**Memoization and Dynamic Programming**

**Reading:**  *CLRS* Ch. 15, *CLR* Ch. 16

---

### Standard Exponentiation

Below is a recursive function `Raise2` that computes $2^n$:

```python
Raise2(n)
    if n = 0 then
        return 1
    else
        return 2 * Raise2(n-1)
```

For any input, can draw an invocation tree in which each node is labeled by `Raise2(i)` for some `i`, and each `Raise(i-1)` is a child of `Raise(i)`. E.g., here is the invocation tree for `Raise2(3)`:

```
  Raise2(3) 8
    └── Raise2(2) 4
        ├── Raise2(1) 2
        │    └── Raise2(0) 1
```

In the above tree, each node has been annotated with the result computed for that node. What is the recurrence relation and solution for the running-time of `Raise2`?

---

### Slow Exponentiation

Suppose we had a machine that didn’t have a multiply operator. Then we might write:

```python
Raise2-Slow(n)
    if n = 0 then
        return 1
    else
        return Raise2-Slow(n-1) + Raise2-Slow(n-1)
```

Here is an invocation tree for `Raise2-Slow(3)`, where we use `RS(i)` to abbreviate `Raise2-Slow(i)`:

```
  RS(3) 8
    ├── RS(2) 4
    │    ├── RS(1) 2
    │    │    └── RS(0) 1
    │    └── RS(1) 2
    │        └── RS(0) 1
    └── RS(2) 4
        ├── RS(1) 2
        │    ├── RS(0) 1
        │    └── RS(0) 1
        └── RS(1) 2
            └── RS(0) 1
```

What is the recurrence relation and solution for the running-time of `Raise2-Slow`?
Invocation Trees vs. DAGs

The reason $\text{Raise2-Slow}$ is so slow is that it re-solves the same subproblems many times. We can make it fast again by remembering the result of a subproblem once it is solved. This effectively "glues" together nodes with the same label in the function call tree to form a DAG (Directed Acyclic Graph).

In the case of $\text{Raise2-Slow}$, we can effectively turn the invocation tree into an invocation DAG by remembering the subproblem result in a local variable:

```plaintext
\text{Raise2-Fast}(n)
  \text{if } n = 0 \text{ then}
    \text{return } 1
  \text{else}
    \text{subresult } \leftarrow \text{Raise2-Fast}(n-1)
    \text{return subresult + subresult}
```

This function runs in time linear in $n$. Indeed, replacing $\text{subresult + subresult}$ by $2\cdot\text{subresult}$ and inlining $\text{Raise2-Fast}(n-1)$ for $\text{subresult}$ yields the original $\text{Raise2}$. 
Memoization

As we shall see, it is not always straightforward to rewrite a recursive function so that it removes subproblem duplication and produces an invocation DAG rather than an invocation tree.

A more general technique for creating invocation DAGs that *always* works is to remember the results of a function in a table indexed by the argument(s) of the function. This technique is called **memoization** (*not* memorization!)

Here’s a memoized version of `Raise2`:

```
Raise2-Fast2(n)
▷ Create a memoization table `T` indexed by the single argument of `Raise2`.
    T ← new array[0..n]
    for i ← 0 to n do
        T[i] ← 0 ▷ 0 is an "illegal" result indicating an empty slot
    return Raise2-Memo(T,n)

Raise2-Memo(T,n)
    if T[n] = 0 then
        ▷ Calculate result first time and remember it in `T`
        if n = 0 then
            T[n] ← 1
        else
            T[n] ← Raise2-Memo(T,n-1) + Raise2-Memo(T,n-1)
        ▷ Look up previously computed result in `T`
    return T[n]
```

Below, show how the computation `Raise2-Memo(T,3)` proceeds, showing both changes to the table `T` and the invocation tree for `Raise2-Memo`:

<table>
<thead>
<tr>
<th>n</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Slow Recursive Fibonacci

Using memoization for \texttt{Raise2} seems like overkill, especially when there is a simpler way to achieve the same result without a memoization table (i.e., using a local variable). An example where memoization is a clearer "win" (though still not strictly necessary) is calculating Fibonacci numbers. Here is the naive recursive function for calculating these:

\begin{verbatim}
Fib-Slow(n)
    if n \leq 1 then
        return n
    else
        return Fib-Slow(n-1) + Fib-Slow(n-2)
\end{verbatim}

Here is an invocation tree for \texttt{Fib-Slow(5)}, where we use \texttt{FS(i)} to abbreviate \texttt{Fib-Slow(i)}:

What is the running time for \texttt{Fib-Slow}?

