Chapter 6 - Dynamic Programming

Lenny Wu   Professor Cho-Jui Hsieh

May 2021

Example: Fibonacci numbers.

$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 5, F_5 = 8, ...$

Goal: compute n^{th} Fibonacci number ($n \geq 0$).

Recursive approach:

```
F(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return F(n - 1) + F(n - 2)
```

Problem with recursive approach: repeated function calls with the same input value; this wastes time and space (exponential time complexity). Instead, we can use dynamic programming with a top-down or bottom-up approach.

```
Fibonacci(n) (top-down):
   M = []   // size n + 1
   M[0] = 0, M[1] = 1:

    if M[n] != empty:
        return M[n]
    else:
       M[n] = F(n - 1) + F(n - 2)
        return M[n]

Fibonacci(n) (bottom-up):
   M = []   // size n + 1
   M[0] = 0, M[1] = 1

    for i = 2, 3, ..., n:
       M[i] = M[i - 1] + M[i - 2]

    return M[n]
```

Weighted Scheduling

Input: n jobs, each job has:

- s_n: starting time
- f_i: finishing time
- w_i: weight (value) of the job

Goal: find a set of compatible jobs S to maximize the total weight $\sum_{i \in S} w_i$.

DP: sort jobs by finishing time.
$f_1 \leq f_2 \leq \ldots \leq f_n$
$OPT(i)$: optimal total weight solution for subproblem with jobs $\{1, ..., i\}$.
The final solution is $OPT(n)$.

$$OPT(i) = \begin{cases} OPT(i-1) & \text{if job } i \text{ is not selected} \\ w_i + OPT(P_i) & \text{if job } i \text{ is selected} \end{cases}$$

P_i = max index such that P_i is not overlapping with i.
$OPT(i) = max(OPT(i-1), w_i + OPT(P_i)$.

Algorithm:

```
DP (bottom up):
    M = []   // size n + 1
    M[0] = 0

    for i = 1, ..., n:
        // M[i] = OPT(i)
        M[i] = max(M[i - 1], w_i + M[P_i])

    return M[n]

// computing P_i:
for each i:
    find the last j < i such that f_j <= s_i (==> not
       overlapping)
        => binary search (f_1 <= f_2 <= ... <= f_{i - 1}
```

Time complexity: binary search with $f_1 \leq f_2 \leq \ldots \leq f_{i-1}$ is $O(logn)$ for each i. Thus, the total is $O(nlogn)$.

Print the job list:

```
A = []   // size n + 1

i = n
while (i > 0):
```

```
        if M[i - 1] < w_i + M[P_i]:
            A = A + {i}
            i = P_i
        else:
            i -= 1
```

Time complexity: $O(n)$.

Knapsack: n items with values $v_1, v_2, ..., v_n$, weights $w_1, w_2, ..., w_n$.
find a subset S of items to maximize $\sum_{i \in S} v_i$ under the constraint $\sum_{i \in S} w_i \leq W_{max}$.

DP: $OPT(i, w)$: the solution with item $\{1, ..., i\}$

$$OPT(i, w) = \begin{cases} v_i + OPT(i-1, w-w_i) & \text{if choosing item } i \text{ (if } w \geq w_i) \\ OPT(i-1, w) & \text{if not choosing } i \end{cases}$$

$$\longrightarrow OPT(i, w) = max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\}$$

Algorithm:

```
M = []   // size (n + 1) * (w_{max} + 1)
M[0][w] = 0 for all w

for i = 1, ..., n:
    for w = 0, ..., w_max:
        if w > w_i:
            M[i, w] = max(M[i - 1, w], v_i + M[i - 1, w - w_i
                ])
        else:
            M[i, w] = M[i - 1, w]

return M[n, w_max]
```

Time complexity: $O(n \cdot W_{max})$ (not polynomial to input size; input value $\neq$ size).

RNA Secondary Structure

Input: a sequence of $\{A, U, C, G\}$, and $B = b_1, \dots, b_n$.

Secondary structure is a set of pairs in this sequence which satisfies the following:

1. can only pair $(A, U), (U, A), (C, G), (G, C)$,
2. (non-sharp): pair (b_i, b_j) has to satisfy $i \le j - 4$,
3. (non-crossing): for any two pairs $(b_i, b_j), (b_k, b_l)$, cannot have $i < k < j < l$.

Goal: find the maximum number of pairs that satisfy these 3 conditions.

1D DP (first attempt):
    $OPT(i)$: optimal solution of $b_1, \dots, b_i$.

