CSC 373 — Lecture Summary for Week 3 — Winter 2019

Dynamic Programming

Example: Activity Scheduling with Profits

- Just like activity scheduling but each activity has a "profit" and we want schedule with maximum profit. More precisely:
    - Input: Activities A_1=(s_1,f_1,w_1), ..., A_n=(s_n,f_n,w_n) where s_i = start time, f_i = finish time, w_i = profit -- all non-negative integers with s_i < f_i.
    - Output: Subset of activities S (_ {A_1,A_2,...,A_n} such that no activities in S overlap and profit(S) is maximal.

- Greedy by finish time doesn't work:

```
|-1-|          |-1-|
 |-------3-------|
```

- Greedy by max profit doesn't work:

```
|----------2----------|
 |-1-| |-1-| ... |-1-|
```

- Greedy by max unit profit doesn't work: first counter-example above

- Combinations (e.g., try by finish time then by max profit) don't work:

```
|-1-|          |-1-|  |-2-| |-2-| |-2-|
 |-------3-------|  |--------5--------|
```

- What next? Turns out no other sorting strategy will make greedy work. Back to brute force? Not necessarily...

- Sort activities by finish time, as before (f_1 <= ... <= f_n). Consider optimal schedule S. Two possibilities: A_n (- S or A_n !(- S. If A_n (- S, rest of S must consist of optimal way to schedule activities A_1,...,A_k, where k is largest index of activities that do not overlap with A_n (i.e., A_{k+1},...,A_{n-1} all overlap with A_n). If A_n !(- S, S must consist of optimal way to schedule activities A_1,...,A_{n-1}.

  In other words, problem has recursive structure: optimal solutions for A_1,...,A_n = optimal solution for A_1,...,A_{n-1} or optimal solution for A_1,...,A_k together with A_n, where k is largest index of activity that does not overlap with A_n.

- Recursive solution:
  Define p[i] = largest index of activity that does not overlap with A_i, for i=1,2,...,n (so A_{p[i]} does not overlap with A_i but A_{p[i]+1},...,A_{i-1} all overlap with A[i] -- degenerate case: p[i]=0 if A_i overlaps with all of A_1,A_2,...,A_{i-1}). Assume values of p[] computed once and stored in an array -- part of initial processing, like sorting by finish times.

```
# Return the maximum profit possible from activities A_1,...,A_n.
RecOpt(n):
    if n = 0:  return 0
    else:  return max( RecOpt(n-1), w_n + RecOpt(p[n]) )
```

Correctness is immediate from reasoning above: either schedule A_n or don't, and since we don't know which choice leads to best schedule, just try both.
Runtime? Exponential! On any instance with not too much overlap, many repeated recursive calls (like recursive Fibonacci).

- Memoization:
  There are only n+1 subproblems to solve: RecOpt[0], ..., RecOpt[n].
  Exponential runtime of recursive algorithm due to wasted time recomputing values.
  Idea: store values in an array and compute each only once, looking it up afterwards (must store sentinel value in array locations not yet computed).
  Let OPT[i] = max profit from scheduling activities from {A_1,...,A_i}.
  By reasoning above, we know either OPT[i] = OPT[i-1] or OPT[i] = w_i + OPT[p[i]]. We just have to check to figure out which one.

```
    for i <- 1,...,n:
        OPT[i] <- oo  # represents "empty"
    OPT[0] <- 0

    MemRecOpt(n):
        if OPT[n] = oo:
            OPT[n] <- max( MemRecOpt(n-1), w_n + MemRecOpt(p[n]) )
        return OPT[n]
```

  Correctness: as before.
  Runtime? O(n): time for work outside 'if' statement is O(1). For each value of n, condition 'OPT[n] = oo' will be true at most once, so each value of OPT[] computed at most once. Hence, at most n+1 calls to MemRecOpt made in total.

- Iterative bottom-up algorithm:

```
    OPT[0] <- 0
    for i <- 1,2,...,n:
        OPT[i] <- max( OPT[i-1], w_i + OPT[p[i]] )
```

  Correctness and runtime as before, but avoids overhead of recursion (at the expense of computing every value in OPT, even if some of them may not be needed).