What we’d really like is an invocation DAG for calculating Fibonacci numbers:
Memoized Recursive Fibonacci

Here’s the result of applying the memoization strategy to Fib-Slow:

\[
\text{Fib-Fast}(n) \\
\begin{align*}
\text{T} &\leftarrow \text{new array[0..n]} \\
\text{for } i &\leftarrow 0 \text{ to } n \text{ do} \\
\text{T}[i] &\leftarrow -1 \triangleright -1 \text{ is an ”illegal” result indicating an empty slot} \\
\text{return } \text{Fib-Memo(T,n)}
\end{align*}
\]

\[
\text{Fib-Memo(T,n)} \\
\begin{align*}
\text{if } T[n] &= -1 \text{ then} \\
\triangleright &\; \text{Calculate result first time and remember it in } T \\
\text{if } n &\leq 1 \text{ then} \\
\text{T}[n] &\leftarrow n \\
\text{else} \\
\text{T}[n] &\leftarrow \text{Fib-Memo(T,n-1)} + \text{Fib-Memo(T,n-2)} \\
\triangleright &\; \text{Look up previously computed result in } T \\
\text{return } T[n]
\end{align*}
\]

Below, show how the computation \text{Fib-Memo(T,5)} proceeds, showing both changes to the table \text{T} and the invocation tree for \text{Fib-Memo}:

<table>
<thead>
<tr>
<th>n</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Tupled Fibonacci

Even in the Fibonacci case, a table is unnecessary for making the recursive computation efficient. The recursive function can be made efficient by a technique known as **tupling**, in which a function returns a tuple containing not only its own result but also the results of some of its subproblems.

\[
\text{Fib-Fast2}(n) \quad \begin{align*}
&\text{let } (\text{ans}_n, \text{ans}_{n-1}) \leftarrow \text{Fib-Tuple}(n) \\
&\text{in return } \text{ans}_n
\end{align*}
\]

\[
\text{Fib-Tuple}(n) \quad \begin{align*}
&\text{if } n \leq 1 \text{ then} \\
&\quad \text{return } (n,0) \\
&\text{else} \\
&\quad \text{let } (\text{ans}_{n-1}, \text{ans}_{n-2}) \leftarrow \text{Fib-Tuple}(n-1) \\
&\quad \text{in return } (\text{ans}_{n-1} + \text{ans}_{n-2}, \text{ans}_{n-1})
\end{align*}
\]

Here is the invocation tree for \(\text{Fib-Tuple}(5)\):

```
Fib-Tuple(5) (5,3)
  Fib-Tuple(4) (3,2)
    Fib-Tuple(3) (2,1)
      Fib-Tuple(2) (1,1)
        Fib-Tuple(1) (1,0)
```

Note that the tuples returned in the above invocation tree are the state variables for an iterative calculation of Fibonacci numbers. So \(\text{Fib-Tuple}\) performs in bottom-up fashion the “same” iteration performed top-down by a typical tail-recursive calculation of Fibonacci numbers.
Dynamic Programming

The key aspect of the memoized \texttt{Raise2} and \texttt{Fib} function is that they use a table to avoid recomputation. Rather than keeping the recursive structure of the naive \texttt{Raise2} or \texttt{Fib}, we can instead organize the process around filling in the slots of the table. The technique of organizing a computation around filling in a table that avoids recomputation is called \textit{dynamic programming}.

Here are the dynamic programming solutions for \texttt{Raise2} and \texttt{Fib}:

\begin{verbatim}
Raise2-DP(n)
    T ← new array[0..n]
    ▷ By dependencies of Raise2, fill in slots from low to high.
    T[0] = 1
    for i <- 1 to n do
        T[i] ← 2*T[i-1]
    return T[n]

Fib-DP(n)
    T ← new array[0..n]
    ▷ By dependencies of Fibonacci, fill in slots from low to high.
    T[0] = 0
    T[1] = 1
    for i <- 2 to n do
        T[i] ← T[i-1] + T[i-2]
    return T[n]
\end{verbatim}

