CS4102 Algorithms

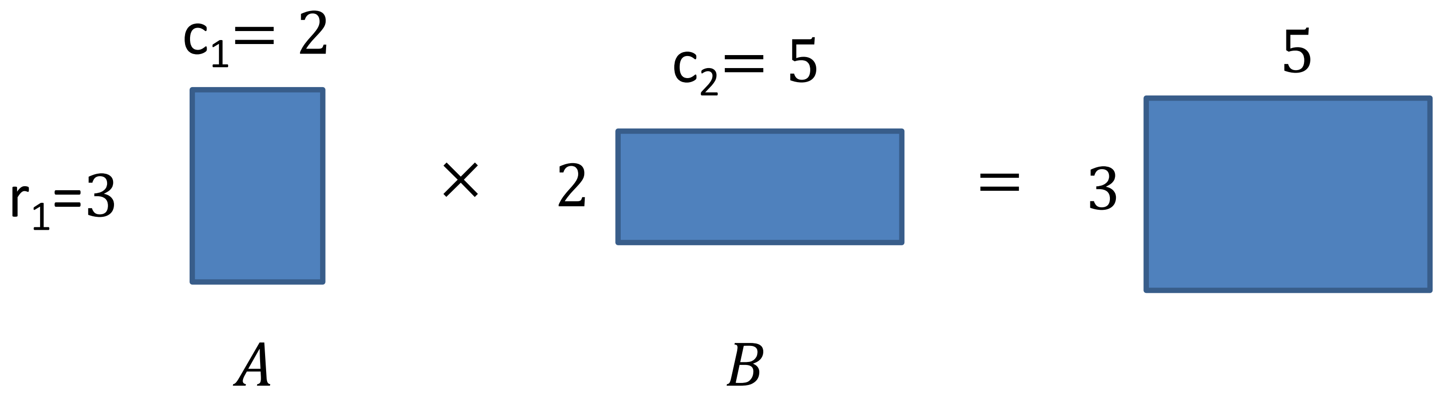
Spring 2021 – Floryan and Horton

Module 3:
Dynamic Programming

Greedy Algorithms

Motivating Example

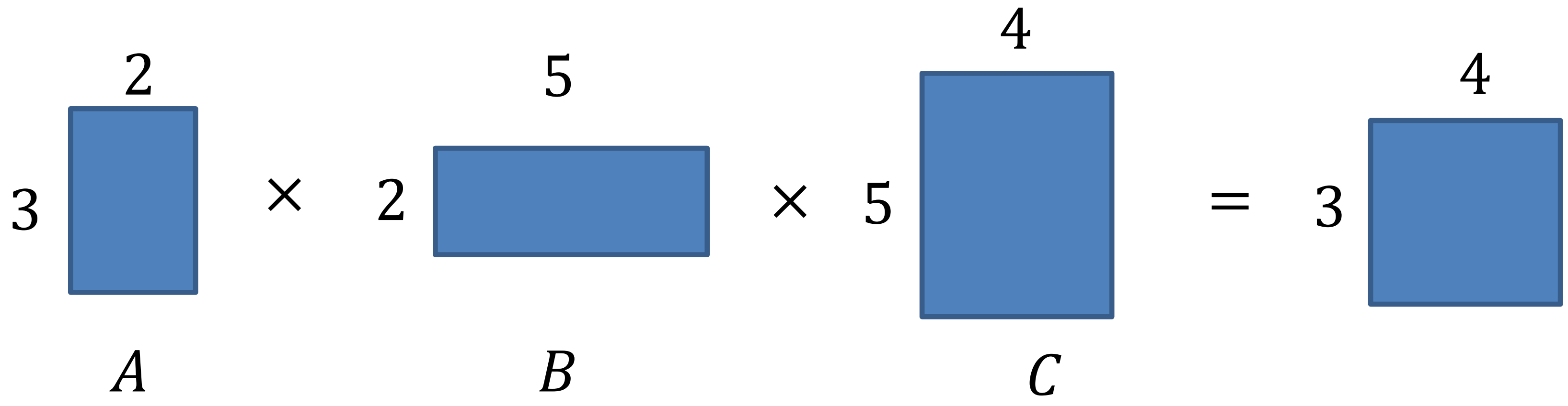
How many scalar multiplications are required to multiply matrices A and B in this example?



