Week 8: Dynamic programming

Agenda:

- 0/1 Knapscak
- Chain Matrix Multiplication

Reading:

- Textbook pages 331-349

Dynamic programming introduction:

- An algorithm design technique
- Usually for optimization problems
- Typically like divide-and-conquer uses solutions to subproblems to solve the problem, BUT
- Key idea:
  - *Avoids re-computation*
  - of repeated subproblems by storing subproblem answers in tables/arrays
- 1st example problem — Fibonacci numbers

$$f(n) = \begin{cases} n, & \text{when } n = 0, 1 \\ f(n-1) + f(n-2), & \text{when } n \geq 2 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9
$f(n)$	0	1	1	2	3	5	8	13	21	34

Question: how do we compute $f(n)$?

1^{st} Naive Fibonacci implementation – recursion

- Pseudocode:

  procedure $f(n)$

  if $n < 2$ **then**
      return n
  else
      return $f(n-1) + f(n-2)$

- Recursion tree:

$$f(5)$$

$$f(4) \qquad\qquad\qquad f(3)$$

$$f(3) \qquad f(2) \qquad f(2) \qquad f(1)$$

$$f(2) \quad f(1) \quad f(1) \quad f(0) \quad f(1) \quad f(0)$$

$$f(1) \quad f(0)$$

- Notice that there are a lot of repeated function calls

- Running time recurrence

$$T(n) = \begin{cases} c_1, & \text{when } n = 0, 1 \\ c_2 + T(n-1) + T(n-2), & \text{when } n \geq 2 \end{cases}$$

- Conclusion: $T(n) > f(n) \longrightarrow T(n) \in \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$

- Problem with the 1st implementation — repeated function calls

- Key idea:

  Do the computation in a bottom up manner, and
  Store the compute values for future use.

  - Define array $F[0..n]$ where $F[i]$ is going to store $f(i)$.
  - Fill in the values of $F[0]$ and $F[1]$ from the definition (initialization)
  - Start computing $F[i]$, from $i = 2$ onward, using the recurrence $F[i] \longleftarrow F[i-1] + F[i-2]$.
  - Each time, we want to compute $F[i]$, $F[i-1]$ and $F[i-2]$ are *already computed!* (bottom up).

2nd Fibonacci implementation — dynamic programming

- Pseudocode:

  procedure Dynfib(n)

  $F[0] \leftarrow 0$
  $F[1] \leftarrow 1$
  for $j \leftarrow 2$ **to** n **do**
  $\quad F[j] \leftarrow F[j-1] + F[j-2]$
  return $F[n]$

- Running time

$$T(n) \in \Theta(n)$$

General steps in designing a dynamic programming solution:

- Step 1: Describe an array of values that you want to compute.

  Do not say how you compute the array, but define what you store in the array. (e.g. array $F[0..n]$ and $F[i]$ is going to hold $f(i)$).

- Step 2: Give a recurrence relating some values in the array to other values in the array.

  The basis of your recurrence should specify how to initialize the array (e.g. $F[0] = 0$, $F[1] = 1$, and $F[i] = F[i-1] + F[i-2]$).

- Step 3: Give a high level program to compute the entries of the array using the recurrence above.

- Step 4: State how to extract the solution from the array (e.g. return $F[n]$).

2nd example problem — Knapsack

- We have a knapsack with capacity W
- n items with weights $w_1, \ldots, w_n \in \mathbb{N}$ and values $v_1, \ldots, v_n \in \mathbb{N}$.
- Want to fill the knapsack with items to maximize the value without exceeding its capacity.
- Formally, for each $S \subseteq \{1, \ldots, n\}$ let $K(S) = \sum_{i \in S} w_i$.
- Find $M = \max_{S \subseteq \{1,\ldots,n\}} \{K(S) | K(S) \leq W\}$.
- Example: $w_1 = 10$, $w_2 = 10$, $w_3 = 15$, $v_1 = 10$, $v_2 = 10$, $v_3 = 16$, $W = 20$.
  Greedy picks item 3 whereas the optimal is to pick 1 and 2.

1st (naive) solution: try all possible subsets of items and select the best.

- Consider each set S of all 2^n possible subsets of $\{1, \ldots, n\}$.
- Compute the weight and value of set S.
- Find the set with maximum value among those that have $K(S) \leq W$.
- Running time: at least $\Omega(2^n)$ subsets to consider!!

2nd solution: use Dynamic programming.

- Step 1: Define array $A[i, D]$, $0 \leq i \leq n$ and $0 \leq D \leq W$ where $A[i, D]$ is the value of best possible knapsack of weight at most D using only items from 1 to i. Final sol. value: $A[n, W]$.