Show how the computations of \texttt{Raise2-DP(3)} and \texttt{Fib-DP(5)} proceed in the following tables:

\begin{tabular}{|c|c|}
\hline
n & \texttt{Raise2} \\
\hline
0 & \hline
1 & \hline
2 & \hline
3 & \hline
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
n & \texttt{Fib} \\
\hline
0 & \hline
1 & \hline
2 & \hline
3 & \hline
4 & \hline
5 & \hline
\hline
\end{tabular}

Note that in dynamic programming, it is not necessary to first initialize each slot of the table with a distinguished value. Rather, each table slot is filled exactly once in an order determined by the dependencies of the computation.

In the above cases, it’s not necessary to use a table whose size is linear in \( n \) – we can get by with a constant number of variable slots.

- How many variables are needed for \texttt{Raise2-DP}?
- How many variables are needed for \texttt{Fib-DP}?
Pascal’s Triangle

Pascal’s triangle is a more compelling example for memoization/dynamic programming. Recall Pascal’s triangle: each element is the sum of the two elements above it, except for edge elements, which are 1:

```
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
1  6  15  20  15  6  1
...```

For our purposes, it will help to “rotate” the triangle so it appears as a diagonal corner of a square:

```
1  1  1  1  1  1  
1  2  3  4  5  6  
1  3  6 10 15  
1  4 10 20  
1  5 15  
1  6  
1  
```

The following function computes the element in the $r$th row and $c$th column of the rotated Pascal’s triangle ($r$ and $c$ are 1-based):

```
Pascal(r, c)
if r = 1 or c = 1 then
    return 1
else
    return Pascal(r, c-1) + Pascal(r-1, c)
```

Below is an invocation tree for $Pascal(4, 3)$ in which each call $Pascal(r, c)$ has been abbreviated $P(r, c)$:

```
What is the worst case running time of $Pascal(r, c)$?
Memoized Pascal

The exponential running time of Pascal is due to subproblem duplication. The sharing implied by the invocation DAG is not expressible by tricks like local variables and tupling, but can be expressed by table-based memoization:

```plaintext
Pascal-Fast-Memo(r,c)
  T ← new array[1..r,1..c] ▷ Two parameters implies 2D table
  for i ← 0 to r do
    for j ← 0 to c do
      T[i,j] <- 0 ▷ 0 is an "illegal" result indicating an empty slot
  return Pascal-Memo(T,r,c)
```

```plaintext
Pascal-Memo(T,r,c)
  if T[r,c] = 0 then
    if r = 1 or c = 1 then
      T[r,c] ← 1
    else
      T[r,c] ← Pascal-Memo(T,r,c-1) + Pascal-Memo(T,r-1,c)
  return T[r,c]
```

An element is stored into each array element at most twice: once during initialization and at most once during Pascal-Memo. The running time of Fast-Pascal(r,c) is therefore \( \Theta(r \cdot c) \).

Below, show how the computation Pascal-Memo(T,4,3) proceeds, showing both changes to the table T and the invocation tree for Pascal-Memo:

<table>
<thead>
<tr>
<th>T</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Programming Pascal

We can also construct a dynamic programming version of Pascal that avoids the recursive control structure of Pascal altogether and instead fills in the 2D table according to the data dependencies implied by Pascal:

Pascal-Fast-DP(r,c)
T ← new array[1..r,1..c]
▷ Fill in edges of table with 1s:
for i ← 1 to r do
    T[i,1] ← 1
for j ← 2 to c do
    T[1,j] ← 1
▷ Fill in rest of table:
for i ← 2 to r do
    for j ← 2 to c do
        T[i,j] ← T[i,j-1] + T[i-1,j]
return T[r,c]

Below, show how the computation Pascal-Fast-DP(4,3) fills in the table T:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Longest Common Subsequence**

Suppose that we want to find the longest common subsequence (LCS) that is shared among two sequences. E.g the longest common subsequence of \([B, A, C, B]\) and \([C, A, B, A, B]\) is \([B, A, B]\).

Often the LCS is not unique. E.g. for \([B, A, C, B]\) and \([A, B, C, A, B]\), all of \([A, C, B]\), \([B, A, B]\), and \([B, C, B]\) are LCSs.

