CS 340 - Analysis of Algorithms
Dynamic Programming

Knapsack
CMM

Announcements

Quiz Thursday (Divide and Conquer)

HW6 released - due Nov 10th

Outline

- **Review:** Weighted Interval Scheduling
- Knapsack Problem
- Chain Matrix Multiplication (CMM)

Dynamic Programming

- Smart recursion - *without repetition*
- Stores the solutions of intermediate subproblems, usually in tables (arrays)
- Optimization problems that can be solved by a greedy algorithm are VERY rare
- Your first instinct should be DP, not greedy

DP Design

Break down the problem by **expressing the optimal solution in terms of optimal solutions to sub-parts**

- Subproblems may overlap
- The number of subproblems must be reasonably small

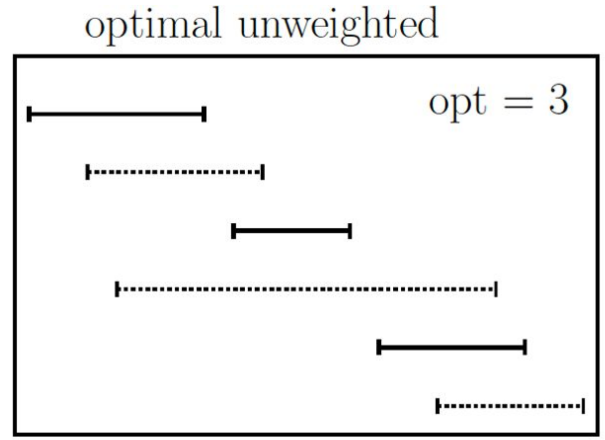
To do this, determine the recursive formulation

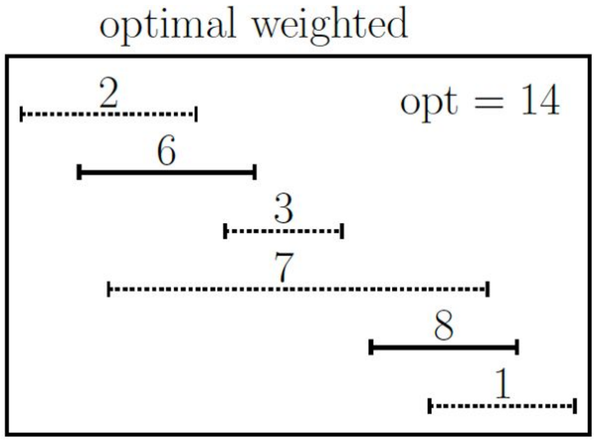
- First describe the precise function you want to evaluate in English
- Then, give a formal recursive definition of the function

Needs an evaluation order and a measure of “optimal”

Review: Weighted Interval Scheduling

- A more general version of interval scheduling
- Each interval also has a weight w_i
- Find a set of compatible intervals that have maximized total weights



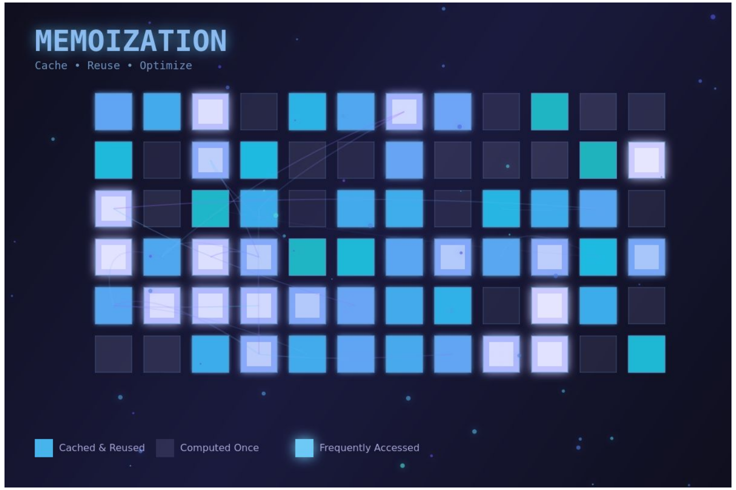


Review Weighted Interval Scheduling

- Given a set R of n activities with start-finish times $[s_i, f_i], 1 \leq i \leq n$

