CSC006P1M: Design and Analysis of Algorithms
Lecture 09 *(Dynamic Programming)*

Sumit Kumar Pandey

September 07, 2022

Dynamic Programming

Those who cannot remember the past are condemned to repeat it.

-Dynamic Programming

Dynamic Programming

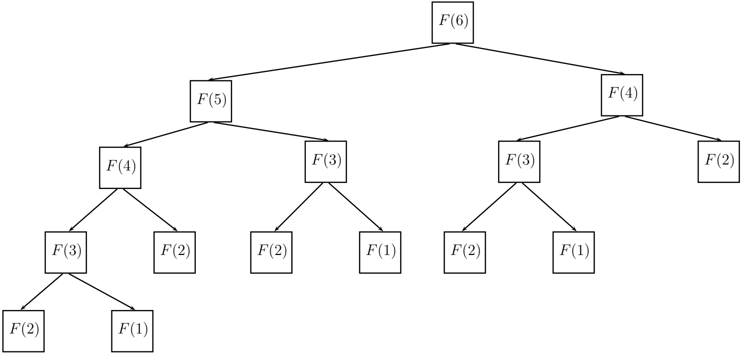
- Dynamic Programming, like the Divide-and-Conquer method, solves problems by combining the solutions to sub-problems.
- Dynamic Programming is applied when the sub-problems overlap - that is, when sub-problems share sub-sub-problems.
- A dynamic-programming algorithm solves each sub-problem just once and saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each sub-problem.

Fibonacci Series.
$F(n) = F(n-1) + F(n-2),\ F(1) = 1, F(2) = 1.$

Algorithm $F(n)$
Input: n
Output: Fib
begin
    if $n <= 2$, then return 1;
    else $Fib := F(n-1) + F(n-2)$;
    return Fib;
end

Dynamic Programming

Let $n = 6$.



Fibonacci Series.
$F(n) = F(n-1) + F(n-2),\ F(1) = 1, F(2) = 1.$

Algorithm $F(n)$
Input: n
Output: Fib
begin
    $Fib[1] := 1, Fib[2] := 1;$
    for $i := 3$ to n do
        $Fib[i] := Fib[i-1] + Fib[i-2];$
end

Running Time $T(n) = \Theta(n)$; Space required $S(n) = \Theta(n)$.

Majority (not all) of the Dynamic Programming can be categorised into two types:

1. Optimization Problems: Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with optimal value (minimum or maximum).
2. Combinatorial Problems: Such problems deal with the number of ways to do something, or the probability of some event happening.

Dynamic Programming

A sequence of four steps is generally followed while developing a dynamic-programming algorithm.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

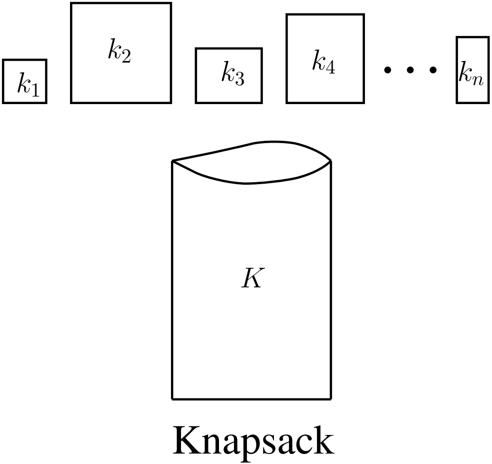
A Variant of a Knapsack Problem

Given a positive integer K and n items of different sizes such that the i^{th} item has a positive integer size k_i, find a subset of the items whose sizes sum to exactly K, or determine no such subset exists.

A Variant of a Knapsack Problem : Decisional

Given a positive integer K and n items of different sizes such that the i^{th} item has a positive integer size k_i, determine whether a subset of the items whose sizes sum to exactly K, exists or not?

Knapsack Problem



$S \overset{?}{\subseteq} \{1, 2, \cdots, n\}$ such that $\sum_{s \in S} k_s = K$

Knapsack Problem

We denote the problem by $P(n, K)$, such that n denotes the number of items and K denotes the size of the knapsack. Thus, $P(i, k)$ denotes the problem with the **first** i **items** and a knapsack of size k.

