

**MEMOIZATION AND DYNAMIC PROGRAMMING****Reading:** CLR Chapter 16Below 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 a function call tree in which each node is labelled by Raise2(i) for some i, and each Raise2(i-1) is a child of Raise2(i). E.g. Draw Raise2(3):

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

```

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

```

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

Draw Raise2-Slow(3):

The reason Raise2-Slow is so slow is that it resolves 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 this case, we can remember the subproblem result in a local variable:

```

Raise2-Fast1(n)
  if n = 0 then
    return 1
  else
    subresult <- Raise2(n-1)
    return subresult + subresult

```

More generally, we need an auxiliary table to remember results.

```

Raise2-Fast2(n)
  R <- new array[0..n]
  for i <- 0 to n do
    R[i] <- 0
  return Raise2-Memo(R,n)

Raise2-Memo(R,n)
  if R[n] = 0 then
    if n = 0 then
      return 1
    else
      R[n] <- Raise2-Memo(R,n-1) + Raise2-Memo(R,n-1)
  return R[n]

```

This strategy of remembering the results of subproblems in a table is called **memoization** (not memorization!).

### Pascal's Triangle

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

```

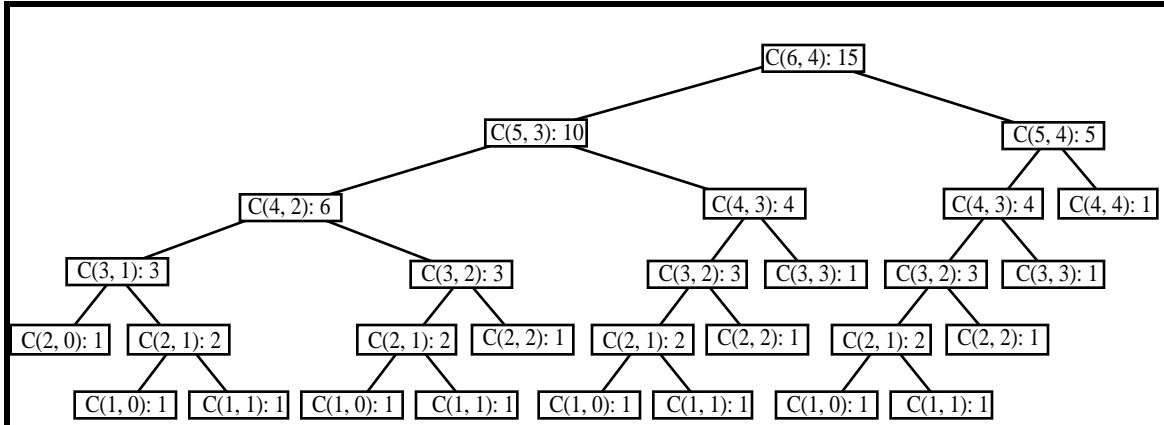
The following Pascal function computes the kth element of the nth row (where both k and n are 0-based):

```

Pascal(n,k)
  if k = 0 or k = n then
    return 1
  else
    return Pascal(n-1,k-1) + Pascal(n-1, k)

```

Below is a function call tree for Pascal(6, 4) in which each call Pascal(n,k) has been abbreviated C(n,k).



In the worst case,  $C(n,k)$  can take time exponential in  $n$ . Note that the exponential nature is due to subproblem duplication, which can be removed by memoization. We use a two-dimensional table  $P(i,j)$  to store previously computed results:

```

Fast-Pascal(n,k)
  P <- new array[0..n][0..n]
  for i <- 1 to n do
    for j <- 1 to n do
      P[i,j] <- 0
  Pascal-Memo(P, n, k)

Pascal-Memo(P,n,k)
  if P(n,k) = 0 then
    if k = 0 or k = n then
      P(n,k) <- 1
    else
      P(n,k) <- Pascal-Memo(P,n-1,k-1) + Pascal-Memo(P,n-1,k)
  return P(n,k)

```

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 is therefore  $\Theta(n^2)$ .

We can be cleverer by initializing the top and left edge to 1 to avoid the inner if within Pascal-Memo.

## Longest Common Subsequence