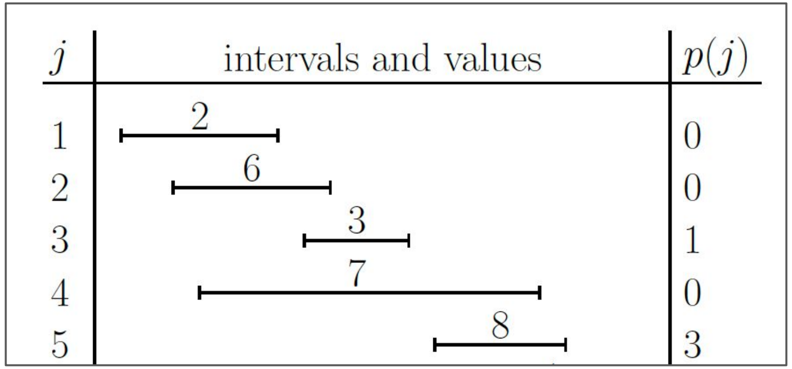
- **Goal**: select a subset of compatible intervals S as to maximize the sum of weights:

$$\sum_{i \in S} w_i$$



Review: Weighted Interval Scheduling

```
compute-opt(){
  M[0] = 0
  for (j = 1 to n) {
    if (M[j-1] > w[j]+M[p[j]]) {
      M[j] = M[j-1];
    }
    else {
      M[j] = w[j]+M[p[j]];
    }
  }
}
```



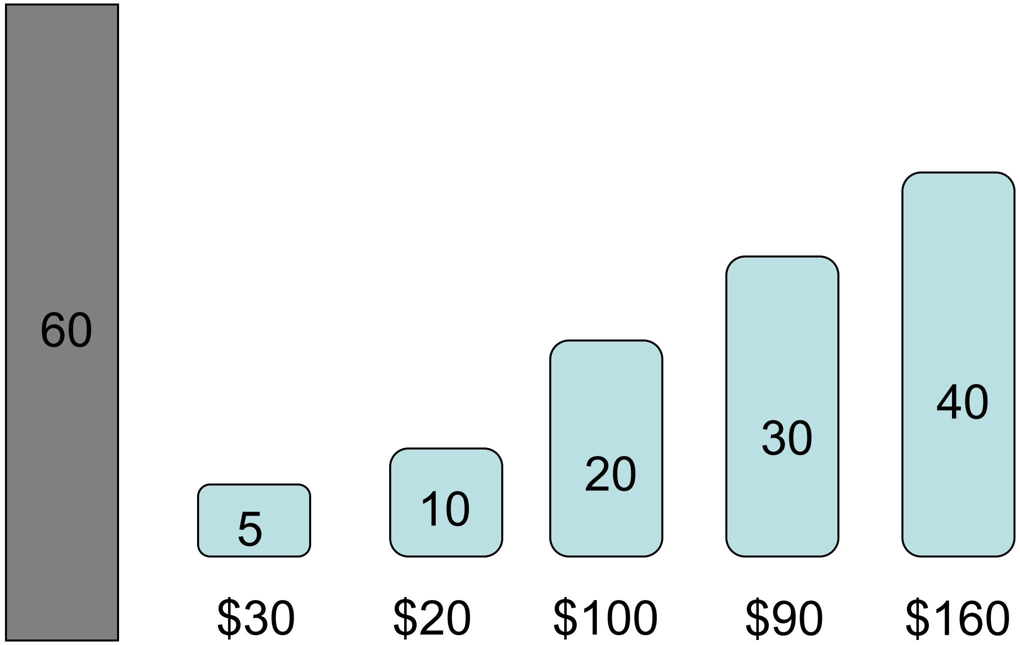
j	intervals and values	$p(j)$
1	2	0
2	6	0
3	3	1
4	7	0
5	8	3

```
j = n
solution = ∅
while (j > 0) {
    if (M[j] != M[j-1]) {
        solution.add(j)
        j = p[j]
    } else {
        j = j-1
    }
}
```

