Journal of Integer Sequences, Vol. 20 (2017),
Article 17.2.2

On Counting the Number of Tilings of a Rectangle with Squares of Size 1 and 2

Johan Nilsson
LIPN Université Paris 13
93430 Villetaneuse
France
nilsson@lipn.univ-paris13.fr

Abstract

We consider tilings of a rectangle of size $n \times k$ with square tiles of size 1×1 and 2×2. We present a method to calculate the number of such tilings via matrix multiplication, where we optimize the number of multiplications needed and reduce the space required for the matrix multiplication by dynamically generating the matrices involved.

1 Introduction

In this paper we study the problem of calculating the number of tilings of a rectangle R with square tiles of size 1×1 and 2×2. This tiling problem is easily seen to be equivalent to another classical problem, the problem of counting the number of ways of placing non-attacking kings on a rectangular chessboard. Our version and the chess formulation of the problem have been widely studied in different formulations and aspects; see [1, 2, 5], as well as the sequences A063443, A193580 and A245013 in the On-Line Encyclopedia of Integer Sequences OEIS [4].

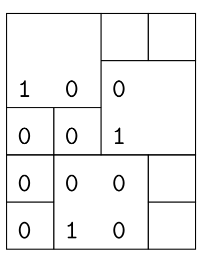
Here we discuss a method using matrix multiplication to calculate the number of tilings of R. The algorithm we introduce uses the idea of transforming the problem into a graph problem and then applies the corresponding transition matrix A_n for the calculation. Our way of approach is optimized in the number of multiplications needed to be done, with respect to the non-zero entries of the sparse matrix A_n, which is exponentially less than the size of A_n. The method we give here improves the idea by Race et al. [2], since we generate the matrix on the fly and thereby substantially reduce the space needed. By running an implementation of our algorithm we have extended the sequence A063443 with 15 new entries. A summary of our results can be found below.

Result 1. *We present an algorithm for calculating the number of tilings of an $n \times k$ rectangle R via matrix multiplication. The number of operations needed to generate the matrix A_n is linear in the number of non-zero entries in A_n and requires only $O(n)$ space. The space required to perform the actual multiplication is linear in the number of rows of A_n.*

The paper is organized as follows. In the next section, we give necessary definitions and formulate the ideas of our method to calculate the number of tilings in detail. Thereafter we present how to generate the adjacency matrix A_n for the matrix multiplication, and then how to perform the actual calculation.

2 Graphs and matrices

In order to count the number of ways to tile an $(n+1) \times (k+1)$ rectangle R with square tiles of size 1×1 and 2×2, we introduce a binary $k \times n$ matrix, $M_{k,n}$, to represent one such tiling. The ones in $M_{k,n}$ represent the lower left corner of a big tile; see Figure 1. The zeros represent the small tiles or work as space fillers in the 3 remaining parts of a big tile. We see a row in $M_{k,n}$ as a binary word. Our use of the ones implies that we cannot have two consecutive ones in a row. It suffices to consider the $k \times n$ matrix $M_{k,n}$ instead of a $(k+1) \times (n+1)$ matrix, since the rightmost column and the top row of the latter would just contain zeros.

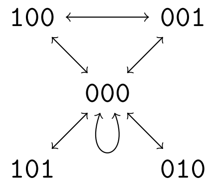


$$M_{4,3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

100
↑
001
↑
000
↑
010

Figure 1: A tiling of a 4×5 square and its representation by the $M_{4,3}$ matrix and by a sequence of words of length 3.

Further, a row in $M_{k,n}$ is dependent on its neighboring rows. Hence, we may view the rows in $M_{k,n}$ as nodes in a graph G_n, where we have an edge between two rows $r_1 = (r_{1,1}, r_{1,2}, \ldots, r_{1,n})$ and $r_2 = (r_{2,1}, r_{2,2}, \ldots, r_{2,n})$ if and only if r_1 can be placed next to r_2; see Figure 2 and compare Figure 1. The row r_1 can be placed next to r_2 if and only if $r_{2,j} = 0$ for $j \in \{i-1, i, i+1\} \cap \{1, \ldots, n\}$ for all i such that $r_{1,i} = 1$.



$$A_3 = \begin{array}{c} \\ \texttt{000} \\ \texttt{001} \\ \texttt{010} \\ \texttt{100} \\ \texttt{101} \end{array} \begin{array}{c} \begin{array}{ccccc} \texttt{000} & \texttt{001} & \texttt{010} & \texttt{100} & \texttt{101} \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 2: A graph and its corresponding adjacency matrix representing the ways to tile a $4 \times n$ rectangle R. A path of length $n-1$ in the graph corresponds to a tiling of R.

