Analysis & Design of Algorithms

Angel Napa

May 30, 2024

1 Fibonacci Numbers

Consider the recurrence:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

The numbers $F(0), F(1), F(2), \ldots, F(n)$ are called *Fibonacci Numbers.* $(0, 1, 1, 2, 3, 5, 8, 13, 21, 42, ...)$

Problem: given an integer n, compute $F(n)$. Consider the following recursive algorithm:

Input: A number n
Output: $F(n)$
RECURSIVE-FIBONACCI(n)

```
1: if n == 0
2:    return 0
3: if n == 1
4:    return 1
5: return RECURSIVE-FIBONACCI(n − 1)+ RECURSIVE-FIBONACCI(n − 2)
```

It is clear that the running time of the algorithm (assuming all constants are 1) is given by

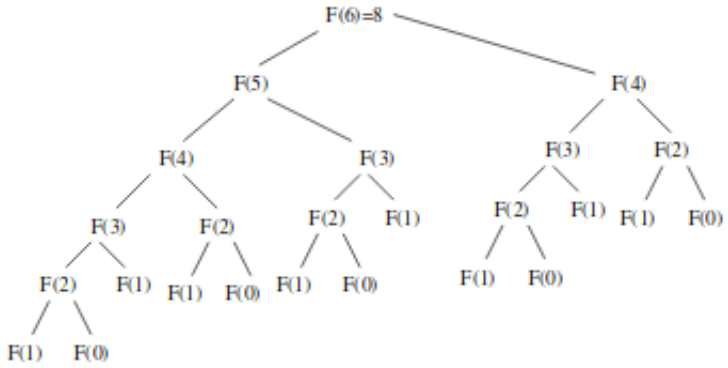
$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + 1 & \text{if } n > 1 \end{cases}$$

We will prove by induction that $T(n) \geq \frac{2}{3}(3/2)^n$ for $n \geq 1$. When $n = 0$ we have that $T(0) = 1 \geq \frac{2}{3}(3/2)^0 = 2/3$. When $n = 1$ we have that $T(1) = 1 \geq$

$\frac{2}{3}(3/2)^1 = 1$. Now suppose $n > 1$. Then,

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq \frac{2}{3}(3/2)^{n-1} + \frac{2}{3}(3/2)^{n-2} + 2 \\ &= \frac{2}{3}(3/2)^n(\frac{2}{3} + \frac{4}{9}) + 2 \\ &\geq \frac{2}{3}(3/2)^n \end{aligned}$$

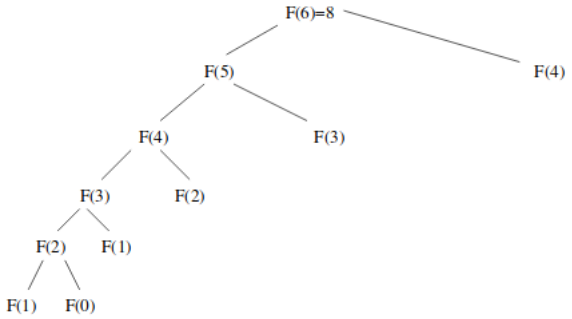
Therefore, the running time of the RECURSIVE-FIBONACCI algorithm is $\Omega((\frac{3}{2})^n)$.



Consider the following improvement.

Input: A number n
Output: $F(n)$
MEMOIZED-FIBONACCI(n)
1: **if** $M[n] \neq -1$
2: **return** $M[n]$
3: **else**
4: $M[n] =$ MEMOIZED-FIBONACCI($n-1$)+MEMOIZED-FIBONACCI($n-2$)
5: **return** $M[n]$
MAIN(n)
1: $M[0] = 0$
2: $M[1] = 1$
3: **for** $i = 2$ **to** n
4: $M[i] = -1$
5: **return** MEMOIZED-FIBONACCI(n)

The previous algorithm is a *memoized* version of the recursive version. What we are doing is storing results that have already been computed in memory.