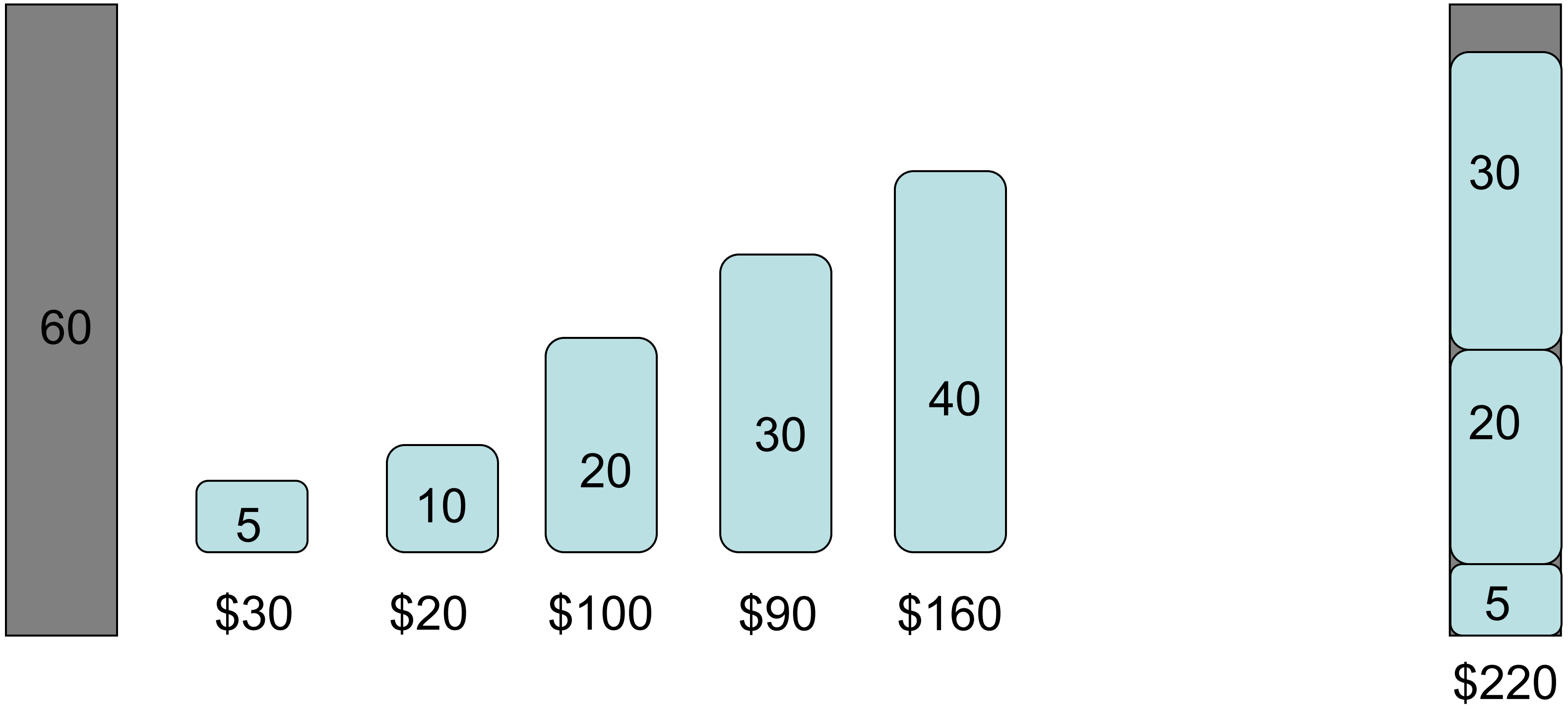
0-1 Knapsack Problem

- Given a weight limit W, and n items with associated values $\langle v_1, v_2, \dots, v_n \rangle$ and weights $\langle w_1, w_2, \dots, w_n \rangle$,
- determine the subset T of items that maximizes $\sum_{i \in T} v_i$, subject to $\sum_{i \in T} w_i \leq W$
- Assume all weights are integers