- $r_1 \cdot c_2$ elements in the result that we need to compute
- c_1 scalar multiplications per element in result
- Total cost: $r_1 \cdot c_1 \cdot c_2$
- So the answer is… $(3 \cdot 2 \cdot 5) = 30$

Trickier Question

What's the smallest number of scalar multiplications required to calculate the matrix product ABC in this example?



- For a pair of matrices, remember it's $r_1 \cdot c_1 \cdot c_2$
- Calculate this cost for multiplying one pair of matrices
- You need to multiply that result with the 3rd matrix, too, so there's a cost for that
- Total cost is the sum of these two costs
- So the answer is… $(3 \cdot 2 \cdot 5) + (3 \cdot 5 \cdot 4) = 90$

Nope! The answer is 64.
Think about how this might be!

CLRS Readings

- Chapter 15, Dynamic Programming
  - Section 15.1, Log/Rod cutting, optimal substructure property
    - Note: r_i in book is called Cut() or C[] in our slides. We do use their example.
  - Section 15.3, More on elements of DP, including optimal substructure property
- Chapter 16, Greedy Algorithms
  - Intro, page 414
  - Section 16.2, Elements of the Greedy Strategy, Knapsack problem
  - Later Section 16.1, Activity Selection problem

Dynamic Programming and Greedy Approach

- Module 3 is on Dynamic Programming and the Greedy Approach
- This term we're doing something unusual:

  We'll introduce these *together*, not in sequence

  - They have a lot in common
  - Goal: teach you enough about both early enough so you can work on HWs on both topics

Optimization Problems

- Both DP and Greedy solve ***optimization problems:***
  Find the best solution among all ***feasible*** solutions
- An example you know: *Find the shortest path in a weighted graph G from s to v*
  - Form of the solution: a path (and sum of its edge-weights)
- Feasible solutions must meet problem constraints
  - Example: All edges in solution are in graph G and form a simple path from *s* to *v*
- We can get a score for each feasible solution on some criteria:
  We call this the ***objective function***
  - Example: the sum of the edge weights in path
- One (or more) feasible solutions that scores highest (by the objective function) is the ***optimal solution(s)***

Example #1: Knapsack Problems

- Pages 425-427 in textbook
- **Description:** Thief robbing a store finds n items, each with a profit amount p_i and a weight w_i
  - Wants to steal as valuable a load as possible
  - But can only carry total weight C in their knapsack
  - Which items should they take to maximize profit?
- Form of the solution: an x_i value for each item, showing if (or how much) of that item is taken
- Inputs are: C, n, the p_i and w_i values

Two Types of Knapsack Problem

- 0/1 knapsack problem
  - Each item is discrete: must choose all of it or none of it. So each x_i is 0 or 1
  - Greedy approach does not produce optimal solutions
  - But dynamic programming does

- Fractional knapsack problem (AKA continuous knapsack)
  - Can pick up fractions of each item. So each x_i is a value between 0 or 1
  - A greedy algorithm finds the optimal solution

Formal Statement of Fractional Knapsack Problem

- Given n objects and a knapsack of capacity C, where object i has weight w_i and earns profit p_i, find values x_i that maximize the total profit

$$\sum_{i=1}^{n} x_i p_i$$

subject to the constraints

$$\sum_{i=1}^{n} x_i w_i \le C, \quad 0 \le x_i \le 1$$

Greedy Approach

- Let's use a **greedy strategy** to solve the fractional knapsack
  - Build solution by stages, adding one item to partial solution found so far
  - At each stage, make locally optimal choice based on the **greedy choice** (sometimes called the **greedy rule** or the **selection function**)
    - Locally optimal, i.e. best choice given what info available now
  - Irrevocable: a choice can't be un-done
  - Sequence of locally optimal choices leads to globally optimal solution (hopefully)
    - Must prove this for a given problem!
    - Approximation algorithms, heuristic algorithms

A Bit More Terminology

- Problems solvable by both Dynamic Programming and the Greedy approach have the **optimal substructure property:**
  - An optimal solution to a problem contains within it optimal solutions to subproblems
  - This allows us to build a solution one step at a time, because we can solve increasingly smaller problems with confidence
- Dynamic Programming not a good solution for problems that have the **greedy-choice property:**
  - We can assemble a globally-optimal solution for the current by making a locally-optimal choice, without considering results from subproblems

Greedy Approach for Fractional Knapsack?