2D DP:
    $OPT(i, j)$: solution (max number of pairs) in $b_i, b_{i+1}, \dots, b_j$.

Consider $\underbrace{b_i, b_{i+1}, \dots, b_{t-1}}_{A}\ \underbrace{b_t, \dots, b_j}_{B}$.

$$OPT(i,j) = \begin{cases} OPT(i, j-1) & \text{if } b_j \text{ is not in any pair} \\ 1 + OPT(i, t-1) + OPT(t+1, j-1) & \text{if } b_j \text{ is paired with some } b_t,\ i \le t \le j-4 \end{cases}$$

With b_t such that it can be paired with b_j, we have:

$$\longrightarrow OPT(i,j) = max\{OPT(i, j-1), max_{i \le t \le j-4} 1 + OPT(i, t-1) + OPT(t+1, j-1)\}$$

In paired case:

- 1 comes from (b_t, b_j),
- $OPT(i, t-1) = \#$ pairs in A,
- $OPT(t+1, j-1) = \#$ pairs in B.

Algorithm:

```
OPT[i, j] = 0 when j <= i + 4    // initialize the table
 for j = 1, ..., n:
      for i = 1, ..., j:
           compute OPT[i, j]    // for loop inside
```

Another ordering (increasing order of $|i - j|$):

```
 for k = 5, ..., n - 1:
      for i = 1, ..., n - k:
           j = i + k
           compute OPT[i, j]
```

Time complexity: $O(n^3)$.

RNA Sequence Alignment

Input: two strings:

- $X = x_1, ..., x_m$
- $Y = y_1, ..., y_n$

Example:

$X = CUACCG$
$Y = UACAUG$

Goal: insert, delete, and/or substitute to transform X into Y; find the alignment with the smallest cost.

$$\text{Cost of each operation} = \begin{cases} \delta & \text{insert or delete} \\ \alpha_{x_i,y_j} & \text{substitute } x_i \text{ into } y_j \end{cases}$$

Worst case:

```
_ _ _ _ _ _ C U A C C G
U A C A U G _ _ _ _ _ _
```

The alignment cost of this is $\delta \cdot (n+m)$.

Special case: no substitution; equivalent to longest common subsequence (LCS):

```
// in this example, LCS = 4:
C U A C C _ _ G
_ U A C _ A U G
```

The alignment cost of this is $\delta \cdot (n + m - 2 \cdot LCS)$.

2D DP:

$OPT(i, j) =$ min alignment cost for $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$.
The final solution is $OPT(m, n)$.

$$OPT(i,j) = \begin{cases} OPT(i-1, j-1) + \alpha_{x_i,y_j} & \text{match } (x_i, y_j) \\ OPT(i-1, j) + \delta & \text{delete } x_i \\ OPT(i, j-1) + \delta & \text{insert } y_i \text{ into } X \end{cases}$$

$$\longrightarrow OPT(i,j) = min\{OPT(i-1, j-1) + \alpha_{x_i,y_j}, OPT(i-1, j) + \delta, OPT(i, j-1) + \delta\}$$

Algorithm:

```
M = []  // size (m + 1) * (n + 1)
M[i, 0] = i * delta for all i
M[0, i] = i * delta for all i

 for i = 1, ..., m:
```

```
        for j = 1, ..., n:
            M[i, j] = min(  M[i - 1, j - 1] + alpha_{x_i, y_j},
                            M[i - 1, j] + delta,
                            M[i, j - 1] + delta)    )

    return M[m, n]
```

Time complexity: $O(m \cdot n)$.
Space complexity: $O(m \cdot n)$. To reduce space to linear complexity: use two 1D arrays, one storing the current row and one storing the previous row.

How to get the optimal way of alignment?
Remember the previous node for each node.
$\to$ only one path from (m, n) to $(0, 0)$
$\to$ find shortest path from $(0, 0)$ to (m, n)

Run DP with linear space:

- $f(\frac{m}{2}, q)$ = shortest path from $(0, 0)$ to $(\frac{m}{2}, q)$,
- $g(\frac{m}{2}, q)$ = shortest path from $(\frac{m}{2}, q)$ to (m, n).