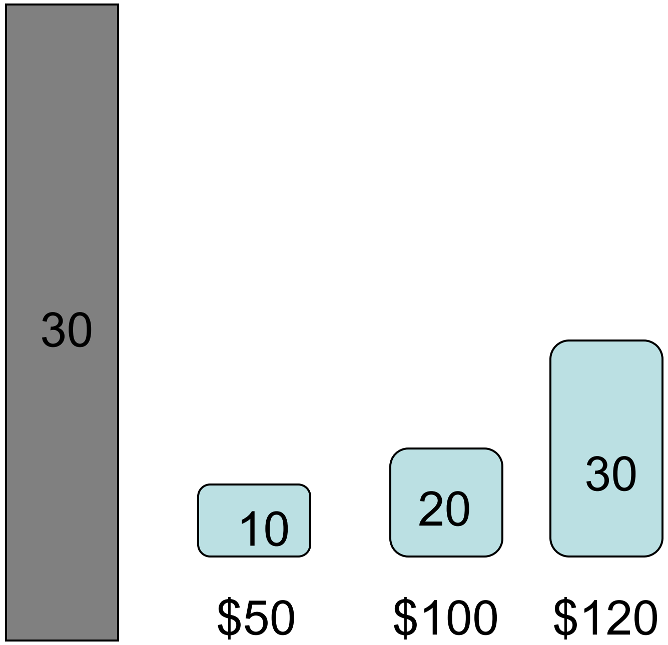
Example



Greedy Solution?



Example 2

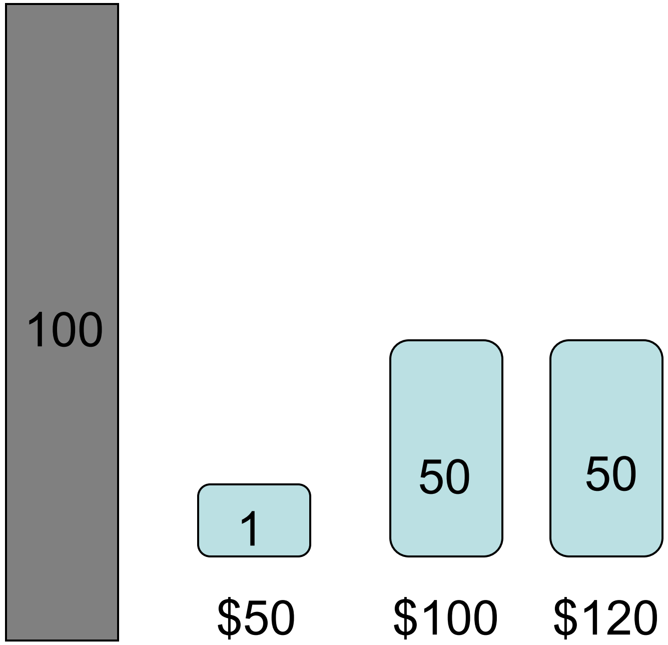


Optimal: {1, 2} for a value of $150

Greedy algorithm: select highest value item first

selects 3 which prevents it from selecting any other items.

Example 3



Optimal: {2, 3} for a value of $220

Greedy algorithm: select smallest weight first

selects 1 which prevents us from selecting any others

Greedy fails... Let’s try brute force

****express the optimal solution in terms of optimal solutions to sub-parts****

The optimal solution for `n` items and capacity `W` is the maximum of either **including** the `n`th item (adding its value and reducing capacity) or **excluding** it (using the best solution for `n-1` items).

$$
\begin{aligned}
OPT(n, W) = MAX(& \\
&V[n] + OPT(n-1, W-w_n), \\
&OPT(n-1, W) \\
)&
\end{aligned}
$$

Runtime?

Our memoization is now 2D

- Fill $V[0..n], 0 \leq i \leq n$
- $V[i]$ stores the max value of any subset of objects $\{1, 2, \ldots, i\}$
  - item i is not selected
  - item i is selected

- Need more parameters/subproblems

Take it or Leave it

- Fill $V[0..n, 0..W], 0 \leq i \leq n, 0 \leq j \leq W$
- $V[i, j]$ stores the max value of any subset of objects $\{1, 2, \dots, i\}$ that can fit into weight j
- $V[0, j] = 0, 0 \leq j \leq W$
- $V[i, j] =$
$$\begin{cases} V[i-1, j], w_i > j \\ \max(V[i-1, j], v_i + V[i-1, j-w_i]), w_i \leq j \end{cases}$$

