

Memoization and Dynamic Programming

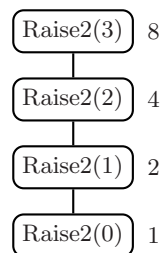
Reading: *CLRS* Ch. 15, *CLR* Ch. 16

Standard Exponentiation

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

```
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 `Raise2(i-1)` is a child of `Raise2(i)`. E.g., here is the invocation tree for `Raise2(3)`:



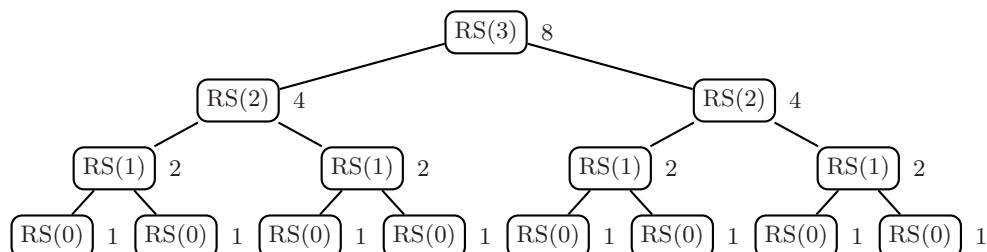
In the above tree, each node has been annotated with the result computed for that node. What is the recurrence relation and solution for the running-time of `Raise2`?

Slow Exponentiation

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

```
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)`:



What is the recurrence relation and solution for the running-time of `Raise2-Slow`?

Invocation Trees vs. DAGs

The reason `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 `Raise2-Slow`, we can effectively turn the invocation tree into an invocation DAG by remembering the subproblem result in a local variable:

```
Raise2-Fast(n)
  if n = 0 then
    return 1
  else
    subresult ← Raise2-Fast(n-1)
    return subresult + subresult
```

This function runs in time linear in n . Indeed, replacing `subresult + subresult` by `2*subresult` and inlining `Raise2-Fast(n-1)` for `subresult` yields the original `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`:

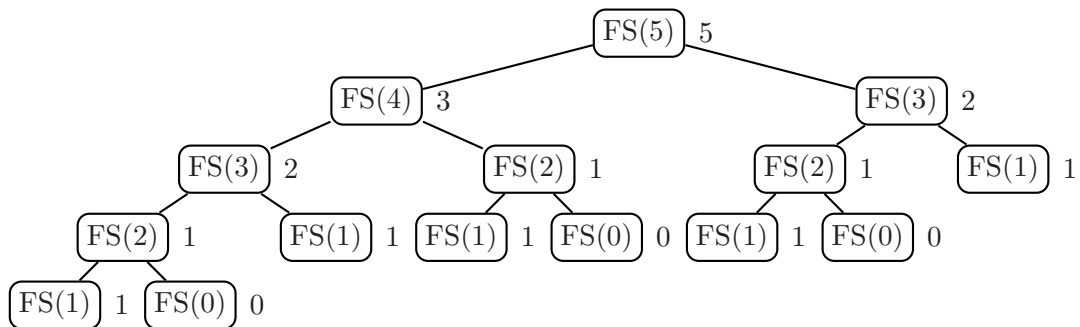
n	result
0	
1	
2	
3	

Slow Recursive Fibonacci

Using memoization for `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:

```
Fib-Slow(n)
  if n ≤ 1 then
    return n
  else
    return Fib-Slow(n-1) + Fib-Slow(n-2)
```

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



What is the running time for `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:

```
Fib-Fast(n)
  T ← new array[0..n]
  for i ← 0 to n do
    T[i] ← -1 ▷ -1 is an "illegal" result indicating an empty slot
  return Fib-Memo(T,n)
```

```
Fib-Memo(T,n)
  if T[n] = -1 then
    ▷ Calculate result first time and remember it in T
    if n ≤ 1 then
      T[n] ← n
    else
      T[n] ← Fib-Memo(T,n-1) + Fib-Memo(T,n-2)
    ▷ Look up previously computed result in T
  return T[n]
```

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

n	result
0	
1	
2	
3	
4	
5	

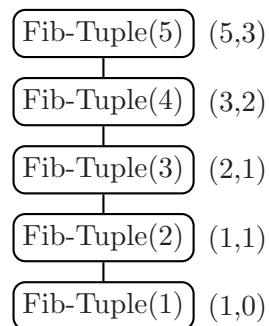
Tupled Fibonacci

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.

```
Fib-Fast2(n)
  let (ansn, ansn-1) ← Fib-Tuple(n)
  in return ansn

Fib-Tuple(n)
  if n ≤ 1 then
    return (n,0)
  else
    let (ansn-1, ansn-2) ← Fib-Tuple(n-1)
    in return (ansn-1 + ansn-2, ansn-1)
```

Here is the invocation tree for `Fib-Tuple(5)`



Note that the tuples returned in the above invocation tree are the state variables for an iterative calculation of Fibonacci numbers. So `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 `Raise2` and `Fib` function is that they use a table to avoid recomputation. Rather than keeping the recursive structure of the naive `Raise2` or `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 **dynamic programming**.

Here are the dynamic programming solutions for `Raise2` and `Fib`:

`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]
```

Show how the computations of `Raise2-DP(3)` and `Fib-DP(5)` proceed in the following tables:

n	Raise2
0	
1	
2	
3	

n	Fib
0	
1	
2	
3	
4	
5	

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 `Raise2-DP`?
- How many variables are needed for `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 ...
1 2 3 4 5 6 ...
1 3 6 10 15 ...
1 4 10 20 ...
1 5 15 ...
1 6 ...
1 ...

