$, $B \in \mathbb{R}^{q \times r}$
- The product $C = AB \in \mathbb{R}^{p \times r}$ is defined so that

$$C(i,j) = \sum_{k=1}^{q} A(i,k) \cdot B(k,j)$$

- Run-time of the standard matrix multiplication algorithm is proportional to *pqr*
  - #scalar multiplying operations (smo)

Associative matrix multiplication

- Input: matrix dimensions $p_0, p_1, \ldots, p_n \in \mathbb{Z}_{>0}$
- Help compute the product $A_1 \cdots A_n$, where $A_i \in \mathbb{R}^{p_{i-1} \times p_i}$
- Matrix multiplication is associative:
  order of multiplying operations doesn't affect the product
  - $(A_1 A_2) A_3 = A_1 (A_2 A_3)$
- Order of multiplying operations does affect the run-time
  - $\mathrm{smo}\left((A_1 A_2) A_3\right) = p_0 p_1 p_2 + p_0 p_2 p_3$
  - $\mathrm{smo}\left(A_1 (A_2 A_3)\right) = p_1 p_2 p_3 + p_0 p_1 p_3$
- Goal: determine the optimal order
  - Notice: we don't compute the actual product $A_1 \cdots A_n$
  - Don't even need to know the matrices $A_1, \ldots, A_n$
    - Only the dimensions $p_0, p_1, \ldots, p_n$ matter

Full parenthesization

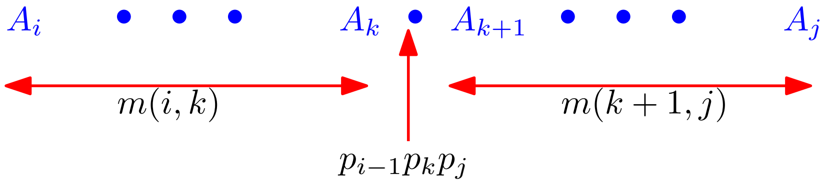
- Matrix product expression is *fully parenthesized* if it is either
  - a single matrix; or
  - the product of two fully parenthesized matrix product expressions, surrounded by parentheses
- Order of multiplying operations is (fully) determined iff expression is (fully) parenthesized
- Goal: determine the full parenthesization that minimizes smo
- How many possibilities to fully parenthesize the matrix product?
  - Full parenthesization is encoded by an n-leaf full binary tree
    - $M(x)$ = fully parenthesized matrix expression associated with node x
    - If x_i is the ith leftmost leaf, then $M(x_i) = A_i$
    - If x is an internal node with left child x_ℓ and right child x_r, then $M(x) = (M(x_\ell) \cdot M(x_r))$
  - #full binary trees with n leaves $= \frac{1}{n}\binom{2n-2}{n-1} = \Omega\left(4^n / n^{3/2}\right)$
    - The $(n-1)$st Catalan number
    - Way too many possibilities for a brute force approach

Relating the problem to its subproblems

- Define $m(i,j) = \mathrm{smo}$ of an optimal full parenthesization for $A_i \cdots A_j$
  - $1 \le i \le j \le n$
- Key observation:

$$m(i,j) = \begin{cases} 0, & i = j \\ \min_{i \le k < j} \left\{ m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \right\}, & i < j \end{cases}$$

  - $i = j$: there is nothing to multiply
  - $i < j$: break $A_i \cdots A_j$ into $A_i \cdots A_k$ and $A_{k+1} \cdots A_j$ for some $i \le k < j$



Pseudocode

Compute the smo of an optimal full parenthesization for $A_1 \cdots A_n$

`Matrix_Chain_Order`(p)

$n = p.size - 1$
new table $m[1 \ldots n, 1 \ldots n]$
for $i = 1, \ldots, n$ **do**
  $m[i, i] = 0$
for $\ell = 2, \ldots, n$ **do** ▷ subexpression length
  for $i = 1, \ldots, n - \ell + 1$ **do**
    $j = i + \ell - 1$
    $m[i, j] = \infty$
    for $k = i, \ldots, j - 1$ **do**
      $m[i, j] = \min\{m[i, j], m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\}$
return $m[1, n]$

Run-time and space analysis

- Run-time:
  - Initialization takes $O(n)$ time
  - 3 nested loops, each with $O(n)$ iterations
  - $O(1)$ time for each inner-most iteration
  - $O(n^3)$ time in total
- Space: $O(n^2)$