Bottom-up DP

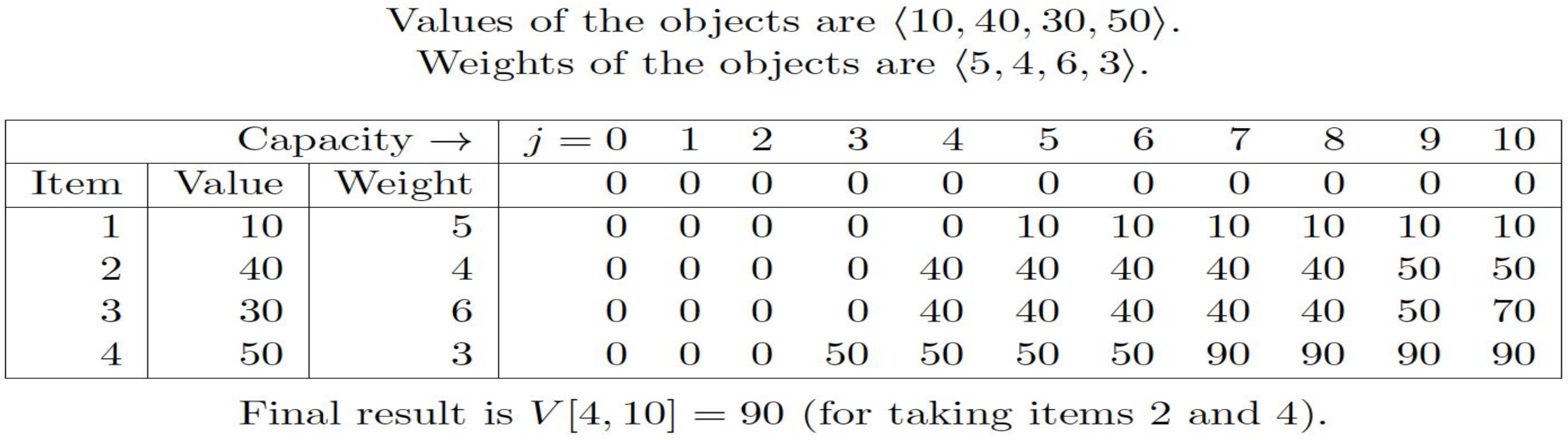
```
// Initialize the table: If no items or weight capacity is 0, max value is 0
for i from 0 to n:
    for j from 0 to W:
        if i == 0 or j == 0:
            V[i, j] = 0  // Base case: No items or no capacity
        else if w[i] > j:
            V[i, j] = V[i-1, j]  // Item too heavy, cannot include
        else:
            V[i, j] = max(V[i-1, j], v[i] + V[i-1, j-w[i]])
            // Max of excluding or including the item

// The answer is stored in value[n, W]
print(value[n, W])
```

W= 10

Item	Value	Weight
1	10	5
2	40	4
3	30	6
4	50	3

Example



Values of the objects are $\langle 10, 40, 30, 50 \rangle$.
Weights of the objects are $\langle 5, 4, 6, 3 \rangle$.

		Capacity →	$j = 0$	1	2	3	4	5	6	7	8	9	10
Item	Value	Weight	0	0	0	0	0	0	0	0	0	0	0
1	10	5	0	0	0	0	0	10	10	10	10	10	10
2	40	4	0	0	0	0	40	40	40	40	40	50	50
3	30	6	0	0	0	0	40	40	40	40	40	50	70
4	50	3	0	0	0	50	50	50	50	90	90	90	90

Final result is $V[4, 10] = 90$ (for taking items 2 and 4).

Analysis

- Time
  - There are $n \cdot W$ entries – $O(nW)$
  - not polynomial, pseudo-polynomial
- Correctness by (strong) induction
  - base: $V[0, j] = 0, V[i, 0] = 0$
  - IH: $V[i', j']$ is optimal $\forall\, i' + j' < i + j$
  - inductive: prove $V[i, j]$ is optimal

Optimality of V[i, j]

$$V[i,j] = \begin{cases} V[i-1,j] & \text{if } w_i > j \quad \text{(item } i \text{ does not fit in the knapsack)} \\ \max(V[i-1,j], V[i-1,j-w_i] + v_i) & \text{if } w_i \leq j \quad \text{(choose max between including or not including item } i\text{).} \end{cases}$$

Case 1: Weight of item i exceeds remaining capacity.
we cannot take item i so the optimal solution remains V[i-1, j]

Case 2: We can either include or exclude i.
Depends on two earlier locations (V[i-1, j] and V[i-1, j-w_i])

The inductive hypothesis covers both these locations in the matrix. We can assume they are both optimal.