Let $T(n)$ be the running time of the MEMOIZED-FIBONACCI routine. Note that the running time of the recursive call with parameter $n-2$ will always be constant, since $M[n-2]$ will have been previously computed in the call with parameter $n-1$. Therefore,

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1)+1 & \text{if } n > 1 \end{cases}$$

which implies that $T(n) = \Theta(n)$ (another way to arrive at this result is to observe that line 4 of the algorithm is executed at most n times). It is clear that a bit more memory is used compared to the previous case; however, the memory used is also linear in n. We conclude that the efficiency of MEMOIZED-FIBONACCI is much greater than the efficiency of RECURSIVE-FIBONACCI.

We can further improve MEMOIZED-FIBONACCI by eliminating the recursive calls while maintaining the execution time.

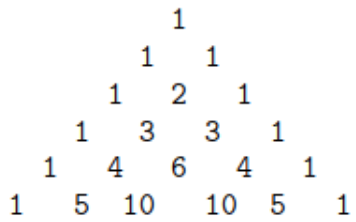
Input: A number n
Output: $F(n)$
DYN-PROG-FIBONACCI(n)
1: $M[0] = 0$
2: $M[1] = 1$
3: **for** $i = 2$ **to** n
4: $\quad M[i] = M[i-1] + M[i-2]$
5: **return** $M[n]$

The previous technique is called *dynamic programming*.

2 Binomial Coefficients

For every pair of natural numbers n, k with $n \geq k$, consider the following formula: $C(n,k) = \binom{n}{k} = \frac{n!}{(n-k)!k!}$. (This formula counts the number of ways to choose k elements from n possible ones).

If we want to compute the function $C(n,k)$, we can take advantage of a property given by Pascal's triangle:



Note that $C(n,k) = C(n-1,k-1)+C(n-1,k)$. Why is this true? We want to choose k items out of n. If item number n is not chosen, then the answer is $C(n-1,k)$. If item number n is chosen, then we must choose $k-1$ items out of the remaining $n-1$, and the answer is $C(n-1,k-1)$.

Therefore, we have the recurrence:

$$C(n,k) = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ C(n-1,k-1) + C(n-1,k) & \text{otherwise} \end{cases}$$

Consider the following recursive algorithm.

Input: Two numbers n and k
Output: $C(n,k)$
RECURSIVE-BINOMIAL(n,k)
1: **if** $k == 0$ **or** $k == n$
2: **return** 1
3: **return** RECURSIVE-BINOMIAL$(n-1,k-1)$ + RECURSIVE-BINOMIAL$(n-1,k)$

Similarly to the recursive version of Fibonacci, the running time of the algorithm is exponential. We can avoid redundant calculations by storing already computed values in memory, obtaining the following memoized version.

Input: Two numbers n and k
Output: $C(n,k)$
MEMOIZED-BINOMIAL(n,k)
1: **if** $M[n][k] \neq -1$
2: **return** $M[n][k]$
3: **if** $k == 0$ **or** $k == n$
4: **return** 1
5: $M[n][k]$ = MEMOIZED-BINOMIAL$(n-1,k-1)$ + MEMOIZED-BINOMIAL$(n-1,k)$
6: **return** $M[n][k]$

MAIN(n,k)
1: **for** $i = 0$ **to** n
2: **for** $j = 0$ **to** k
3: $M[i][j] = -1$
4: **return** MEMOIZED-BINOMIAL(n,k)

Finally, we can eliminate the recursion as follows, using dynamic programming.

Input: Two numbers n and k
Output: $C(n, k)$
DYN-PROG-BINOMIAL(n, k)
1: **for** $i = 0$ **to** n
2: **for** $j = 0$ **to** k
3: **if** $j == 0$ **or** $j == i$
4: $M[i][j] = 1$
5: **else**
6: $M[i][j] = M[i-1][j-1] + M[i-1][j]$
7: **return** $M[n][k]$

3 Integer Partition

Given a positive integer n, we want to determine the number of ways in which n can be written as a sum of positive integers. We denote this number by $p(n)$. For example, $p(4) = 5$ because:

$$\begin{aligned} 4 &= 1+1+1+1 \\ &= 1+1+2 \\ &= 1+3 \\ &= 2+2 \\ &= 4 \end{aligned}$$

We can obtain the recurrence $p(n) = p(n-1)+p(n-2)+p(n-3)+\cdots+p(1)+1$. We add 1 to take into account the sum of n itself. We can construct the following recursive algorithm.