How to Solve?

Use Induction.

Induction Hypothesis

We know how to solve $P(n-1, K)$.

Knapsack Problem

Induction Hypothesis

We know how to solve $P(n-1, K)$.

- The base case, $n = 1$, is easy; there is a solution only if the single element has size K.
- If a solution to $P(n-1, K)$ exists, then we are done; we will not use the n^{th} item.
- But what if no solution to $P(n-1, K)$ exists?
    - We must include n^{th} item.
    - In this case, the rest of the items must fit into a smaller knapsack of size $K - k_n$.

The problem $P(n, K)$ is reduced into two smaller sub-problems - $P(n-1, K)$ and $P(n-1, K-k_n)$.

Knapsack Problem

$$S$$

$$n \notin S \qquad\qquad n \in S$$

$$P(n-1, K) \qquad\qquad P(n-1, K-k_n)$$

Induction Hypothesis (Strong)

We know how to solve $P(n-1, k)$ for all $0 \leq k \leq K$.

Knapsack Problem

- Let "A" denotes an array of "n" integers.
- "P" be a two dimensional array of size "$n \times K$". The two dimensional array "P" is in-fact an array of structures.
- Each structure contains two attributes - "exist" and "belong".
- If "$P[i, k]$.exist := true", it means that there exists a **subset of the first "i" items** such that their sum equals "k". Else, no subset of first "i" items exists so that their sum equals "k".
- "$P[i, k]$.belong := true" means that the subset contains the item "i". Else the subset does not contain "i".

If $P[i-1, k]$.exist := true $P[i, k]$.exist := true; $P[i, k]$.belong := false;	If $k - A[i] \geq 0$ and $P[i-1, k-A[i]]$.exist := true $P[i, k]$.exist := true; $P[i, k]$.belong := true;

Knapsack Problem

```
Algorithm Knapsack(A, K);
Input: A (an array of size n storing the size of the items)
    and K (the size of the knapsack).
Output: P (a two dimensional array)
begin
    P[0, 0].exist := true;
    for k := 1 to K do
        P[0, k].exist := false;
    {there is no need to initialize P[i, 0] for i ≥ 1, because it will}
    {be computed from P[0, 0].}
    for i := 1 to n do
        for k := 0 to K do
            P[i, k].exist := false; { the default value }
            if P[i − 1, k].exist then
                P[i, k].exist := true;
                P[i, k].belong := false;
            else if k − A[i] ≥ 0 then
                if P[i − 1, k − A[i]].exist then
                    P[i, k].exist := true;
                    P[i.k].belong := true;

end
```

Knapsack Problem

Example: Let $n = 4, K = 9, k_1 = 2, k_2 = 3, k_3 = 5, k_4 = 6$

	0	1	2	3	4	5	6	7	8	9
$k_0 = 0$	T	F	F	F	F	F	F	F	F	F
$k_1 = 2$	O	$-$	I	$-$	$-$	$-$	$-$	$-$	$-$	$-$
$k_2 = 3$	O	$-$	O	I	$-$	I	$-$	$-$	$-$	$-$
$k_3 = 5$	O	$-$	O	O	$-$	O	$-$	I	I	$-$
$k_4 = 6$	O	$-$	O	O	$-$	O	I	O	O	I

(exist, belong)

$T \rightarrow (T, *)$; $F \rightarrow (F, *)$; $O \rightarrow (T, F)$; $I \rightarrow (T, T)$; $- \rightarrow (F, F)$

Knapsack Problem

Time Complexity

$T(n, K) = O(nK)$.

The running time is **pseudo-polynomial**.

Pseudo-Polynomial

An algorithm runs in pseudo-polynomial time if its running time is a polynomial in the numeric value of the input but not in the input size.

Dynamic Programming vs Divide-and-Conquer

- Dynamic Programming is similar to Divide-and-Conquer in the sense that it is based on a recursive division of the problem into simpler problems of the same type.
- However, whereas the Divide-and-Conquer utilizes top-down approach, the Dynamic Programming utilizes bottom-up approach.
- Unlike many instances of the Divide-and-Conquer, the Dynamic Programming typically never considers a given sub-problem more than once.