- Step 2: How to compute $A[i, D]$?
  - If $i = 0$ or $D = 0$ then trivially $A[i, D] = 0$.
  - Else, consider item i:
    * If we do not choose item i: knapsack must be packed optimally with items from $1 \ldots (i - 1)$.
    * If we choose item i (assuming $D \geq w_i$): rest of $D - w_i$ remaing cap. must be packed with items $1 \ldots (i - 1)$.
    * So $A[i, D] = \max \begin{cases} A[i-1, D] \\ (\text{if } D \geq w_i) \; v_i + A[i-1, D - w_i] \end{cases}$

- Step 3:

```
procedure Knapsack

for i ← 1 to n do
    A[i,0] ← 0
for D ← 0 to W do
    A[0,D] ← 0
for i ← 1 to n do
    for D ← 1 to W do
        A[i,D] ← A[i−1,D]
        if D ≥ w_i and A[i,D] < A[i−1,D−w_i] + v_i then
            A[i,D] ← A[i−1,D−w_i] + v_i
return A[n,W]
```

- The running time is $O(nW)$ since the are only a constant number of opertion per each iteration of the inner loop.

- Step 4: How to find the set of items of the optimal packing?

- Consider item n. It can be seen that if $A[n, W] = A[n-1, W]$ then item n is not in the optimal solution. Else it is in the solution and $A[n, W]$ is obtained from $A[n-1, W-w_n]$ by adding item n.

- We can so go to this new entry to find out if item $n-1$ is in the solution or not.

- This suggests the following algorithm to find the optimal solution itself:

```
procedure Print-Opt-Knapsack (i, D)

If i = 0 or D = 0 then
   return
Else
   If A[i, D] ≠ A[i − 1, D] then
      Print (i)
      Print-Opt-Knapsack (i − 1, D − w_i)
   else if A[i, D] = A[i − 1, D] then
      Print-Opt-Knapsack (i − 1, D)
```

Matrix-chain multiplication:

- Input: matrices A_1, A_2, ..., A_n with dimensions $d_0 \times d_1$, $d_1 \times d_2$, ..., $d_{n-1} \times d_n$, respectively.

- Output: an order in which matrices should be multiplied such that the product $A_1 \times A_2 \times \ldots \times A_n$ is computed using the minimum number of scalar multiplications.

- Fact: suppose A_1 is a $d_1 \times d_2$ matrix, A_2 is a $d_2 \times d_3$ matrix.

  Then A_1 and A_2 is multipliable, and $B = A_1 \times A_2$ can be computed using $d_1 \times d_2 \times d_3$ scalar multiplications.

- Example: $n = 4$ and $(d_0, d_1, \ldots, d_n) = (5, 2, 6, 4, 3)$

  Possible orders with different number of scalar multiplications:

$((A_1 \times A_2) \times A_3) \times A_4$	$5 \times 2 \times 6 + 5 \times 6 \times 4 + 5 \times 4 \times 3 = 240$
$(A_1 \times (A_2 \times A_3)) \times A_4$	$5 \times 2 \times 4 + 2 \times 6 \times 4 + 5 \times 4 \times 3 = 148$
$(A_1 \times A_2) \times (A_3 \times A_4)$	$5 \times 2 \times 6 + 5 \times 6 \times 3 + 6 \times 4 \times 3 = 222$
$A_1 \times ((A_2 \times A_3) \times A_4)$	$5 \times 2 \times 3 + 2 \times 6 \times 4 + 2 \times 4 \times 3 = 102$
$A_1 \times (A_2 \times (A_3 \times A_4))$	$5 \times 2 \times 3 + 2 \times 6 \times 3 + 6 \times 4 \times 3 = 138$

1^{st} Matrix-chain multiplication — Recursion:

- Let $T(n)$ be the number of multiplication orders for n matrices.

  How big is $T(n)$???

n	1	2	3	4	5	6	...
$T(n)$	1	1	2	5	14	42	...

- Consider the highest level parenthesis: $(A_1 \ldots A_i)(A_{i+1} \ldots A_n)$.

- There are $n-1$ possiblities: i can be anyware between 1 to $n-1$ (e.g $A_1(A_2 \ldots A_n)$ to $(A_1 \ldots A_{n-1})A_n$.

- The number of ways to put parentheses for each of $(A_1 \ldots A_i)$ and $(A_{i+1} \ldots A_n)$ is $T(i)$ and $T(n-i)$, respectively.