```

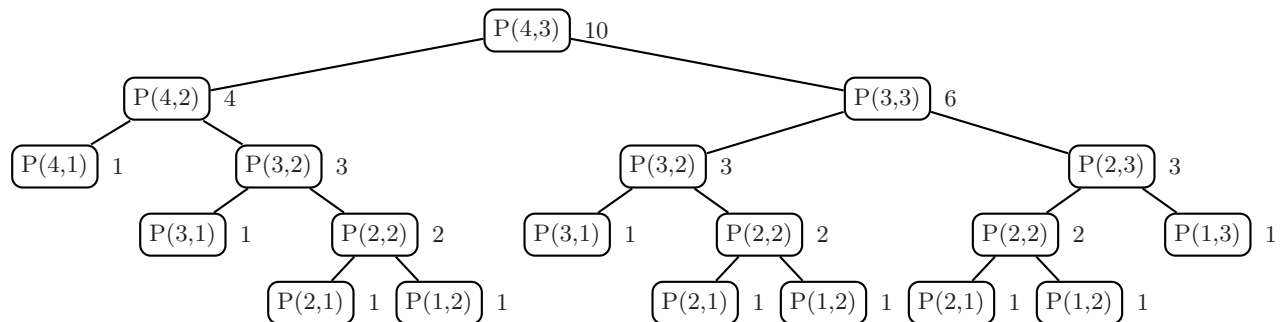
The following function computes the element in the r th row and c th column of the rotated Pascal's triangle (r and c are 1-based):

```

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 $\text{Pascal}(4, 3)$ in which each call $\text{Pascal}(r, c)$ has been abbreviated $P(r, c)$:



What is the worst case running time of $\text{Pascal}(r, c)$?

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:

```
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)
```

```
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`:

T	1	2	3
1			
2			
3			
4			

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`:

T	1	2	3
1			
2			
3			
4			

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_{eager} algorithm computing the LCS of two linked lists:

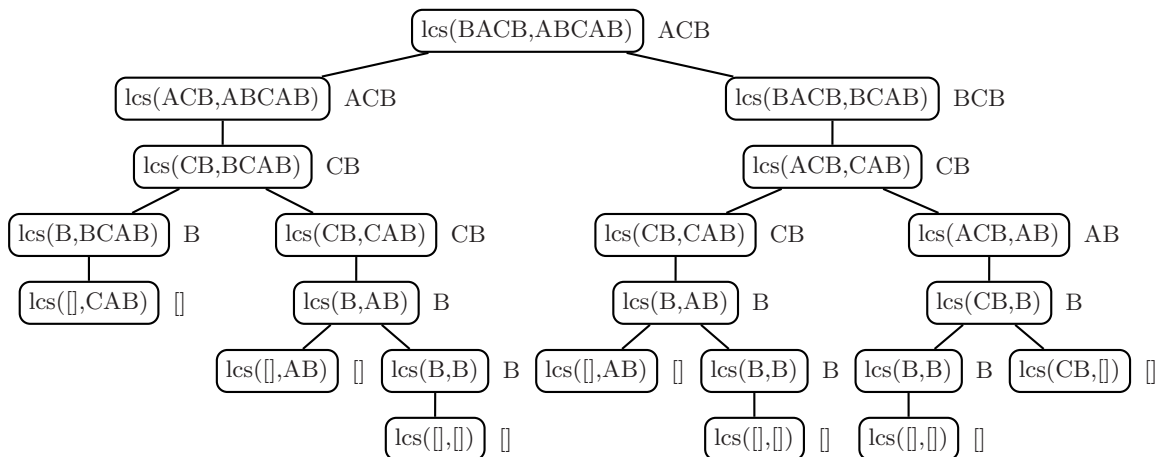
```

lcs([],_) = []
lcs(_,[]) = []
lcs(x:xs,y:ys)
  | x == y = x:(lcs(xs,ys))
  | otherwise =
    let l1 = lcs(xs,y:ys)
        l2 = lcs(x:xs,ys)
    in if (length l1) >= (length l2) then l1 else l2

```

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

Below is an invocation tree for `lcs([B, A, C, B], [A, B, C, A, B])`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 `lcs` for two strings of length m and n ?

¹In actual Haskell, the characters would need to be delimited by single quotes, as in `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 `lcs("BACB", "ABCAB")`.

A More General LCS

Before optimizing `lcs`, it helps to generalize it:

```
lcs([],_) = []
lcs(_,[]) = []
lcs(x:xs,y:ys)
  | x == y = ⊗(x, lcs(xs,ys))
  | otherwise = ⊕(lcs(xs,y:ys), lcs(x:xs,ys))
```

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

```
∅ = []
⊗(x,zs) = x:zs
⊕(xs,ys)
  | length(xs) >= length(ys) = xs
  | otherwise = ys
```

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

```
∅ = [[]]
⊗(x,zss) = map (x:) zss -- prepend x to every list in zss
⊕(xs:xss,ys:yss) -- arguments are non-empty lists of lists, all of same length
  | length(xs) > length(ys) = xs:xss
  | length(xs) < length(ys) = ys:yss
  | otherwise = (xs:xss)++(ys:yss)
```

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]:

	[A,B,C,A,B]	[B,C,A,B]	[C,A,B]	[A,B]	[B]	[]
[B,A,C,B]	\oplus	\otimes -B				
[A,C,B]	\otimes -A		\oplus	\otimes -A		
[C,B]		\oplus	\otimes -C		\oplus	\emptyset
[B]		\otimes -B		\oplus	\otimes -B	
[]			\emptyset	\emptyset		\emptyset

- The slots of the table are filled in with values determined by \emptyset , \otimes , and \oplus .
- 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.