Having the graph G_n, it is easy to get its corresponding adjacency matrix A_n. We index the rows and the columns in A_n with the allowed words (or rows) that can occur in $M_{k,n}$ in lexicographical order. We denote the set of these allowed words by T_n, that is,

$$T_n = \left\{ t \in \{\texttt{0}, \texttt{1}\}^n : t_i t_{i+1} \neq \texttt{11} \text{ where } t = t_1 \cdots t_n \right\}.$$

It is easy to see (and a standard exercise to show) that

$$|T_n| = F_{n+2}, \tag{1}$$

where $|\cdot|$ denotes the cardinality of a set, and F_n is the nth Fibonacci number (that is, $F_n = F_{n-1} + F_{n+2}$ with $F_0 = 0$ and $F_1 = 1$; see A000045). By looking at the structure of the set T_n, we see that we can give a recursive definition of the adjacency matrix A_n, as previously noted in [1, 2]. We define

$$A_n = \begin{pmatrix} A_{n-1} & A_{n-2} \\ A_{n-2} & 0 \end{pmatrix}, \qquad A_0 = \begin{pmatrix} 1 \end{pmatrix}, \quad \text{and} \qquad A_1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \tag{2}$$

where we fill the missing space in A_n with zeros; see again the matrix in Figure 2. The number of rows in A_n is given by the size of T_n, that is,

$$\text{Rows}(A_n) = |T_n| = F_{n+2}, \tag{3}$$

where Rows gives the number of rows of a square matrix. From the recursion in (2), together with its initial conditions, we have that the number of ones in A_n satisfies the recursion

$$\text{Ones}(A_n) = \text{Ones}(A_{n-1}) + 2 \cdot \text{Ones}(A_{n-2}),$$

with $\text{Ones}(A_0) = 1$ and $\text{Ones}(A_1) = 3$, where the function Ones gives the number of ones in a matrix; see Table 1.

It is straightforward to find a closed expression for the number of ones in A_n. Standard techniques give

$$\text{Ones}(A_n) = \frac{4}{3}\, 2^n - \frac{1}{3}\, (-1)^n, \tag{4}$$

n	1	2	3	4	5	6	7	8
$\mathrm{Ones}(A_n)$	3	5	11	21	43	85	171	341

Table 1: The number of ones in A_n for small values of n. The numbers follow the Jacobsthal sequence; see A001045.

for $n \geq 0$. This shows that A_n is a sparse matrix, since its number of entries are $(F_{n+2})^2$, by (3).

The question of finding the number of tilings of an $n \times k$ rectangle now reduces to making a series of matrix multiplications. If we let $a(n,k)$ be the number of tilings of such a rectangle then

$$a(n,k) = \mathbf{1}^T A_{n-1}^{k-2} \mathbf{1}, \tag{5}$$

where $\mathbf{1}$ is a column vector of ones. This formula was also considered by Calkin et al. and Race et al. [1, 2]. Note that we have $a(n,k) = a(k,n)$. The number of matrix multiplications can be reduced by noticing the following, which holds since the adjacency matrix A_n is symmetric,

$$\mathbf{1}^T A_n^{\alpha+\beta} \mathbf{1} = \mathbf{1}^T A_n^{\alpha} \cdot A_n^{\beta} \mathbf{1} = (A_n^{\alpha} \mathbf{1})^T (A_n^{\beta} \mathbf{1}). \tag{6}$$

For the case when $\alpha + \beta = 2\gamma$ in the above expressions we have $\mathbf{1}^T A_n^{2\gamma} \mathbf{1} = |A_n^{\gamma} \mathbf{1}|^2$.