Input: A number n
Output: $p(n)$
RECURSIVE-PARTITION(n)
1: **if** $n == 0$
2: **return** 1
3: sum=0
4: **for** $i = 1$ **to** n
5: sum = sum + RECURSIVE-PARTITION$(n - i)$
6: **return** sum

Similarly to the recursive versions of Fibonacci and Binomial coefficients, the running time of the algorithm is exponential. We can avoid redundant calculations by storing already computed values in memory, obtaining the following memoized version.

Input: A number n
Output: $p(n)$

MEMOIZED-PARTITION(n)
1: **if** $M[n] \neq -1$
2: **return** $M[n]$
3: **if** $n == 0$
4: **return** 1
5: sum=0
6: **for** $i = 1$ **to** n
7: sum = sum + MEMOIZED-PARTITION($n - i$)
8: $M[n] = sum$
9: **return** sum

MAIN(n)
1: **for** $i = 0$ **to** n
2: $M[i] = -1$
3: **return** MEMOIZED-PARTITION(n)

Finally, we can eliminate the recursion as follows, using dynamic programming.

Input: A number n
Output: $p(n)$
DYN-PROG-PARTITION(n)
1: $M[0] = 1$
2: **for** $i = 1$ **to** n
3: sum=0
4: **for** $j = 1$ **to** i
5: sum = sum + $M[i - j]$
6: $M[i] = sum$
7: **return** $M[n]$

4 Longest Common Subsequence (LCS)

Given two sequences $X = (x_1, x_2, ..., x_n)$ and $Y = (y_1, y_2, ..., y_m)$, we want to find the longest sequence Z that is a subsequence of both X and Y. A *subsequence* of a sequence X is obtained by removing some or none of the elements of X without changing the order of the remaining elements. We denote by $L(i, j)$ the length of the LCS of the sequences $(x_1, x_2, ..., x_i)$ and $(y_1, y_2, ..., y_j)$. We can obtain the following recurrence:

$$L(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{L(i, j-1), L(i-1, j)\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Consider the following recursive algorithm.

Input: Two sequences X and Y
Output: $L(i, j)$
RECURSIVE-LCS(X, Y, i, j)

```
1: if i == 0 or j == 0
2:    return 0
3: if x_i == y_j
4:    return RECURSIVE-LCS(X, Y, i − 1, j − 1) + 1
5: else
6:    return max{RECURSIVE-LCS(X, Y, i, j − 1), RECURSIVE-LCS(X, Y, i − 1, j)}
```

Similarly to the recursive versions of Fibonacci, Binomial coefficients, and Integer Partition, the running time of the algorithm is exponential. We can avoid redundant calculations by storing already computed values in memory, obtaining the following memoized version.

Input: Two sequences X and Y
Output: $L(i, j)$

```
MEMOIZED-LCS(X, Y, i, j)
1: if M[i][j] ≠ −1
2:    return M[i][j]
3: if i == 0 or j == 0
4:    return 0
5: if x_i == y_j
6:    return MEMOIZED-LCS(X, Y, i − 1, j − 1) + 1
7: else
8:    return max{MEMOIZED-LCS(X, Y, i, j − 1), MEMOIZED-LCS(X, Y, i − 1, j)}
9: M[i][j] = MEMOIZED-LCS(X, Y, i, j)
10: return M[i][j]
```

```
MAIN(X, Y)
1: for i = 0 to n
2:    for j = 0 to m
3:       M[i][j] = −1
4: return MEMOIZED-LCS(X, Y, n, m)
```

Finally, we can eliminate the recursion as follows, using dynamic programming.

Input: Two sequences X and Y
Output: $L(i, j)$

```
DYN-PROG-LCS(X, Y)
1: for i = 0 to n
2:    for j = 0 to m
3:       if i == 0 or j == 0
4:          M[i][j] = 0
5:       else
6:          if x_i == y_j
7:             M[i][j] = M[i − 1][j − 1] + 1
8:          else
9:             M[i][j] = max(M[i][j − 1], M[i − 1][j])
10: return M[n][m]
```