- Build up a partial solutions:
  - Determine which of the remaining items to add
  - How much can you add (its x_i)
  - Repeat until knapsack is full (or no more items)

- Which item to choose next?
  What's a good **greedy choice** (AKA **greedy selection)**?
- Let's try several obvious options on this example:

n = 3, C = 20

Item	Value	Weight
1	25	18
2	24	15
3	15	10

Possible Greedy Choices for Knapsack

Greedy choice #1: by highest profit value

n = 3, C = 20

Item	Value	Weight
1	25	18
2	24	15
3	15	10

Select item 1 first, then item 2, then item 3. Take as much of each as fits!

1. Item 1 first. Can take all of it, so x_1 is 1. Capacity used is 18 of 20. Profit so far is 25.
2. Item 2 next. Room for only 2 units, so x_2 is 2/15 = 0.133. Capacity used is 20 of 20. Profit so far is 25 + (24 x 0.133) = 28.2.
3. Item 3 would be next, but knapsack full! x_3 is 0. **Total profit is 28.2. x_i = (1, .133, 0)**

Possible Greedy Choices for Knapsack

Greedy choice #2: by lowest weight

n = 3, C = 20

Item	Value	Weight
1	25	18
2	24	15
3	15	10

Select item 3 first, then item 2, then item 1. Take as much of each as fits!

1. Item 3 first. Can take all of it, so x_3 is 1. Capacity used is 10 of 20. Profit so far is 15.
2. Item 2 next. Room for only 10 units, so x_2 is 10/15 = 0.667. Capacity used is 20 of 20. Profit so far is 15 + (24 x 0.667) = 31.
3. Item 1 would be next, but knapsack full! x_1 is 0. **Total profit is 31.0.** **x_i = (0, .667, 1)**

Note it's better than previous greedy choice. Best possible?

Possible Greedy Choices for Knapsack

Greedy choice #3: highest value-to-weight ratio

$n = 3$, $C = 20$

Item	Value	Weight	Ratio
1	25	18	1.4
2	24	15	1.6
3	15	10	1.5

Select item 2 first, then item 3, then item 1. Take as much of each as fits!

1. Item 2 first. Can take all of it, so x_2 is 1. Capacity used is 15 of 20. Profit so far is 24.
2. Item 3 next. Room for only 5 units, so x_1 is 5/10 = 0.5. Capacity used is 20 of 20. Profit so far is 24 + (15 x 0.5) = 31.5.
3. Item 1 would be next, but knapsack full! x_1 is 0. **Total profit is 31.5.** **x_i = (0, 1, 0.5)**

This greedy choice produces optimal solution! Must prove this (but we won't today).

Fractional Knapsack Algorithm

```
FRACTIONAL_KNAPSACK(a, C)
n = a.last
for i = 1 to n
    ratio[i] = a[i].p / a[i].w
sort(a, ratio)
weight = 0
i = 1
while (i ≤ n and weight < C)
    if (weight + a[i].w ≤ C)
        println "select all of object " + a[i].id
        weight = weight + a[i].w
    else
        r = (C – weight) / a[i].w
        println "select " + r + " of object " + a[i].id
        weight = C
    i = i+1
```

Worst-case runtime:
for loop and while loop take $\theta(n)$ time,
sorting takes $\theta(n\lg n)$ time,
so algorithm takes $\theta(n\lg n)$ time

Another Knapsack Example to Try

- Assume for this problem that: $\sum_{i=1}^{n} w_i \leq C$
- Ratios of profit to weight:

  $p_1/w_1 = 5/120 = .0417$
  $p_2/w_2 = 5/150 = .0333$
  $p_3/w_3 = 4/200 = .0200$
  $p_4/w_4 = 8/150 = .0533$
  $p_5/w_5 = 3/140 = .0214$

- What order do we examine items?
- What are the x_i values that result?
- What's the total profit?

Proving a Greedy Algorithm Correct

- For fractional knapsack, we can prove greedy choice of p_i/w_i leads to optimal solution
- In general, given a greedy algorithm, how do approach such a proof?
- Recall we’ve done this for Dijkstra’s SP and Prim’s MST
- We can compare the solution our algorithm finds with an optimal solution
  - Show they’re the same
  - Or, assume they’re not and show a contradiction
  - Remember *exchange argument* for Dijkstra’s or for Prim’s?