If we perform the matrix multiplications in (5) with iterations, that is,

$$v_{j+1} = A_{n-1} v_j, \quad j = 1, \ldots, k-2 \quad \text{with} \quad v_1 = \mathbf{1} \tag{7}$$

then the number of multiplications needed to be done is $O(2^n k)$, which is not symmetric in n and k. So, making several matrix multiplications with a small matrix is often less costly than few matrix multiplications with a large matrix. On the other hand, if we do the matrix multiplications in (5) via squaring (i.e. by calculating A_n^2, A_n^4, $\ldots$) then we need to make $O(\log k)$ matrix multiplications, resulting in $O\big((F_n)^3 \log k\big)$ multiplications. This occurs because the matrix A_n^2 is no longer sparse, it contains only non-zero entries. Also, the space needed is substantially larger, as we need to store the matrix A_n^j. In this paper we shall apply the method of iterated matrix multiplications.

Example 2. The number of ways to tile a 4×4 square is given by

$$a(4,4) = \mathbf{1}^T A_3^2 \mathbf{1} = (A_3 \mathbf{1})^T (A_3 \mathbf{1}) = |A_3 \mathbf{1}|^2 = 35,$$

and the number of ways to tile a 5×9 rectangle is given by

$$a(5,9) = \mathbf{1}^T A_4^7 \mathbf{1} = (A_4^4 \mathbf{1})^T (A_4^3 \mathbf{1}) = a(9,5) = \mathbf{1}^T A_8^3 \mathbf{1} = (A_8^2 \mathbf{1})^T (A_8 \mathbf{1}) = 59925.$$

The number of multiplications made in the two different ways to find the result in the calculation above is for $a(5,9)$

$$4 \cdot \mathrm{Ones}(A_4) + \mathrm{Rows}(A_4) = 4 \cdot 21 + 8 = 92$$

compared to

$$2 \cdot \mathrm{Ones}(A_8) + \mathrm{Rows}(A_8) = 2 \cdot 341 + 54 = 736,$$

for $a(9,5)$, clearly in favor of the method using the smaller matrix. ⋄

3 Generating the adjacency matrix A_n

In general the matrix A_n is too large to store in the memory of a computer. It is better to generate it dynamically when performing a matrix multiplication. In this section we will show that this way of dealing with A_n can be done in asymptotically optimal time. This means that the time needed to generate A_n is linear in the number of multiplications performed in the matrix multiplication – or similarly, linear in the number of ones in A_n. The space required for this is of order $O(n)$.

3.1 Binary counters

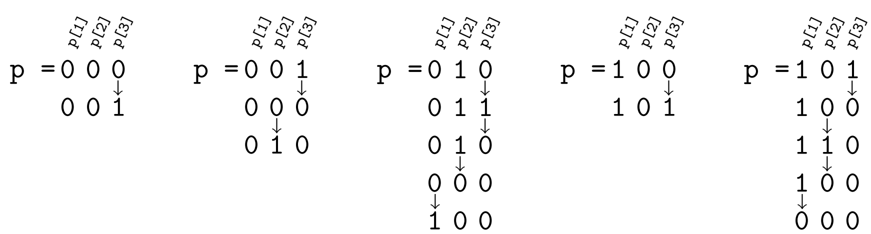
To develop a method to generate the adjacency matrix A_n, we start by considering *binary counters*. A binary counter is a binary vector $p = p_1 \cdots p_n \in \{\mathtt{0}, \mathtt{1}\}^n$ in which we step through all the 2^n possible words in lexicographical order by flipping the bits (or entries) in p. It is a standard exercise to show that each increment (or to generate the next word) of the counter can be made in amortized constant time; see [3]. Recall that amortized constant time means that on average we have to make $O(1)$ operations, for more of this, again see [3].