- Compute optimal answer:

```
    S <- {}
    i <- n
    while i > 0:
        if OPT[i] = OPT[i-1]:  # don't schedule job i
            i <- i - 1
        else:  # schedule job i
            S <- S u {i}
            i <- p[i]
    return S
```

- Runtime? \Theta(n) assuming p[i] precomputed and jobs already sorted by finish time; \Theta(n log n) otherwise.

Dynamic Programming Paradigm:

- For optimization problems that satisfy the following properties:
  . "subproblem optimality": an optimal solution to the problem can

always be obtained from optimal solutions to subproblems;
  . "simple subproblems": subproblems can be characterized precisely using a constant number of parameters (usually numerical indices);
  . "subproblem overlap": smaller subproblems are repeated many times as part of larger problems (for efficiency).

- Step 0: Describe recursive structure of problem: how problem can be decomposed into simple subproblems and how global optimal solution relates to optimal solutions to these subproblems.

- Step 1: Define an array indexed by the parameters that define subproblems, to store the optimal value for each subproblem (make sure one of the "sub"problems actually equals the whole problem).

- Step 2: Based on the recursive structure of the problem, describe a recurrence relation satisfied by the array values from step 1 (including degenerate or base cases).

  Difference between Step 1 and Step 2? Step 1 gives *meaning* of array (What does value stored at each array location represent?); Step 2 states property of these values (if all values were filled in, they would relate to one another in the way stated by the recurrence).

- Step 3: Write iterative algorithm to compute values in the array, in a bottom-up fashion, following recurrence from step 2. (Turn relationship between array values into computation.)

- Step 4: Use computed array values to figure out actual solution that achieves best value (generally, describe how to modify algorithm from step 3 to be able to find answer; can require storing additional information about choices made while filling up array in Step 3).

Matrix Chain Multiplication.

- Given matrix chain product: A_0 A_1...A_{n-1}, many ways to parenthesize (e.g., A(BC) or (AB)C). All will yield same answer but not same running time. Example: A 1x10 B 10x10 C 10x100
```
    (AB)C = 1*10*10 + 1*10*100 = 100 + 1000 = 1100 ops
    A(BC) = 10*10*100 + 1*10*100 = 10000 + 1000 = 11000 ops
```

- Matrix Chain Multiplication problem:
```
    Input: A_0, A_1, ..., A_{n-1} with dimensions
        [d_0 x d_1], [d_1 x d_2], ..., [d_{n-1} x d_n]
    Output: Fully parenthesized product with smallest total cost.
```

- Brute force algorithm:
  How many possible ways to put in parentheses? Answer is called "Catalan number" and is \Omega(4^n).

- Greedy algorithm:
  . Product with smallest cost first, or with smallest dimension eliminated first.
```
    Counter-example: 10 1 10 100
        greedy: 10 1 10 + 10 10 100 = 10100
        other:  1 10 100 + 10 1 100 = 2000
```
  . Product with smallest cost last, or with smallest dimension eliminated last, or with largest dimension eliminated first.
```
    Counter-example: 1 10 100 1000
        greedy: 10 100 1000 + 1 10 1000 = 1,010,000
        other:  1 10 100 + 1 100 1000 = 101,000
```

- Nothing works!

0. Structure of optimal subproblems:
   - Idea: instead of trying to find where to put first product, try to find where to put *last* product. (Common way of thinking to come up with recursive problem structure.)

     $A_0 (A_1 \ldots A_{n-1})$ -- last product costs $d_0 d_1 d_n$
     $(A_0 A_1) (A_2 \ldots A_{n-1})$ -- last product costs $d_0 d_2 d_n$
     ... ...
     $(A_0 \ldots A_{n-2}) A_{n-1}$ -- last product costs $d_0 d_{n-1} d_n$
   - Only n-1 possibilities. What information would help us find best answer? Knowing best cost of doing each subproduct.
   - IMPORTANT: best overall solution must include optimal subproducts (otherwise, could improve on best overall).

