Dynamic Programming:
Evaluating Recursive Functions

Shiyuan DENG
Department of Computer Science and Engineering
Chinese University of Hong Kong
Pitfall of Recursion

A recursive algorithm does considerable redundant work if the same subproblem is encountered over and over again.
Problem 1

Let $A$ be an array of $n$ integers. Define a function $f(x)$ — where $x \geq 0$ is an integer — as follows:

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ \max_{i=1}^{x}(A[i] + f(x - i)) & \text{otherwise} \end{cases}$$

Consider the following algorithm for calculating $f(x)$:

**algorithm** $f(x)$
1. if $x = 0$ then return 0
2. $max = -\infty$
3. for $i = 1$ to $x$
4. $v = A[i] + f(x - i)$
5. if $v > max$ then $max = v$
6. return $max$

Prove: The above algorithm takes $\Omega(2^n)$ time to calculate $f(n)$. 

Shiyuan DENG
Dynamic Programming: Evaluating Recursive Functions
We will prove the statement by induction. Executing $f(n)$ will launch function calls $f(n - 1), f(n - 2), \cdots, f(0)$.

Let $g(n)$ denote the running time of $f(n)$. So we have:

$$g(0) \geq 1,$$

$$g(1) \geq 1,$$

$$g(n) \geq \sum_{i=0}^{n-1} g(i) \text{ for } n \geq 2.$$

We will prove $g(n) \geq 2^{n-1}$ for all $n \geq 1$ by induction on $n$. 


Solution

The base case $n = 1$ is obviously correct. Next, assuming $g(n) \geq 2^{n-1}$ for $n \leq k$ where $k$ is an integer at least 1, we will prove $g(k + 1) \geq 2^k$.

As $k + 1 \geq 2$, we have:

$$g(k + 1) \geq \sum_{i=0}^{k} g(k).$$

By the inductive assumption, we have:

$$g(k + 1) \geq 1 + \sum_{i=1}^{k} 2^{k-1} = 2^k.$$
Principle of Dynamic Programming

Resolve subproblems according to a certain order. Remember the output of every subproblem to avoid re-computation.
Let $A$ be an array of $n$ integers. Define function $f(a, b)$ — where $a \in [1, n]$ and $b \in [1, n]$ — as follows:

$$f(a, b) = \begin{cases} 
0 & \text{if } a \geq b \\
(\sum_{c=a}^{b} A[c]) + \min_{c=a+1}^{b-1} \{f(a, c) + f(c, b)\} & \text{otherwise}
\end{cases}$$

Design an algorithm to calculate $f(1, n)$ in $O(n^3)$ time.
List all the subproblems.

\[ f(5, 8) \]
Solution

\[ f(a, b) = 0 \text{ when } a \geq b. \]
Solution

\[ f(a, b) = \left( \sum_{c=a}^{b} A[c] \right) + \min_{c=a+1}^{b-1} \{ f(a, c) + f(c, b) \} \text{ when } a < b. \]

Find out the dependency relationships.
Solution

\[ f(a, b) = (\sum_{c=a}^{b} A[c]) + \min_{c=a+1}^{b-1} \{ f(a, c) + f(c, b) \} \] when \( a < b \).

Let us start with the gray cells — they correspond to \( f(a, b) \) where \( a = b - 1 \). These cells depend on no other cells.
Solution

Let us continue with the green cells — they correspond to $f(a, b)$ where $a = b - 2$. Every such cell depends on two gray cells, which have already been computed.
Solution

Let us continue with the red cells — they correspond to $f(a, b)$ where $a = b - 3$. Every such cell depends on two gray cells and two green cells, all of which have been computed.
The order can be summarized as follows.

- All cells \( f(a, b) \) with \( b - a = 1 \), each computed in \( O(1) \) time.
- All cells \( f(a, b) \) with \( b - a = 2 \), each computed in \( O(2) \) time.
- ...
- All cells \( f(a, b) \) with \( b - a = k \), each computed in \( O(k) \) time.
- ...
- All cells \( f(a, b) \) with \( b - a = n - 1 \), each computed in \( O(n - 1) \) time.

There are \( O(n^2) \) values to calculate.
Total time complexity = \( O(n^3) \).
Problem 3 (Space Consumption)

In Lecture Notes 8, our algorithm for computing $f(n, m)$ used $O(nm)$ space. Next, we will reduce the space complexity to $O(n + m)$.

Recall the dependency graph:

$$
\begin{array}{c|cccc}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
y & x & B & D & C & A \\
\hline
0 & & & & & \\
1 & A & & & & \\
2 & B & & & & \\
3 & C & & & & \\
\end{array}
$$
Solution

We can calculate the values in the row-major order, i.e., row 0 to row 3 and left to right in each row. We used $O(mn)$ space because we stored all the values. Observe, however, that only two rows need to be stored at any moment.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td></td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Solution

Same idea for the column-major order.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>y</strong></td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So the space complexity is $O(\min\{m, n\})$, in addition to the $O(n + m)$ space needed to store $x$ and $y$. 
Think: Can this trick be used to reduce the space in Problem 2?