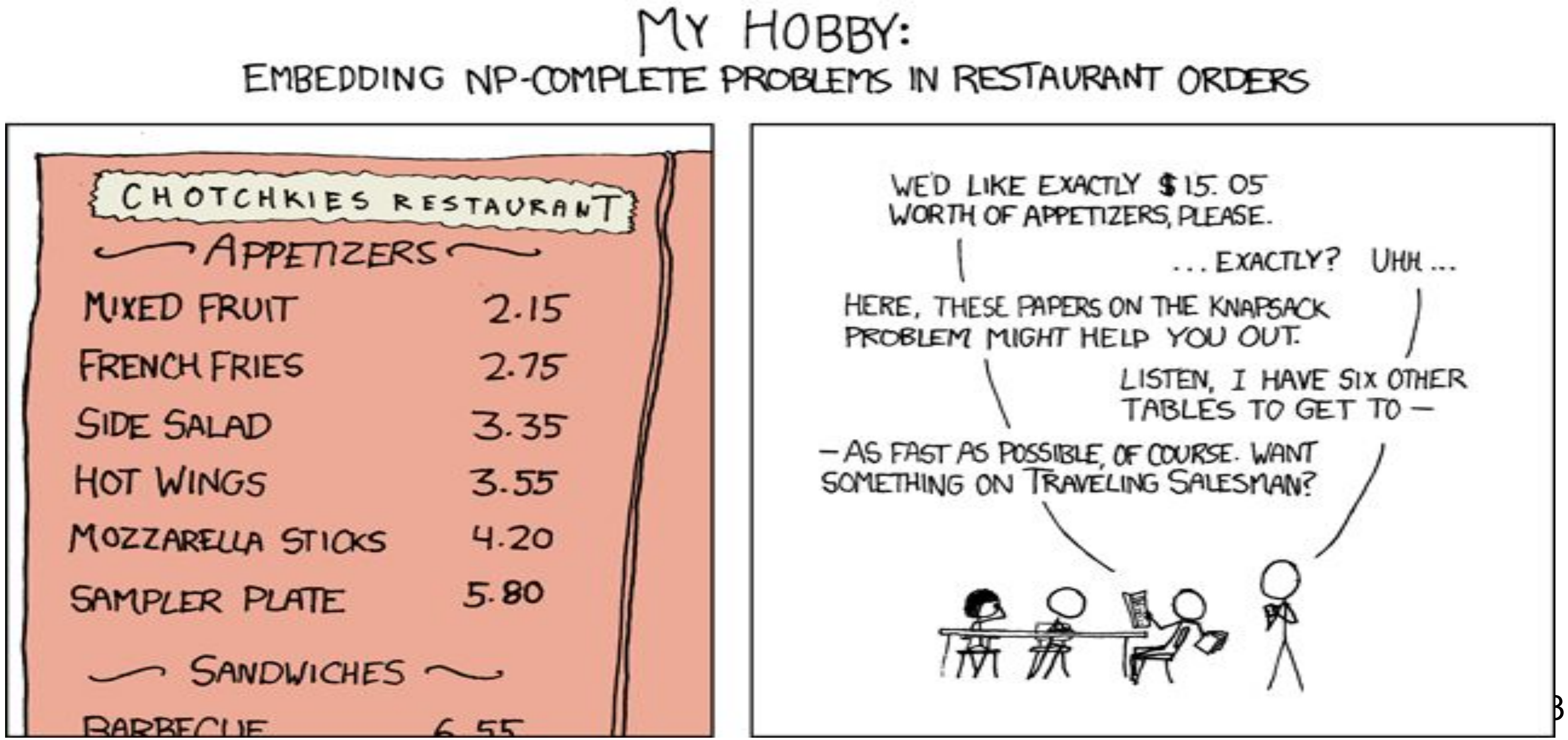
Computing a solution

```
selected_items = []
i = n
j = W

while i > 0 and j > 0:
    # If the value comes from including item i
    if V[i, j] != V[i-1, j]:
        selected_items.append(i)  # Item i was included
        j = j - w[i]  # Reduce capacity by item's weight
        i = i - 1
    else:
        # Value comes from not including item i
        i = i - 1
```