We give an algorithm (see Figure 3) implemented by the function `Increment_P`, which given $p \in T_n$ generates the lexicographically next word in T_n. We will show that the function `Increment_P` makes such an incrementation of p in amortized constant time. An incrementation is made by the function call `Increment_P(n,p)`, which changes the bits in p and returns true if a word in T_n has been generated and false when there are no more elements to generate.

```
boolean Increment_P(i,p)
{  flip p[i]
   if( p[i] == 0 )
   {  if( i > 1 )
      {  return Increment_P(i-1,p)
      }else
      {  return false
      }
   }else
   {  if( i > 1 and p[i-1] == 1 )
      {  flip p[i]
         return Increment_P(i-1,p)
      }else
      {  return true
      }
   }
}
```

Figure 3: The pseudocode for the function `Increment_P` that increments the vector p. A call of the function gives the lexicographically next word of p in T_n. It returns true if the next word has been found and false if we have generated all words.

The idea in the function `Increment_P` is to check for the subpattern `11` or `0` when flipping bit i in p. If one of these subpatterns exists, we have to go deeper in the recursion and flip bits closer to the front of p. See Figure 4 for a walk–through of the case when p is of length three.



(a) (b) (c) (d) (e)

Figure 4: The function `Increment_P` applied to the vector p of length $n = 3$. The function is called 5 times, illustrated in (a)–(e), where all but the last time (e) returns true. The arrows indicate how the function flips the bits in p. Starting in (a) with p equal to zero, we flip the last bit and return true. In (b) we see that we have to flip bits closer to the front before finding an allowed word. Similar recursive steps happen in (c) and (e).

Let us turn to the performance of this increment function. For this, let $f(n)$ denote the total number of flips of bits in p performed by the function `Increment_P` when called F_{n+2}

times starting form the zero word, that is, during the walk–through of the elements of T_n. Similarly, let $f(n,i)$ be the total number of times that the bit i is flipped during these F_{n+2} calls. Then clearly

$$f(n) = \sum_{i=1}^{n} f(n,i).$$

In Table 2 the result of a computer enumeration of $f(n)$ for some small values of n is presented.

n	1	2	3	4	5	6	7	8
$f(n)$	2	6	12	22	38	64	106	174

Table 2: The number of flips performed by `Increment_P` when stepping through the words of length n without the pattern `11`.

Proposition 3. *The number of times that, starting form the zero word, bit i in a binary counter of length n is flipped during F_{n+2} calls of the function `Increment_P` is given by*

$$f(n,i) = 2F_{i+1}, \tag{8}$$

for $n \geq 1$ and $1 \leq i \leq n$.

Proof. We give a proof by induction on n. By inspection it is clear that $f(n,i) = 2F_{i+1}$ for $n = 1,2$ and $1 \leq i \leq n$.

Now assume for induction that (8) holds for $1 \leq n \leq k$. For the induction step $n = k+1$, notice first that the function `Increment_P` sets all bits back to zero once it has listed all words. This implies that generating the words in T_{k+1} from $\texttt{00}\cdots\texttt{0}$ to $\texttt{10}\cdots\texttt{0}$ uses precisely $f(k,i-1)$ flips for bit number $2 \leq i \leq k+1$ and one flip for bit number 1.

Similarly, to generate all words in T_{k+1} from $\texttt{100}\cdots\texttt{0}$ to $\texttt{110}\cdots\texttt{0}$ requires precisely $f(k-1,i-2)$ flips for bit number $3 \leq i \leq k+1$ and one flip for bit number 2. Thus

$$\begin{aligned} f(k+1,i) &= f(k,i-1) + f(k-1,i-2) \\ &= 2F_{i-1+1} + 2F_{i-2+1} \\ &= 2F_{i+1}, \end{aligned}$$

for $3 \leq i \leq k+1$. For bit number 2, we see that we have flipped it two times when listing the words from $\texttt{00}\cdots\texttt{0}$ to $\texttt{10}\cdots\texttt{0}$ and then one more time when listing the words from $\texttt{100}\cdots\texttt{0}$ to $\texttt{110}\cdots\texttt{0}$, and then finally one more to set it back to `0`. We get $f(k+1,2) = 4 = 2F_3$. For bit number 1 it is clear that it is flipped twice, once when listing the words from $\texttt{00}\cdots\texttt{0}$ to $\texttt{10}\cdots\texttt{0}$ and then one more to reset it. $\square$