Matrix Chain Multiplication

Matrix Chain Multiplication

We are given a sequence (chain) of matrices $\langle A_1, A_2, \cdots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute,

$$A_1 A_2 \cdots A_n.$$

One solution: $((\cdots((A_1 A_2)A_3)\cdots)A_n)$.
Another solution: $(A_1(\cdots(A_{n-2}(A_{n-1}A_n))\cdots))$.
There could be many more ways.

Matrix Chain Multiplication

Example:

If the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then we can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways:

1. $(A_1(A_2(A_3A_4)))$,
2. $(A_1((A_2A_3)A_4))$,
3. $((A_1A_2)(A_3A_4))$,
4. $((A_1(A_2A_3))A_4)$,
5. $(((A_1A_2)A_3)A_4)$.

Question

Will the cost be same in all cases?

Answer

No.

Matrix Chain Multiplication

Let A be a $p \times q$ and B be a $q \times r$ matrix. Suppose $C = A \times B$ be a $p \times r$ matrix. The total cost (the number of multiplications) to compute C is pqr.

Example: Let A_1 be a 10×100, A_2 be a 100×5 and A_3 be a 5×50 matrix. There could be two ways to compute $A_1A_2A_3$.

1. $((A_1A_2)A_3)$
   - $B = A_1A_2 \rightarrow 10 \cdot 100 \cdot 5 = 5000$.
   - $C = BA_3 \rightarrow 10 \cdot 5 \cdot 50 = 2500$.
   - The total cost to compute $((A_1A_2)A_3)$ is $5000 + 2500 = 7500$.
2. $(A_1(A_2A_3))$
   - $B = A_2A_3 \rightarrow 100 \cdot 5 \cdot 50 = 25000$.
   - $C = A_1B \rightarrow 10 \cdot 100 \cdot 50 = 50000$.
   - The total cost to compute $((A_1A_2)A_3)$ is $50000 + 25000 = 75000$.

Matrix Chain Multiplication

Matrix Chain Multiplication Problem

Given a chain $\langle A_1, A_2, \cdots, A_n \rangle$ of n matrices, where for $i = 1, 2, \cdots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, *the matrix chain multiplication problem* is to fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of multiplications.

Solution

Use Dynamic Programming.

Four step sequence for dynamic programming.

1. Characterise the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Four step sequence for dynamic programming.

1. Characterise the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

The structure of an optimal solution.

- Let $A_{i \dots j}$ denotes the product $A_i A_{i+1} \cdots A_j$.
- If $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, it must be split between A_k and A_{k+1} for some integer k in the range $i \leq k \leq j$.
- Which means, first compute $A_{i \dots k}$ as well as $A_{k+1 \dots j}$. And then compute $A_{i \dots j}$ by multiplying $A_{i \dots k}$ and $A_{k+1 \dots j}$.
- The optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i \cdots A_k$. (Why?)
- The optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_{k+1} \cdots A_j$.

Example:

1. $(A_1(A_2(A_3A_4)))$, $k = 1$.
2. $(A_1((A_2A_3)A_4))$, $k = 1$.
3. $((A_1A_2)(A_3A_4))$, $k = 2$.
4. $((A_1(A_2A_3))A_4)$, $k = 3$.
5. $(((A_1A_2)A_3)A_4)$, $k = 3$.

Four step sequence for dynamic programming.

1. Characterise the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Matrix Chain Multiplication

A recursive solution.

- Let $m[i,j]$ be the minimum number of multiplications needed to compute the matrix $A_{i\ldots j}$.
- Therefore, $m[1,n]$ denotes the lowest number of multiplications needed to compute $A_{1\ldots n}$.
- $m[i,j]$ can be recursively defined as

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j. \end{cases}$$

  where each matrix A_i is $p_{i-1} \times p_i$.
- $m[i,j]$ contains the optimal solutions to subproblems, but do not keep information of where to split, i.e. k.
- Let $s[i,j] = k$ so that $m[i,j] = \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\}$

Four step sequence for dynamic programming.

1. Characterise the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Matrix Chain Multiplication

Computing the value of an optimal solution.