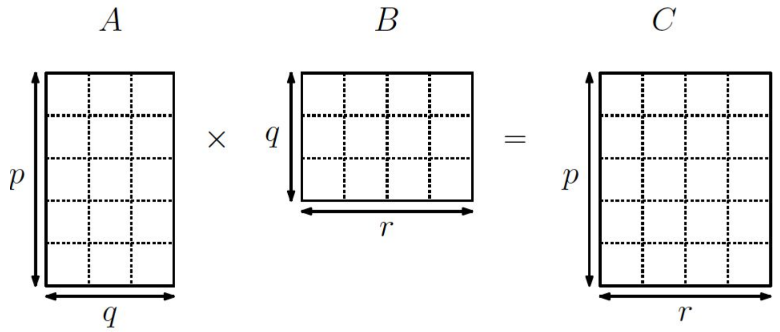
CMM Problem

XKCD 287


MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
BARBECUE 6.55
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO —
— AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

Matrix Multiplication

- Associative but not commutative
- Parenthesize but not rearrange order
- $C[i,j] = \sum_{k=1}^{q} A[i,k] \cdot B[k,j]$
- $O(pqr)$



Matrix Multiplication

If M1 and M2 are of sizes (axb)(bxc), how many total multiplication operations occur in M1*M2?

Each entry is computed by multiplying an entire row from M1 by a column from M2

How many operations occur to compute *each entry?*
b!

The resulting matrix will be of size (axc), so we do the above for a*c entries.

Total number of multiplications is a*b*c

Chain Matrix Multiplication

Multiply a series of matrices

C = A1 * A2 * .... An

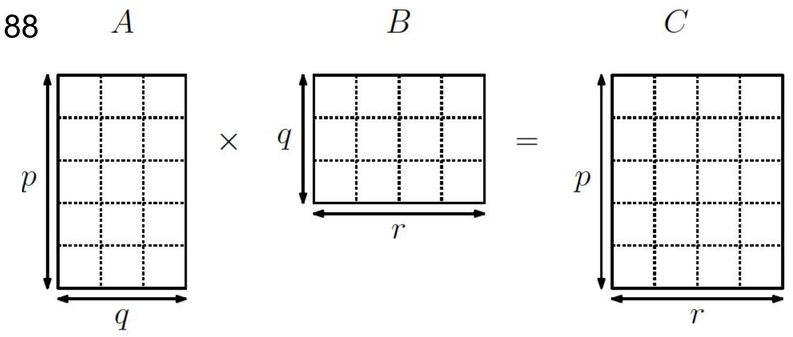
Suppose A1 = (5x4), A2 = (4x6), A3 = (6x2)

What is the total number of multiplications?

((A1A2)A3) = ((5x4)(4x6))A3 = 120 + (5x6)(6x2) = 180

(A1(A2A3) = A1 * (4x6)(6x2) = (5x4)(4x2) + 48 = 88

Matrix multiplication is associative



CMM Statement

- Given a sequence of matrices $A_1, \dots, A_n$ and dimensions $p_0, \dots, p_n$, where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication that minimizes the total number of operations

Example

n = 4

A1 X A2 x A3 x A4

P0 = 5

P1 = 4

P2 = 6

P3 = 2

P4 = 7

- Given a sequence of matrices $A_1, \dots, A_n$ and dimensions $p_0, \dots, p_n$, where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication that minimizes the total number of operations

1. What are the sizes of A1...A4?
2. What are our options for multiplication order?

Example

1. (A1A2)×(A3A4)	1. K = 2
2. A1x(A2A3A4)	2. K = 1
3. (A1A2A3)xA4	3. K = 3

A **split index** refers to the specific position in the sequence of matrices where the multiplication is divided into two smaller subproblems during matrix chain multiplication.