Here is a simple Haskell\textit{eager} algorithm computing the LCS of two linked lists:

\[
\begin{align*}
lcs([], \_ ) &= [] \\
lcs(\_, []) &= [] \\
lcs(x:xs, y:ys) &= \\
& \quad | x == y = x : (lcs(xs, ys)) \\
& \quad | \text{otherwise} = \\
& \quad \quad \text{let } l1 = lcs(xs, y:ys) \\
& \quad \quad \quad l2 = lcs(x:xs, ys) \\
& \quad \quad \quad \text{in if } (\text{length } l1) \geq (\text{length } l2) \text{ then } l1 \text{ else } l2
\end{align*}
\]

In cases where there is more than one solution, the above \texttt{lcs} algorithm will “choose” one of the solutions (since it gives precedence to \texttt{l1} when the lengths of \texttt{l1} and \texttt{l2} are the same).

Below is an invocation tree for \texttt{lcs([B, A, C, B],[A, B, C, A, B])}\textsuperscript{1}. To save space in the diagram, lists of characters (e.g., \([B,A,C,B]\)) have been abbreviated by the string of their elements (e.g., BACB). Note the duplication in some subproblems.

What is the worst-case running time of \texttt{lcs} for two strings of length \(m\) and \(n\)?

---

\textsuperscript{1}In actual Haskell, the characters would need to be delimited by single quotes, as in \texttt{lcs(['B', 'A', 'C', 'B'], ['A', 'B', 'C', 'A', 'B'])}. In Haskell, strings are just lists of characters, so this invocation could also be written as \texttt{lcs("BACB", "ABCAB").}
A More General LCS

Before optimizing lcs, it helps to generalize it:

\[
\begin{align*}
lcs([]=[],\_)&=\emptyset \\
lcs(\_,[])=\emptyset \\
lcs(x:xs,y:ys) &\quad |\ x==y=\otimes(x,\ lcs(xs,ys)) \\
 &\quad |\ otherwise=\oplus(lcs(xs,y:ys),\ lcs(x:xs,ys)) \\
\end{align*}
\]

We can get the effect of the previous solution (which “chooses” a single result list) as follows:

\[
\begin{align*}
\emptyset &= \[] \\
\otimes(x,zs) &= x:zs \\
\oplus(xs,ys) &\quad |\ length(xs) >= length(ys) = xs \\
 &\quad |\ otherwise = ys \\
\end{align*}
\]

Alternatively, we can return a list of all LCS lists via the following definitions:

\[
\begin{align*}
\emptyset &= \[[\]] \\
\otimes(x,zss) &= map (x:) zss -- prepend x to every list in zss \\
\oplus(xs:xss,ys:yss) &\quad -- arguments are non-empty lists of lists, all of same length \\
 &\quad |\ length(xs) > length(ys) = xs:xss \\
 &\quad |\ length(xs) < length(ys) = ys:yss \\
 &\quad |\ otherwise = (xs:xss)+(ys:yss) \\
\end{align*}
\]
Optimized LCS

Can perform LCS more efficiently by using a 2D table indexed by the nodes in the two argument lists. Here is what the table would look like for the arguments [B, A, C, B] and [A, B, C, A, B]:

<table>
<thead>
<tr>
<th></th>
<th>[A, B, C, A, B]</th>
<th>[B, C, A, B]</th>
<th>[C, A, B]</th>
<th>[A, B]</th>
<th>[B]</th>
<th>[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[B, A, C, B]</td>
<td>⊕ B</td>
<td>⊗ B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[A, C, B]</td>
<td>⊗ A</td>
<td>⊕ A</td>
<td>⊗ A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[C, B]</td>
<td>⊕ C</td>
<td>⊗ C</td>
<td>⊗ C</td>
<td>⊗ C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[B]</td>
<td>⊗ B</td>
<td>⊕ B</td>
<td>⊗ B</td>
<td>⊗ B</td>
<td>⊗ B</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>⊗</td>
<td>⊕</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
<td></td>
</tr>
</tbody>
</table>

- The slots of the table are filled in with values determined by ∅, ⊗, and ⊕.
- The slots of the table can be filled in either recursively (memoization, as shown above) or iteratively (dynamic programming).
  - Memoization has the overhead of recursion, but computes fewer entries in the table than dynamic programming.
  - Dynamic programming avoids the overhead of recursion, but computes more entries than it has to.