From Proposition 3 we find the total number of flips performed to generate all words in T_n to be given by

$$f(n) = \sum_{i=1}^{n} 2\,F_{i+1} = 2\,F_{n+3} - 4. \tag{9}$$

Corollary 4. *An incrementation of the binary vector p of length n with a call of the function* `Increment_P` *is made with an amortized constant number of flips.*

Proof. From (1) and (9) we have the bound

$$\frac{2F_{n+3}-4}{|T_n|} = \frac{2F_{n+3}-4}{F_{n+2}} = 2\frac{F_{n+3}}{F_{n+2}} - \frac{4}{F_{n+2}} \leq 4,$$

since $F_{n+1} \leq 2F_n$, which implies that we on average have to do less than 4 flips for each call of the function `Increment_P`. □

The average number of flips to perform in each call of the incrementation function tends to

$$\lim_{n\to\infty} \frac{2F_{n+3}-4}{|T_n|} = \lim_{n\to\infty} \frac{2F_{n+3}-4}{F_{n+2}} = 1+\sqrt{5} \approx 3.2360\cdots$$

To conclude, we have shown that the words in T_n can be generated in linear time in the size of T_n, that is, in $O\big((\frac{1+\sqrt{5}}{2})^n\big)$ time. The space needed to generate these words is $O(n)$, since we only need to store the vector p of length n.

3.2 Side conditions

In the previous section, we showed how to efficiently generate all the tuples in a matrix representing a tiling. Applying this method to generate the adjacency matrix A_n would require at least $O\big((\frac{1+\sqrt{5}}{2})^{2n}\big)$ operations. This occurs since, for each row in A_n we have a second binary counter generating the columns, and then check if there should be a one in the corresponding entry of A_n. This method is straightforward but has the drawback that it also generates all the entries in A_n that are zero. In this section we show how to overcome this and generate just the entries in A_n that are ones, and thereby reduce the amount of work substantially.

We introduce here a function `Increment_PQ`, see Figure 5, that increases a binary vector q with the side condition $p \in T_n$. The function generates all the elements $q \in T_n$ that do not have a collision with p. A collision with p means that if $p_i = 1$, $1 \leq i \leq n$, then $q_{i+j} = 1$ for some $j = -1, 0, 1$ with $1 \leq i+j \leq n$. We also say that words without collisions are allowed. These allowed words correspond to the ones in A_n, where we see p as the row index and q as the column index in A_n.

The function `Increment_PQ` works in the same way as the function `Increment_P`, with the extra check for a collision with p when incrementing q. If there is a collision, we have to go deeper in the recursion and flip bits closer to the front in q. The function returns true if the incrementation was successful, and false if there are no further words to list. Finally, for each word $p \in T_n$, we start with $q = 0$ and call the function repeatedly until false is returned.

Let us turn to the performance of the function `Increment_PQ`. For this, let $f_p(n)$ denote the total number of flips made when generating all words q when calling `Increment_PQ(n,p,q)` for all $p \in T_n$. That is, the total number of flips performed on q when we generate the matrix

```
boolean Increment_PQ(i,p,q)
{  n = length(p)
   flip q[i]
   if( q[i] == 0 )
   {  if( i > 1 )
      {  return Increment_PQ(i-1,p,q)
      }else
      {  return false
      }
   }else
   {  if( i > 1 )
      {  if( q[i-1] == 1 or p[i-1] == 1 or p[i] == 1 or
             (i < n and p[i+1] == 1) )
         {   flip q[i]
             return Increment_PQ(i-1,p,q)
         }else
         {   return true
         }
      }else
      {  if( p[i] == 1 or (i < n and p[i+1] == 1) )
         {  flip q[i]
            return false
         }else
         {  return true
         }
      }
   }
}
```

Figure 5: The pseudocode for the function `Increment_PQ` that increments the vector q with the side condition p. It returns true if the next word has been found, and false if we have generated all words.