DP Formulation

Break down the problem by **expressing the optimal solution in terms of optimal solutions to sub-parts**

- Let $A_{i..j}$ denote the result of multiplying matrices i through j
  - $A_{i..j}$ has dimension $p_{i-1} \times p_j$
- $A_{1..n} = A_{1..k} \cdot A_{k+1..n}$

Recursive Formulation

1. We want to compute the minimum number of scalar multiplications to multiply matrices from A_i to A_j
   a. We can "split the sequence" at different locations with parenthesis

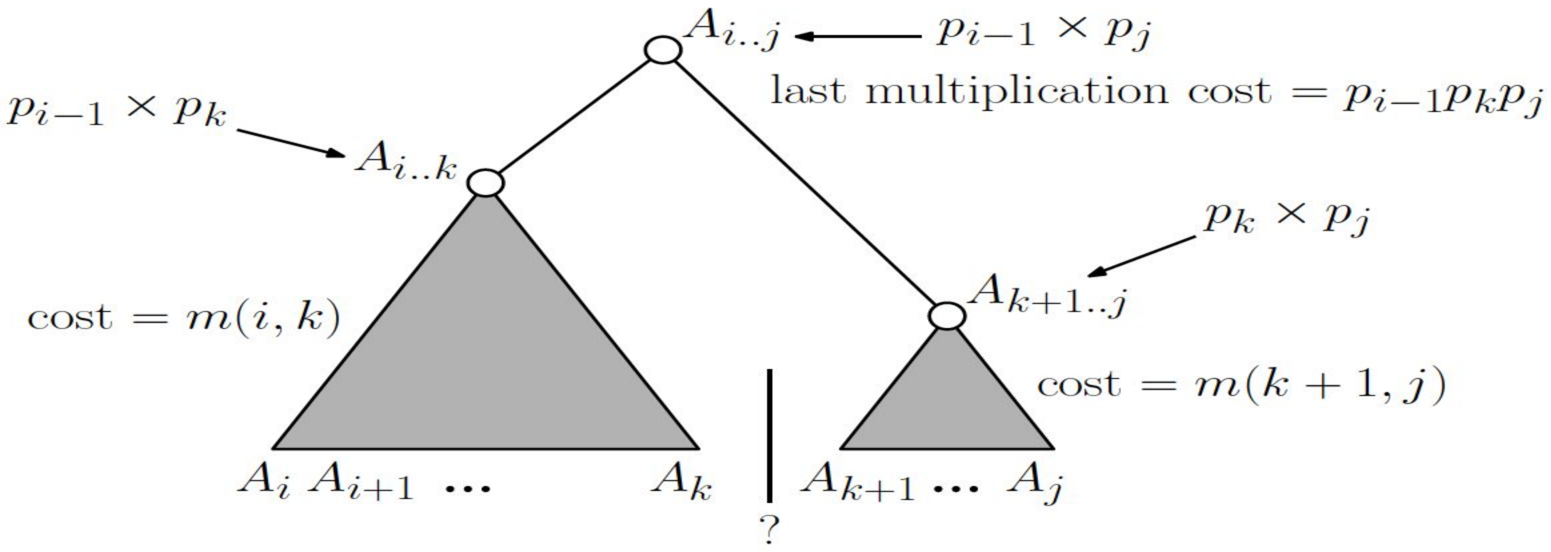
2. Let m(i,j) = minimum number of scalar multiplications needed to compute $A_i ... A_j$

3. $m(i,j) = m(i,k) + m(k+1, j) + (p_{i-1} * p_k * p_j)$

Recursive Formulation

- Let $m(i,j)$ denote the minimum number of operations needed to compute $A_{i..j}$
- $i = j$: $m(i,i) = 0$
- $i < j$: $A_{i..k} \cdot A_{k+1..j} = A_{i..j}, i \leq k < j$

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


$A_{i..j}$
$p_{i-1} \times p_j$
last multiplication cost $= p_{i-1}p_kp_j$
$p_{i-1} \times p_k$
$A_{i..k}$
$p_k \times p_j$
$A_{k+1..j}$
cost $= m(i,k)$
cost $= m(k+1,j)$
A_i A_{i+1} ... A_k
?
A_{k+1} ... A_j

For HW

- Go through the concrete example by hand!
- Complete write-up
  - pseudo code
  - time analysis
  - correctness proof
- Provide bottom-up implementation details
  - imperative (non-recursive) pseudo code
  - must be able to recover multiplication order (i.e. not just min number of multiplications)

Summary

- DP: Break down the problem by expressing the optimal solution in terms of optimal solutions to sub-parts
- More efficient than brute force: removes duplicate calculations
- Knapsack requires 2D matrix