How to find path with $O(m + n)$ space? Idea: divide and conquer:

- look at column $\frac{m}{2}$
- shortest path $(0, 0) \to (m, n)$ = shortest path $(0, 0) \to (\frac{m}{2}, q) \to (m, n)$

Algorithm:

```
    find_path(X, Y):
        m = len(X), n = len(Y)

        // linear space DP; O(m*n) time
        compute f(m/2, q), g(m/2, q) for all q = 1, ..., n

        q* = argmin_q  (f(n/2, q) + g(m/2, q))
        add (m/2, q*) to path

        find_path(X[1, ..., m/2], Y[1, ..., q*]        \\ (1)
        find_path(X[m/2 + 1, ..., m], Y[q*, ..., n]        \\ (2)
```

Recursive call (1) is $T(\frac{m}{2}, q)$, and recursive call (2) is $T(\frac{m}{2}, n - q)$.
$\to T(m, n) = T(\frac{m}{2}, q) + T(\frac{m}{2}, n - q) + c \cdot mn$.
$\to T(m, n) = O(mn)$.

Proof. $T(m, n) = O(mn)$.

Base case: $m = 1$, $n = 1$: $m + n = 1$ (trivial).

Inductive hypothesis: assume $T(m', n') \leq \alpha \cdot m'n'$ for all $m' + n' < m + n$.

Induction step:

$$\begin{aligned}
T(m,n) &\leq T(\frac{m}{2},q) + T(\frac{m}{2},n-q) + c \cdot mn \\
&\leq \alpha \cdot \frac{m}{2}q + \alpha \cdot \frac{m}{2}(n-q) + c \cdot mn \\
&= \frac{\alpha}{2} \cdot mn + c \cdot mn \\
&\leq \frac{\alpha}{2} \cdot mn + \frac{\alpha}{2} \cdot mn \\
&= \alpha \cdot mn.
\end{aligned}$$

Thus, $T(m,n) = O(mn)$. $\square$

Bellman-Ford

Goal: find the shortest path in a graph containing negative edges.

Why not use Dijkstra's algorithm?

1. May have negative cycles $\Rightarrow$ *cost* $\rightarrow -\infty$.
2. Even without negative cycle, Dijkstra's algorithm does not work.

DP:

$OPT(i, v)$: min-cost path from v to t with $\leq i$ edges.

$$OPT(i, v) = \begin{cases} OPT(i-1, v) & \text{only use } i-1 \text{ edges: } v - w \rightsquigarrow t \\ OPT(i-1, w) + l(v, w) & \end{cases}$$

$$\longrightarrow OPT(i, v) = min\{OPT(i-1, v), min_{\{w:(v,w)\in E\}} OPT(i-1, w) + l(v, w)\}$$

Algorithm:

```
M = []  // size n * |V|
for i = 1, ..., n - 1:
    for each node v:
        M[i, v] = min(M[i - 1, v], min(M[i - 1, w] + l(v, w))
```

When should we stop?

Theorem. *If a graph has no negative cycle, then there exists a shortest path with $\leq n-1$ edges.*

Proof. If shortest path has $\geq$ n edges, there exists a node visited twice in the path.
$\Rightarrow$ There exists a cycle in the path.
No negative cycle $\Rightarrow$ removing this cycle will not increase the cost. $\square$

Algorithm using a single array:

```
M = []  // size n
for i = 1, ..., n - 1:
    for each node v:
        // M[v] is always <= M[i, v] at each iteration i
        M[v] = min(M[v], min(M[w] + l(v, w))
```

Time complexity: $O(nm)$.

How to remember the shortest path? Remember the next node for each node:

```
next = []   // size |V|
for i = 1, ..., n - 1:
    for each node v:
        w* = argmin_w (M[w] + l(v, w))
```

```
if l(v, w*) + M[w*] < M[v]:
    next[v] = w*
    M[v] = l(v, w*)
```

What if there exists a negative cycle?

Theorem. *There exists a negative cycle that can reach* t *if and only if* $OPT(n, v) < OPT(n-1, v)$ *for some node* v.

Proof.
$(\Rightarrow)$
Suppose the graph has a negative cycle. Then there exists v such that $OPT(n, v) < OPT(n-1, v)$.
If $OPT(n, v) = OPT(n-1, v)$ for all v, then:
$OPT(n+1, v) = OPT(n-1, v)$
$\vdots$
$OPT(i, v) = OPT(n-1, v)$ for all $i \geq n$.
This contradicts with the definition of negative cycle since there exists v such that cost of path $v \rightsquigarrow t \rightarrow -\infty$.

$(\Leftarrow)$
Previous theorem: if there is no negative cycle, then there exists a shortest path with $\leq n-1$ edges. $\square$