Thus $P(i,j)$ is replaced by $C(i+j,j)$, $P(i-1,j)$ by $C(i+j-1,j)$, and $P(i,j-1)$ by $C(i+j-1,j-1)$. Now the pattern of calls shown by the arrows corresponds to the calculation

$$C(i+j,j) = C(i+j-1,j) + C(i+j-1,j-1)$$

of a binomial coefficient. The total number of recursive calls is therefore exactly $2\binom{i+j}{j} - 2$; see Problem 8.1. To calculate the probability $P(n,n)$ that team A will win given that the series has not yet started, the required time is thus in $\Omega\left(\binom{2n}{n}\right)$.

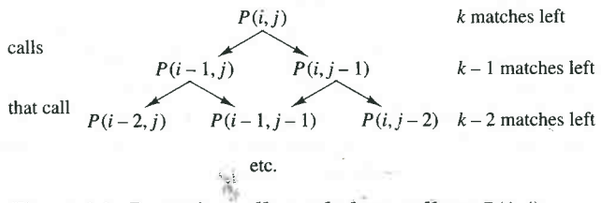

Figure 8.2. Recursive calls made by a call on $P(i,j)$

Problem 1.42 asks the reader to show that $\binom{2n}{n} \geq 4^n/(2n+1)$. Combining these results, we see that the time required to calculate $P(n,n)$ is in $O(4^n)$ and in $\Omega(4^n/n)$. The method is therefore not practical for large values of n. (Although sporting competitions with $n > 4$ are the exception, this problem does have other applications!)

To speed up the algorithm, we proceed more or less as with Pascal's triangle: we declare an array of the appropriate size and then fill in the entries. This time, however, instead of filling the array line by line, we work diagonal by diagonal. Here is the algorithm to calculate $P(n,n)$.

```
function series(n, p)
    array P[0..n, 0..n]
    q ← 1 − p
    {Fill from top left to main diagonal}
    for s ← 1 to n do
        P[0, s] ← 1; P[s, 0] ← 0
        for k ← 1 to s − 1 do
            P[k, s − k] ← pP[k − 1, s − k] + qP[k, s − k − 1]
    {Fill from below main diagonal to bottom right}
    for s ← 1 to n do
        for k ← 0 to n − s do
            P[s + k, n − k] ← pP[s + k − 1, n − k] + qP[s + k, n − k − 1]
    return P[n, n]
```

Since the algorithm has to fill an $n \times n$ array, and since a constant time is required to calculate each entry, its execution time is in $\Theta(n^2)$. As with Pascal's triangle, it is easy to implement this algorithm so that storage space in $\Theta(n)$ is sufficient.

8.2 Making change (2)

Recall that the problem is to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins. In Section 6.1 we described a greedy algorithm for this problem. Unfortunately, although the greedy algorithm is very efficient, it works only in a limited number of instances. With certain systems of coinage, or when coins of a particular denomination are missing or in short supply, the algorithm may either find a suboptimal answer, or not find an answer at all.

For example, suppose we live where there are coins for 1, 4 and 6 units. If we have to make change for 8 units, the greedy algorithm will propose doing so using one 6-unit coin and two 1-unit coins, for a total of three coins. However it is clearly possible to do better than this: we can give the customer his change using just two 4-unit coins. Although the greedy algorithm does not find this solution, it is easily obtained using dynamic programming.

As in the previous section, the crux of the method is to set up a table containing useful intermediate results that are then combined into the solution of the instance under consideration. Suppose the currency we are using has available coins of n different denominations. Let a coin of denomination i, $1 \leq i \leq n$, have value d_i units. We suppose, as is usual, that each $d_i > 0$. For the time being we shall also suppose that we have an unlimited supply of coins of each denomination. Finally suppose we have to give the customer coins worth N units, using as few coins as possible.

To solve this problem by dynamic programming, we set up a table $c[1..n, 0..N]$, with one row for each available denomination and one column for each amount from 0 units to N units. In this table $c[i,j]$ will be the minimum number of coins required to pay an amount of j units, $0 \leq j \leq N$, using only coins of denominations 1 to i, $1 \leq i \leq n$. The solution to the instance is therefore given by $c[n,N]$ if all we want to know is how many coins are needed. To fill in the table, note first that $c[i,0]$ is zero for every value of i. After this initialization, the table can be filled either row by row from left to right, or column by column from top to bottom. To pay an amount j using coins of denominations 1 to i, we have in general two choices. First, we may choose not to use any coins of denomination i, even though this is now permitted, in which case $c[i,j] = c[i-1,j]$. Alternatively, we may choose to use at least one coin of denomination i. In this case, once we have handed over the first coin of this denomination, there remains to be paid an amount of $j - d_i$ units. To pay this takes $c[i, j-d_i]$ coins, so $c[i,j] = 1 + c[i, j-d_i]$. Since we want to minimize the number of coins used, we choose whichever alternative is the better. In general therefore

$$c[i,j] = \min(c[i-1,j], 1 + c[i, j-d_i]).$$

When $i = 1$ one of the elements to be compared falls outside the table. The same is true when $j < d_i$. It is convenient to think of such elements as having the value $+\infty$. If $i = 1$ and $j < d_1$, then both elements to be compared fall outside the table. In this case we set $c[i,j]$ to $+\infty$ to indicate that it is impossible to pay an amount j using only coins of type 1.