0/1 knapsack

Let's try this same greedy solution with the 0/1 version

- New example inputs →

n = 3, C = 4

Item	Value	Weight	Ratio
1	3	1	3
2	5	2	2.5
3	6	3	2

1. Item 1 first. So x_1 is 1.
   Capacity used is 1 of 4. Profit so far is 3.
2. Item 2 next. There's room for it! So x_2 is 1. Capacity used is 3 of 4.
   Profit so far is 3 + 5 = 8.
3. Item 3 would be next, but its weight is 3 and knapsack only has 1 unit left!
   So x_3 is 0. **Total profit is 8.** **x_i = (1, 1, 0)**

But picking items 1 and 3 will fit in knapsack, with total value of 9

- Thus, the greedy solution does not produce an optimal solution to the 0/1 knapsack algorithm
- Greedy choice left unused room, but we can't take a fraction of an item
- The 0/1 knapsack problem doesn't have the *greedy choice property*

Dynamic Programming

Dynamic programming

- Old “bad” name (see Wikipedia or textbook)
- Useful when the solution can be recursively described in terms of solutions to sub-problems (*optimal substructure*)
  - But *greedy choice property* doesn’t hold for the problem
- Algorithm finds solutions to sub-problems and stores them in memory for later use
- More efficient than *brute-force methods* or recursive approaches that solve the same sub-problems over and over again

Optimal Substructure Property

- Definition
  - If S is an optimal solution to a problem, then the components of S are optimal solutions to sub-problems
- Examples:
  - True for coin-changing
  - True for single-source shortest path
  - Not true for longest-simple-path
  - True for knapsack

Dynamic Programming

- Works “bottom-up”
  - Finds solutions to small sub-problems first
  - Stores them
  - Combines them somehow to find a solution to a slightly larger sub-problem
- Comparison to greedy approach
  - Also requires optimal substructure
  - But greedy makes choice first, then solves
  - Greedy looks only at the current situation, not at a past ‘history’
- DP is good when sub-problems overlap, when they’re not independent
  - No need to repeat the calculation to solve them
  - Dynamic programming has stored them, so doesn’t repeat the calculation

Process for Dynamic Programming

1. Recognize what the sub-problems are

2. Identify the recursive structure of the problem in terms of its sub-problems
   - At the top level, what is the “last thing” done?
   - This helps you see a recursive solution for any generic sub-problem in terms of smaller sub-problems

3. Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time

4. Develop an algorithm that loops through data structure solving each sub-problem one at a time
   - Bottom-up: from smallest sub-problems, to next largest, …, to complete problem

Problems Solved with Dyn. Prog.

- Log cutting (first example, uses list data structure)
- 0/1 knapsack problem
- Coin changing with “non-standard” coin selection
- Longest common subsequence
- Multiplying a sequence of matrices
  - Can do in various orders: (AB)C vs. A(BC)
  - Pick order that does fewest number of scalar multiplications

And ones we might not get to:

- All-pairs shortest paths (Floyd’s algorithm)
- Constructing optimal binary search trees

Log Cutting

Given a log of length n, and
a list (of length n) of prices P ($P[i]$ is the price of a cut of size i)
Find the best way to cut the log to maximize our profit.
(Imagine we can sell each piece of the log at price $P[i]$)

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

Select a list of lengths $\ell_1, \ldots, \ell_k$ such that:
$\sum \ell_i = n$
to maximize $\sum P[\ell_i]$

Brute Force: $O(2^n)$

Dynamic Programming

- Requires Optimal Substructure
  – Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Formulate a data structure (array, table) that can look-up solution to any sub-problem in constant time
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively. (Using memorization – we'll do later!)
     - "Bottom Up": Iteratively solve smallest to largest