1. Definition of array of subproblem values:
   - Subproblems? Must consider arbitrary subproduct $A_i \ldots A_j$.
   - $N[i,j]$ = smallest cost of multiplying $A_i \ldots A_j$
   - From structure of optimal solution, best way of doing $A_i \ldots A_j$ (including all parentheses) must have the form

     $(A_i \ldots A_{k-1}) \times (A_k \ldots A_j)$

     for some $i < k \le j$, where subproducts $A_i \ldots A_{k-1}$ and $A_k \ldots A_j$ are done in the best way possible (otherwise wouldn't be best overall).

2. Array recurrence:
   From reasoning above, $N[i,i] = 0$ and for $i < j$,
   $N[i,j] = \min\{ d_i d_k d_{j+1} + N[i,k-1] + N[k,j] : i < k \le j \}$

3. Bottom-up algorithm:
   - Basic recursive solution suffers from combinatorial explosion: many subproblems recomputed multiple times, yielding exponential runtime -- even though only need to compute n^2 values in total.
   - Instead of recomputing values many times, compute smaller values first and store them in an array to be looked up (so we never need to make recursive calls).
   - Constraint: must compute values so that all entries $N[i,k-1]$ and $N[k,j]$ are present by the time $N[i,j]$ is computed. For example,

```
MatrixChain(d, n):
    for i <- n-1,...,0:  # largest down to smallest!
        N[i,i] <- 0
        for j <- i+1,...,n-1:
            N[i,j] <- oo
            for k <- i+1,...,j:
                temp <- d[i]*d[k]*d[j+1] + N[i,k-1] + N[k,j]
                if temp < N[i,j]:
                    N[i,j] <- temp
    return N[0,n-1]
```

   - Trace on example input [2, 3, 5, 1, 8] -- not done in lecture snapshots for each value of i (each row filled left-to-right):

i = 3:

	0	1	2	3
0				
1				
2				
3				0

i = 2:

	0	1	2	3
0				
1				
2			0	40
3				0

i = 1:

	0	1	2	3
0				
1		0	15	39
2			0	40
3				0

i = 0:

	0	1	2	3
0	0	30	21	37
1		0	15	39
2			0	40
3				0

Running time: \Theta(n^3) (nested loops iterating \Theta(n) times).

4. Reconstruct solution: Once values are filled in, how to find actual parenthesized expression?

Possibility: working from N[0,n-1], recompute all possibilities considered in order to find breakpoint k that yielded best value; then recursively do the same for each subproblem. This requires additional \Theta(n^3) time. Instead, use another array B[i,j] to store best value of k used to achieve N[i,j], modify original algorithm to compute values of B at the same time as N. At the end, B[0,n-1] = index of last multiplication to perform, and we can recursively print each subproduct.

```
MatrixChain(d, n):
    for i <- n-1,...,0:
        N[i,i] <- 0
        B[i,i] <- i
        for j <- i+1,...,n-1:
            N[i,j] <- oo
            B[i,j] <- n
            for k <- i+1,...,j:
                temp <- d[i]*d[k]*d[j+1] + N[i,k-1] + N[k,j]
                if temp < N[i,j]:
                    N[i,j] <- temp
                    B[i,j] <- k
    parenthesize(B, 0, n-1)
# print best way to compute A_i...A_j
parenthesize(B, i, j):
    if i = j:  print "A_i"
    else:
        print "("
        parenthesize(B, i, B[i,j]-1)
        print " x "
        parenthesize(B, B[i,j], j)
        print ")"
```

For example with dimensions [2 3 5 1 8] -- not done in lecture:

i = 3:

	0	1	2	3
0				
1				
2				
3				3

i = 2:

	0	1	2	3
0				
1				
2			2	3
3				3

i = 1:

	0	1	2	3
0				
1		1	2	3
2			2	3
3				3

i = 0:

	0	1	2	3
0	0	1	1	3
1		1	2	3
2			2	3
3				3

Result of parenthesize(B, 0, 3): ((A_0 x (A_1 x A_2)) x A_3)

For Next Week

* Readings: Section 6.6, subsection "All-pairs shortest paths".
* Self-Test: Write out the full algorithm, following the steps outlined in class (define an array, give a recurrence, etc.)