- Therefore:

  $$T(n) = \begin{cases} 1, & \text{when } n = 0, 1 \\ \sum_{i=1}^{n-1} T(i) \times T(n-i), & \text{when } n \geq 2 \end{cases}$$

- Solving this recurrence (not easy) shows: $T(n) = \frac{\binom{2n}{n}}{n+1} \approx \frac{4^n}{n\sqrt{\pi n}}$

- Recursive program will have similar running time: $\Omega(3^n)$.

- Cannot afford this!!

Use dynamic programming:

- Step 1: Define $M[i, j]$ $(1 \leq i \leq j)$: the minimum number of scalar multiplications needed to compute product $A_i \times A_{i+1} \times \ldots \times A_j$ $(i \leq j)$

- Step 2: The recurrence to fill in the entries of the array:
$$M[i,j] = \begin{cases} 0, & \text{if } i = j \\ \min_{i \leq k < j}\{M[i,k] + M[k+1,j] + d_{i-1}d_kd_j\}, & \text{if } i < j \end{cases}$$

- for example,
$$M[1,4] = \min \left\{ \begin{array}{l} M[1,1] + M[2,4] + d_0 \times d_1 \times d_4 \\ M[1,2] + M[3,4] + d_0 \times d_2 \times d_4 \\ M[1,3] + M[4,4] + d_0 \times d_3 \times d_4 \end{array} \right\}$$

- Step 3: Pseudocode (to obtain the optimal cost):

```
procedure dpM(1, n)

for i ← 1 to n do
    M[i, i] ← 0
for shift ← 1 to n do
    for i ← 1 to n − shift do
        j ← i + shift
        M[i, j] ← ∞
        for k ← i to j − 1 do
            new ← M[i, k] + M[k + 1, j] + d_{i−1} × d_k × d_j
            if new < M[i, j] then
                M[i, j] ← new
return M[1, n]
```

- To obtain the actual ordering:

```
procedure dpM(1, n)

for i ← 1 to n do
    M[i, i] ← 0
for shift ← 1 to n do
    for i ← 1 to n − shift do
        j ← i + shift
        M[i, j] ← ∞
        for k ← i to j − 1 do
            new ← M[i, k] + M[k + 1, j] + d_{i−1} × d_k × d_j
            if new < M[i, j] then
                M[i, j] ← new
                S[i, j] ← k
return M[1, n]
```

- We call Print-Opt-Order $(S, 1, n)$:

```
procedure Print-Opt-Order(S, i, j)

If i = j then
    Print ("A_i")
Else
    Print ``(``
        Print-Opt-Order (S, i, S[i, j])
        Print-Opt-Order (S, S[i, j] + 1, j)
        Print ``)''
```

- Trace the example $n = 4$ and $(d_0, d_1, \ldots, d_n) = (5, 2, 6, 4, 3)$:

m-matrix

			4		1			
		j 3		102		2 i		
	2		88		72		3	
1		60		48		72		4
	0		0		0		0	
	A_1		A_2		A_3		A_4	

s-matrix

			4		1			
		j 3		1		2 i		
	2		1		3		3	
1		1		2		3		4
	–		–		–		–	
	A_1		A_2		A_3		A_4	

- The innermost `for` loopbody takes constant time ...

  So `dpM`(n) worst case running time $\in \Theta(n^3)$.

- Some final observations:
  - Suppose we have computed the order of multiplications
  - And the last multiplication is between $(A_1 \times \ldots \times A_j)$ and $(A_{j+1} \times \ldots \times A_n)$
  - Then the suborders $(A_1 \times \ldots \times A_j)$ and $(A_{j+1} \times \ldots \times A_n)$ are optimal orders for the subproblems, respectively.
  - We call this ... *optimal substructures*
  - Equivalently, we need to compute optimal orders for
    * multiplying matrices $A_1, A_2, \ldots, A_j$
    * multiplying $A_{j+1}, A_{j+2}, \ldots, A_n$,
  - for every $1 \le j \le n - 1$, and combine them into an order to multiplying $A_1, A_2, \ldots, A_n$
  - choose the best order out of the $(n - 1)$ possibilities

Dynamic programming key characteristics:

- Recurrence relation exists
- Recursive calls overlap
- Small number of subproblems
- Huge number of calls
- Avoid re-computation
- Bottom-up computation
- Top-down trace

Other problems suited to Dynamic programming:

- String matching: Longest Common Subsequence (next lecture)
- Optimal binary search tree construction (textbook page 356)
- All pair shortest paths in (di)graphs (CMPUT 304)
- Optimal layout in VLSI (could be a thesis topic :-))