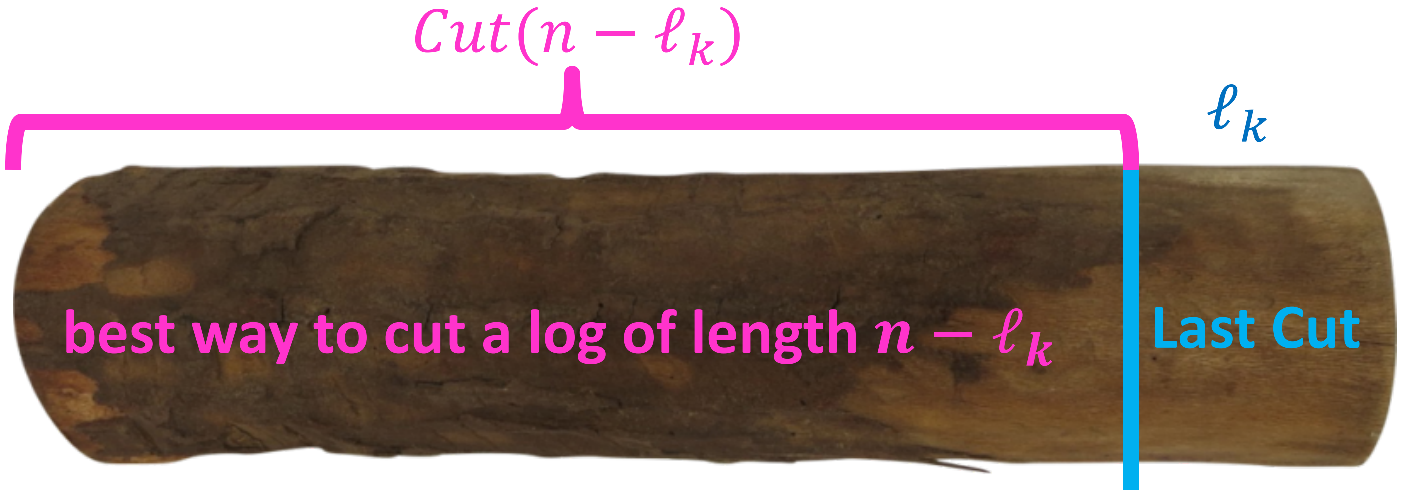
1. Identify Recursive Structure

$P[i]$ = value of a cut of length i

$Cut(n)$ = value of best way to cut a log of length n

$$Cut(n) = \max \begin{cases} Cut(n-1) + P[1] \\ Cut(n-2) + P[2] \\ \dots \\ Cut(0) + P[n] \end{cases}$$

So for a given value of *n*, to find *Cut(n)*, we need sub-problem solutions for *Cut(n-1)* down to *Cut(0)*.



What's the problem with a top-down recursive approach?

Dynamic Programming

- Requires Optimal Substructure
  - Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify the recursive structure of the problem
     - What is the "last thing" done?
  2. Save the solution to each subproblem in memory
  3. Select a good order for solving subproblems
     - "Top Down": Solve each recursively
     - "Bottom Up": Iteratively solve smallest to largest

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(0) = 0$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

0

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(1) = Cut(0) + P[1]$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(2) = \max \begin{cases} Cut(1) + P[1] \\ Cut(0) + P[2] \end{cases}$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(3) = \max \begin{cases} Cut(2) + P[1] \\ Cut(1) + P[2] \\ Cut(0) + P[3] \end{cases}$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

3

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

3. Select a Good Order for Solving Subproblems

Solve smallest sub-problem first

$$Cut(4) = \max \begin{cases} Cut(3) + P[1] \\ Cut(2) + P[2] \\ Cut(1) + P[3] \\ Cut(0) + P[4] \end{cases}$$

$Cut(i)$:	0										
Length:	0	1	2	3	4	5	6	7	8	9	10

4

Price:	1	5	8	9	10	17	17	20	24	30
Length:	1	2	3	4	5	6	7	8	9	10

Log Cutting Pseudocode

```
Initialize Memory C
Cut(n):
    C[0] = 0
    for i=1 to n:  // log size
        best = 0
        for j = 1 to i: // last cut
            best = max(best, C[i-j] + P[j])
        C[i] = best
    return C[n]
```

Run Time: $O(n^2)$

How to find the cuts?

- This procedure told us the profit, but not the cuts themselves
- Idea: remember the choice that you made, then backtrack

Remember the choice made

```
Initialize Memory C, Choices
Cut(n):
    C[0] = 0
    for i=1 to n:
        best = 0
        for j = 1 to i:
            if best < C[i-j] + P[j]:
                best = C[i-j] + P[j]
                Choices[i]=j
        C[i] = best
    return C[n]
```

Gives the size of the last cut

Reconstruct the Cuts

- Backtrack through the choices

Choices:	0	1	1	2	4	3	4	1	2	4	3
Length:	0	1	2	3	4	5	6	7	8	9	10
		1	2				6	7			

Example to demo Choices[] only. Profit of 20 is not optimal!

Backtracking Pseudocode

```
i = n
while i > 0:
        print Choices[i]
        i = i – Choices[i]
```

Our Example: Getting Optimal Solution

i	0	1	2	3	4	5	6	7	8	9	10
C[i]	0	1	5	8	10	13	17	18	22	25	30
Choices[i]	0	1	2	3	2	2	6	1	2	3	10

- If n were 5
  - Best score is 13
  - Cut at Choices[n]=2, then cut at Choices[n-Choices[n]]= Choices[5-2]= Choices[3]=3
- If n were 7
  - Best score is 18
  - Cut at 1, then cut at 6