A_n. Similarly, let $f_p(n,i)$ be the total number of times the bits in q are flipped for the ith word p in T_n, where we index the words lexicographically. In other words, this is the number of flips made when generating row i in A_n. Then clearly

$$f_p(n) = \sum_{i=1}^{F_{n+2}} f_p(n,i). \tag{10}$$

In Table 3 the result of a computer enumeration of $f_p(n)$ for some small values of n is presented.

n	1	2	3	4	5	6	7	8
$f_p(n)$	4	14	40	96	222	488	1052	2222

Table 3: The number of flips performed by `Increment_PQ` when stepping through the allowed words q of length n for all words $p \in T_n$.

Lemma 5. *The total number of flips on all q performed by `Increment_PQ(n,p,q)` when going through all $p \in T_n$ is given by the recursion*

$$f_p(n) = f_p(n-1) + 2f_p(n-2) + 8F_n + 2F_{n-1}, \tag{11}$$

for $n > 2$ with $f_p(1) = 4$ and $f_p(2) = 14$.

Proof. The basis conditions are obtained by stepping through the function `Increment_PQ`. For $n = 1$ we have $f_p(1) = f_p(1,1) + f_p(1,2) = 2 + 2 = 4$ and similarly

$$f_p(2) = \sum_{i=1}^{F_4} f_p(2,i) = 6 + 4 + 4 = 14$$

for $n = 2$.

For the recursion (11) recall the recursive structure of the matrix A_n, see Figure 2. Let p be the ith word in T_n. First let us consider the case when $0 < i \leq F_n$. To generate the allowed words q in the interval from $\texttt{00}\cdots\texttt{0}$ to $\texttt{10}\cdots\texttt{0}$ requires $f_p(n-1,i)+1$ flips, where the plus one comes from flipping the first bit. To generate the words in the interval $\texttt{10}\cdots\texttt{0}$ to $\texttt{110}\cdots\texttt{0}$ requires $f_p(n-2,i)+1$ flips, and then an additional 2 flips are made for the two first bits. We therefore have

$$f_p(n,i) = f_p(n-1,i) + f_p(n-2,i) + 4, \quad \text{for} \quad 0 < i \leq F_n.$$

Secondly, for $F_n < i \leq F_{n+1}$, again notice that to generate the allowed words q in the interval from $\texttt{00}\cdots\texttt{0}$ to $\texttt{10}\cdots\texttt{0}$ requires $f_p(n-1,i)+1$ flips. Furthermore, in this case p starts with `01`. This means that after having generated the last allowed word q smaller than $\texttt{10}\cdots\texttt{0}$ the only extra flip made is on the first bit to reset it. Therefore we have

$$f_p(n,i) = f_p(n-1,i) + 2, \quad \text{for} \quad F_n < i \leq F_{n+1}.$$

Third and finally, for $F_{n+1} < i \le F_{n+2}$ we make $f_q(n-2, i-F_{n+1})+1$ flips to generate the allowed words from $\texttt{00}\cdots\texttt{0}$ to $\texttt{01}\cdots\texttt{0}$. As above we notice here that p starts with $\texttt{10}$. This means that after having generated the last allowed word $q < \texttt{01}\cdots\texttt{0}$, the only extra flips made are on the first bits. So,

$$f_p(n,i) = f_p(n-2, i-F_{n+1}) + 4, \quad \text{for} \quad F_{n+1} < i \le F_{n+2}.$$

Summing up the three cases, recalling (10), gives

$$\begin{aligned} f_p(n) = &\sum_{i=1}^{F_n} \big(f_p(n-1,i) + f_p(n-2,i) + 4\big) \\ &+ \sum_{i=F_n+1}^{F_{n+1}} \big(f_p(n-1,i) + 2\big) \\ &+ \sum_{i=F_{n+1}+1}^{F_{n+2}} \big(f_p(n-2, i-F_{n+1}) + 4\big) \\ = &\, f_p(n-1) + 2f_p(n-2) + 8F_n + 2F_{n-1}, \end{aligned}$$

completing the proof. □

It is straightforward to give an explicit solution to the recursion (11) with its initial conditions. We find

$$f_p(n) = \frac{32}{3}2^n - \frac{2}{3}(-1)^n - 8F_{n+2} - 2F_{n+1}.$$

Corollary 6. *Generating the ones in the adjacency matrix A_n can be made in linear time in the number of ones in A_n.*