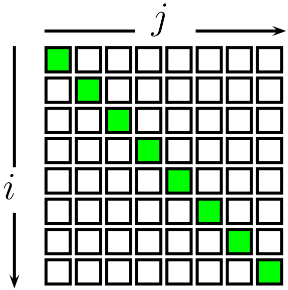
Algorithm Matrix-Chain-Order(p);
Input: $p = \langle p_0, p_1, p_2, \cdots, p_n \rangle$ (an array of size $n+1$ which contains the dimension of matrices $A_1, A_2, \cdots A_n$. The dimension of the matrix A_i is $p_{i-1} \times p_i$).
Output: m and s (both two dimensional arrays). begin

```
n := p.length − 1;
Create arrays m[1···n, 1···n] and s[1···n−1, 2···n].
for i := 1 to n do m[i, i] = 0;
for l := 2 to n do {l is the chain length}
    for i := 1 to n − l + 1 do
        j := i + l − 1;
        m[i, j] := ∞;
        for k := i to 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

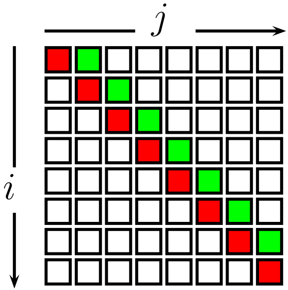
$$T(n) = O(n^3);\ S(n) = \Theta(n^2).$$

Matrix Chain Multiplication

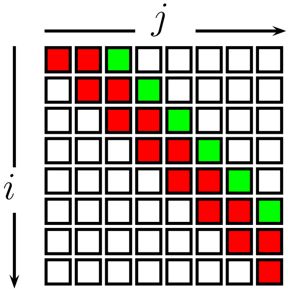


$$m[i,i] = 0$$

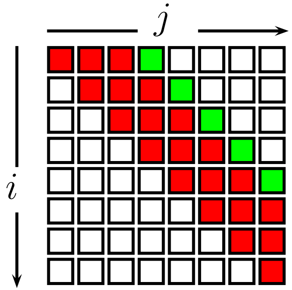
Matrix Chain Multiplication



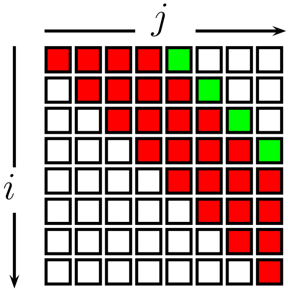
Matrix Chain Multiplication



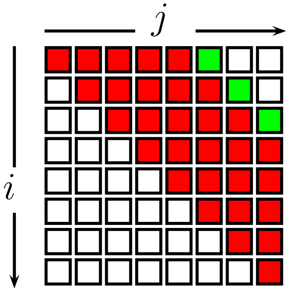
Matrix Chain Multiplication



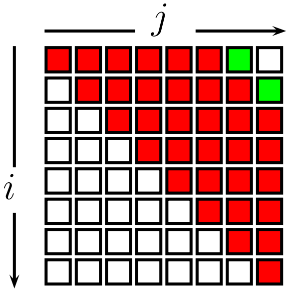
Matrix Chain Multiplication



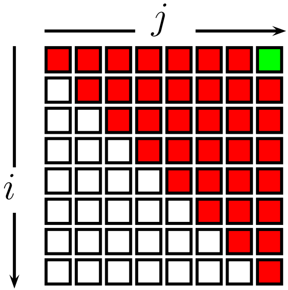
Matrix Chain Multiplication

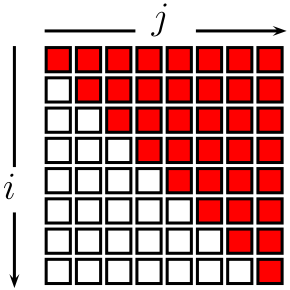


Matrix Chain Multiplication



Matrix Chain Multiplication




j
i

Four step sequence for dynamic programming.

1. Characterise the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Constructing an optimal solution.

Algorithm PrintOptimalParens(s,i,j);
Input: s (a two dimensional array), i and j (two positive integers which indicate $A_i A_{i+1} \cdots A_j$).
Output:
begin
  if $i = j$ then print "A"$_i$;
  else print "(";
    PrintOptimalParens(s,i,$s[i,j]$);
    PrintOptimalParens(s,$s[i,j]+1$,j);
    print ")";
end

Thank You