Review: Memoization and Dynamic Programming

1 Memoization

Fibonacci

A — some really big array, initialized to `null`.

function F(n)
  if $n = 0$ **then**
    return 0
  end if
  if $n = 1$ **then**
    return 1
  end if
  ~~**return** $f(n-1) + f(n-2)$~~
  if $Arr[n] \neq$ `null` **then**
    return $Arr[n]$
  end if
  $Arr[n] \leftarrow f(n-1) + f(n-2)$
  return $Arr[n]$
end function

Without memoization:

$$\begin{aligned} T(n) &= T(n-2) + T(n-1) + \Theta(1) \\ &\geq 2\,T(n-2) + \Theta(1) \\ &= 2^{n/2} \cdot \Theta(1) \end{aligned}$$

∗ Still saving results like DP, but uses a *global* array.

ADVANTAGE of array on outside:

Don't have to rebuild the array every time the function is called,

but fundamentally the exact same as DP.

∗ Linear the first time it runs, but after that it is always constant.

∗ A DP program is always linear.

2 The Shuffle Problem

Given 3 strings

$$\begin{aligned} A &= a_1 \dots a_n \\ B &= b_1 \dots b_m \\ C &= c_1 \dots c_{n+m} \end{aligned}$$

Example. $A = \texttt{abc}$, $B = \texttt{def}$.

$C \in \{\, \texttt{abcdef}, \ \texttt{adbecf}, \ \texttt{abdcef} \,\}$ ↗ last char must be the last of either A or B

Recurrence

$\text{OPT}(i, j)$ = "Are the first $i + j$ chars of C a shuffle of the first i chars from A and the first j from B?"

$$\text{OPT}(i,j) = \begin{cases} \textsc{False} & \text{// if } a_i \neq c_{i+j} \neq b_j \\ \text{OPT}(i, j-1) & \text{// if } a_i \neq c_{i+j} = b_j \\ \text{OPT}(i-1, j) & \text{// if } a_i = c_{i+j} \neq b_j \\ \text{OPT}(i-1, j) \parallel \text{OPT}(i, j-1) & \text{// if } a_i = c_{i+j} = b_j \end{cases}$$

In the last case we *must try both*. For example, with $A = \texttt{ab}$, $B = \texttt{cb}$, $C = \texttt{abcb}$, only taking $\text{OPT}(i-1, j)$ would be wrong.

Base Cases and Table Filling

Base cases

$\text{OPT}(1, 0) = (a_1 == c_1)$
$\text{OPT}(0, 1) = (b_1 == c_1)$

$\text{OPT}(i, 0) = \text{OPT}(i-1, 0)$ && $a_i = c_i$
$\text{OPT}(0, j) = \text{OPT}(0, j-1)$ && $b_j = c_j$

Algorithm

Let A be an $N \times N$ array.

```
A[1,0] ← (a_1 == c_1)
for i = 2, ..., n do
    A[i,0] ← (a_i == c_i) && A[i-1,0]
end for
A[0,1] ← (b_1 == c_1)
for j = 2, ..., m do
    A[0,j] ← (b_j == c_j) && A[0,j-1]
end for
for i = 1, ..., n do
    for j = 1, ..., m do
        if a_i ≠ c_{i+j} ≠ b_j then
            A[i,j] ← False
        end if
    end for
end for
```

3 Backtracking

```
output ← [ ]
i ← n,  j ← m
if A[i,j] = False then
    print "Can't be done"
```

```
    exit
end if
while i > 0 and j > 0 do
    // Figure out where c_{i+j} came from
    if a_i = c_{i+j} ≠ b_j then
        add "add in a_i" to output and i ← i − 1
    else if a_i ≠ c_{i+j} = b_j then
        add "add in b_j" to output and j ← j − 1
    else   // a_i = c_{i+j} = b_j
        if A[i, j − 1] then
            add "add in b_j" to output and j ← j − 1
        else
            add "add in a_i" to output and i ← i − 1
        end if
    end if
end while
if i == 0 then
    add "add in first j chars of B" to output
else   // if j == 0
    add "add in first i chars of A" to output
end if
return output
```