Proof. If we let $f(A_n)$ denote the total number of flips needed to generate A_n, then

$$\begin{aligned} \frac{f(A_n)}{\mathrm{Ones}(A_n)} &= \frac{f_p(n) + f(n)}{\mathrm{Ones}(A_n)} \\ &= \frac{\frac{32}{3}2^n - \frac{2}{3}(-1)^n - 8F_{n+2} - 2F_{n+1} + 2F_{n+3} - 4}{\frac{4}{3}\,2^n - \frac{1}{3}\,(-1)^n} \\ &= \frac{32\cdot 2^n - 2(-1)^n - 18F_{n+2} - 12}{4\cdot 2^n - (-1)^n} \\ &\le \frac{32\cdot 2^n - 18F_{n+2} - 10}{4\cdot 2^n - 1} \\ &\le 8, \end{aligned}$$

where the last inequality follows from

$$8(4\cdot 2^n - 1) - (32\cdot 2^n - 18F_{n+2} - 10) = 18F_{n+2} + 2 \ge 0.$$ □

To conclude, the space needed to generate A_n is $O(n)$, since we just have to store the vectors p and q, and the work needed to do this is linear in the number of ones in A_n.

4 Matrix multiplication

In the previous section we have seen how to generate the adjacency matrix A_n. In this section we describe how to use it efficiently in the matrix multiplication. First we present how to count the number of tilings, without respect to what tiles we use - we turn to that question in the second subsection.

4.1 Counting

The rows and columns in A_n are indexed in lexicographical order with the words of T_n. To simplify the access into a specific position in A_n we introduce the index function $I_n : T_n \to \mathbb{N}$ by

$$I_n(t) = 1 + \sum_{i=1}^{n} F_{n+2-i}\, t_i, \tag{12}$$

for $t \in T_n$ with $t = t_1 \cdots t_n$. It is easy to show that I_n indexes the words of T_n in order without gaps. The function I_n is based on the well-known exercise of writing a natural number in the Fibonacci base system. In the multiplication

$$u = A_n^k\, \mathbf{1}$$

the entry u_i is the number of all paths of length k in the graph G_n (representing the tilings) that end in node N with $i = I_n(N)$. By the length of a path we mean the number of visited edges in the path.

Now we have all the parts for the matrix multiplication A_n with a column vector $v^T = (v_1, v_2, \ldots, v_{F_{n+2}})$. The actual matrix multiplications are now performed according to (7). The pseudocode for the matrix multiplication $A_n\, v$ is given in Figure 6.

The algorithm for the matrix multiplication works as follows. We initiate a binary counter *row* and for each increment of it we initiate and step through a second binary counter *column*. We increment *row* with `Increment_P` and *column* with `Increment_PQ` where we have *row* as the side condition. This corresponds to stepping through the matrix A_n from the upper leftmost position going rightwards to the end of the row before moving to the next row, restarting from the leftmost position, and so on. As the function `Increment_PQ` does not go through all the entries on a row, we have to use the index function I_n to find the correct position. At each position, we just have to add the entry v_i to a temporary vector v_{tmp}. We do not have to make any multiplications since all non-zero entries of A_n are ones. After stepping through all rows of A_n we return the temporary vector v_{tmp}.

4.2 Detailed count

Here we present how to keep track of the number of tilings with a specific number of large tiles. The multiplication follows the pattern given in the previous section, with only a few

```
vector MatrixMultiplication(n,v)
{  initiate the binary vector 'row' of length n to zero
   initiate the vector 'v_tmp' of length F(n+2) with zeros
   do
   {  initiate the binary vector 'column' of length n to zero
      r = I_n(row)
      do
      {  c = I_n(column)
         v_tmp[r] = v_tmp[r] + v[c]
      }
      while(Increment_PQ(n,row,column))
   }
   while(Increment_P(n,row))
   return v_tmp
}
```

Figure 6: The pseudocode for performing the multiplication $A_n v$. The function I_n is the index function from (12). Note that no actual multiplication is performed, as all non-zero entries of A_n are 1.

changes. The vector v is modified to be a vector of $(m+1)$-tuples, where m is the maximum number of big tiles in any of the tilings we are going to count.

