PROBLEM SOLVING 10

CS340

Dynamic Programming: Rod Cutting, Subset Sum, Knapsack, Sequence Alignment

Sequence Alignment

- O_CURR**A**NCE
- OCCURR**E**NCE
- How do the 2 words align?
  - A gap must be added to ocurrance
  - An A must be replaced by an E

Gap and Mismatch Penalties

- δ is a gap penalty.
  - Each gap we insert incurs δ cost
  - $\delta > 0$
- α is a mismatch cost
  - For each pair of letters p,q, there is a cost $\alpha_{p,q}$ for lining up letters that do not match.
  - In general, $\alpha_{p,p} = 0$. No cost to exact matches.
- δ and α are external parameters that must be determined.

Which Alignment is Preferred?

- O_CURR**A**NCE
- OCCURR**E**NCE

VS

- O_CURR_ANCE
- OCCURRE_NCE

- Which is better?
- The first is better if $\delta + \alpha_{ae} < 3\delta$

Optimal Alignment Truth

- In an optimal alignment M of 2 strings X and Y, at least one of the following is true:
  - (m, n) ∈ M
  - the mth position of X is not matched
  - the nth position of Y is not matched

  - opt(i, j) = min[
    $\alpha_{x_i\,y_j}$ + opt(i - 1, j - 1) ↗
    δ + opt(i - 1, j) ⟶
    δ + opt(i, j - 1) ↑
    ]

An example

S	8				
M	6				
A	4				
H	2				
-	0	2	4	6	8
	-	C	L	A	M

α vowel/vowel = 1
α consonant/consonant = 1
α vowel/consonant = 3
$\delta = 2$

MATCH
HAMS and
CLAM

H_AMS
CLAM_

Rod Cutting

- How to cut steel rods into pieces in order to maximize the revenue you can get?
  - Each cut is free. Rod lengths are always an integral number of inches.
  - Input: A length n and table of prices p_i , for i = 1, 2, … , n.
  - Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

Rod Cutting

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

Can determine optimal revenue r_n by taking the maximum of

- p_n: the price we get by not making a cut,
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...
- $r_{n-1} + r_1$.

That is,

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) .$$

- To solve the original problem, solve problems on smaller sizes. It is recursive in nature.

Rod Cutting

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

Dynamic-programming solution

- Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
- Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
- "Store, don't recompute" → time-memory trade-off.
- Can turn an exponential-time solution into a polynomial-time solution.

Rod Cutting

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

Bottom-up

- Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems we need.

Rod Cutting

Length i	0	1	2	3	4	5	6	7	8
Price p_i	0	1	5	8	9	10	17	17	20
Revenue r_i	0	1	5	8	10	13	17	18	22
Cuts s_i	0	1	2	3	2	2	6	1	2

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] be a new array
  r[0] = 0
  for j = 1 to n
      q = -∞
      for i = 1 to j
          q = max(q, p[i] + r[j - i])
      r[j] = q
  return r[n]
```

What is the time complexity?

Rod Cutting

Length i	0	1	2	3	4	5	6	7	8
Price p_i	0	1	5	8	9	10	17	17	20
Revenue r_i	0	1	5	8	10	13	17	18	22
Cuts s_i	0	1	2	3	2	2	6	1	2

Record optimal choices (locations of cuts) in addition to optimal revenues.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n
      q = -∞
      for i = 1 to j
          if q < p[i] + r[j - i]
              q = p[i] + r[j - i]
              s[j] = i
      r[j] = q
  return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  while n > 0
      print s[n]
      n = n - s[n]
```

Interview Questions

- Show, by means of a counterexample, that the following “greedy” strategy does not always determine an optimal way to cut rods.
  - Define the density of a rod of length i to be p_i/i, that is, its value per inch.
  - The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \le i \le n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

Length i	1	2	3	4
price	1	20	33	36
Density	1	10	11	9

Subset Sum

- You have n items {1, 2,…, n}
- Each item has a weight $\{w_1, w_2, \dots, w_n\}$
- You want add as many items as possible, without exceeding a maximum weight, W

How to solve this problem

- Items = {2, 1, 13, 4, 3, 8, 1}
- Total weight = 11

- Optimal solution(s)?

How about a greedy solution?

- Bag will hold W weight, there are 3 items
- Sort by weight, largest to smallest…
  - W/2 + 1, W/2, W/2

- Sort by weight, smallest to largest
  - 1, W/2, W/2

- What is the greedy solution?
- What is the optimal solution?

Optimal Solution

- At the first step, the first item is added to the knapsack
- At each successive step, either
  - The next item is added to the knapsack
  - The next item is not added to the knapsack
- One of these choices will lead to the optimal solution
  - It doesn't matter which if we calculate them all
- There are N items that can be added. Max weight is W
  - How many distinct problems are there?

Optimal Solution

- There are actually only n x W distinct subproblems to consider!
- Hence: Create an n x W matrix
- Each entry corresponds to the OPT total value for the first i items subject to a total weight of W

An example

	3 (4)	0						
	2 (3)	0						
i	1 (2)	0						
i-1	0	0	0	0	0	0	0	0
		0	1	2	3	4	5	6

W

$w-w_i$ $\quad\quad$ w

Items $w_i = \{2,3,4\}$
N = 3
W = 6

Each cell = OPT(i,w)
if $w < w_i$ then OPT(i,w) = OPT(i-1,w)
else OPT(i,w) = MAX(OPT(i-1,w), w_i + OPT(i-1, $w - w_i$))

An example

Items = {1,2,2,4}
N = 4
W = 6

i								
	4 (4)	0						
	3 (2)	0						
i	2 (2)	0						
	1 (1)	0						
	0	0	0	0	0	0	0	0
		0	1	2	3	4	5	6

w

What is the complexity

- What is the complexity of each computation?
- How many computations? (nxW)
- Is this polynomial time, based on the size of the input?
  - What is the size of the input?
  - What happens if W is large?
- W is not the representation of W
  - W depends on how many bits are used to encode it
  - Think about a binary representation of W requiring 2^k bits
  - If we double W, the input size increases by 1 bit, but running time doubles
  - Running time is exponential in k $= O(n2^k)$
- What was another algorithm with time complexity not based on size of input?

The Knapsack Problem

- A lot like subset sum, but each item has a value in addition to a weight.
- Value vector $\{v_1, v_2, \ldots , v_n\}$
- We want to maximize *value*, while not exceeding maximum weight. Some items might be proportionately more valuable than others.

The Knapsack Problem

- Each cell = OPT(i,w)
- if $w < w_i$ then OPT(i,w) = OPT(i-1,w)
- else OPT(i,w) = MAX(OPT(i-1,w), v_i + OPT(i-1, w- w_i)

What's different?

The Knapsack Problem

Weights = {2,1, 2, 4}
Values = {4, 6, 1, 8}
N = 4
W = 6

i

4	0						
3	0						
2	0						
1	0						
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

w

Interview Questions

- How about a greedy “density-based” algorithm for knapsack? W=6

Weight	Value	Density
1	14	
2	22	
3	18	
4	36	
5	50	