Next, the initialization of v is made by

```
for(t in T_n)
{   v[I_n(t)][Ones(t)] = 1
}
```

where Ones gives the number of ones in a word, and all the other entries are set to zero. Thus, in the multiplication

$$u = A_n^k v,$$

u_{ij} represents the number of all paths of length k in G_n that end in node N, with $i = I_n(N)$, which contain precisely j ones. That is, j is the total number of ones in the nodes along such paths.

In the multiplication we have to take into account the number of ones which are in the word representing the current row in A_n, say $o = \text{Ones}(row)$. Now the entry-wise multiplication will just be to add tuple v_c to the tuple $(v_{tmp})_r$, where the latter is shifted o steps to the left. This is to take care of the extra o ones that we get by adding node row to the paths. See the pseudocode in Figure 7.

```
vector MatrixMultiplicationDetailed(n,m,v)
{  initiate the binary vector 'row' of length n to zero
   initiate the array 'v_tmp' of size F(n+2) X m with zeros
   do
   {  initiate the binary vector 'column' of length n to zero
      r = I_n(row)
      o = Ones(row)
      do
      {   c = I_n(column)
          for(i=0; o+i<length(v_tmp[r]); i++)
          {   v_tmp[r][o+i] = v_tmp[r][o+i] + v[c][i]
          }
      }
      while(Increment_PQ(n,row,column))
   }
   while(Increment_P(n,row))
   return v_tmp
}
```

Figure 7: The pseudocode for performing the multiplication $A_n v$, where we count of the number of tilings with a specific number of large tiles.

Example 7. Let us return to the tilings of a 4×4 square that we considered in Example 2. We initialize v to be the following array

$$v = \begin{array}{c c} & \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} \texttt{000} \\ \texttt{001} \\ \texttt{010} \\ \texttt{100} \\ \texttt{101} \end{array} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Clearly, the number of columns in v has to be adopted to the maximum number of large squares in any of the tilings of the rectangle to tile. After multiplication with A_3 we have

$$A_3 v = \begin{bmatrix} 1 & 3 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad A_3^2 v = \begin{bmatrix} 1 & 6 & 4 & 0 & 0 \\ 0 & 1 & 4 & 2 & 0 \\ 0 & 1 & 3 & 1 & 0 \\ 0 & 1 & 4 & 2 & 0 \\ 0 & 0 & 1 & 3 & 1 \end{bmatrix}.$$

Therefore, by summing up the columns,

$$\mathbf{1}^T A_3^2\, v = \begin{bmatrix} 1 & 9 & 16 & 8 & 1 \end{bmatrix}.$$

This tells us there is 1 tiling with no 2×2 square, 9 tilings with precisely one 2×2 square, 16 with precisely two 2×2 squares, and so on. In the same way, we can calculate the number of tilings of a 5×9 rectangle

$$\mathbf{1}^T A_4^7 v = \begin{bmatrix}1 & 32 & 402 & 2\,564 & 9\,009 & 17\,696 & 18\,738 & 9\,636 & 1\,847\end{bmatrix}$$

which of course again sums up to 59925.

Unfortunately there seems to be no way to reduce the number of matrix multiplications in this case with detailed count, as we did in (6). At this moment we must leave the question about finding such a shortcut open. $\diamond$

5 Enumerations

With an implementation in JAVA of this algorithm, we calculated the number of tilings of a square S of size $n \times n$, for $n = 1, 2, \ldots, 40$. The result is in detail presented in the OEIS [4] under the sequence A063443. See also Table 4 for some of the sequence values.

n	$a(n,n)$, A063443
1	1
2	2
3	5
4	35
5	314
6	6427
7	202841
8	12727570
9	1355115601
10	269718819131
⋮	
36	$7.512803 \cdot 10^{160}$
37	$1.198698 \cdot 10^{170}$
38	$3.447777 \cdot 10^{179}$
39	$1.787377 \cdot 10^{189}$
40	$1.670949 \cdot 10^{199}$

Table 4: The number of tilings of a $n \times n$ square.

In the same spirit, in Figure 8 we illustrate the distribution of the number of tilings of a 23×23 square, with a specified number of 2×2 tiles.

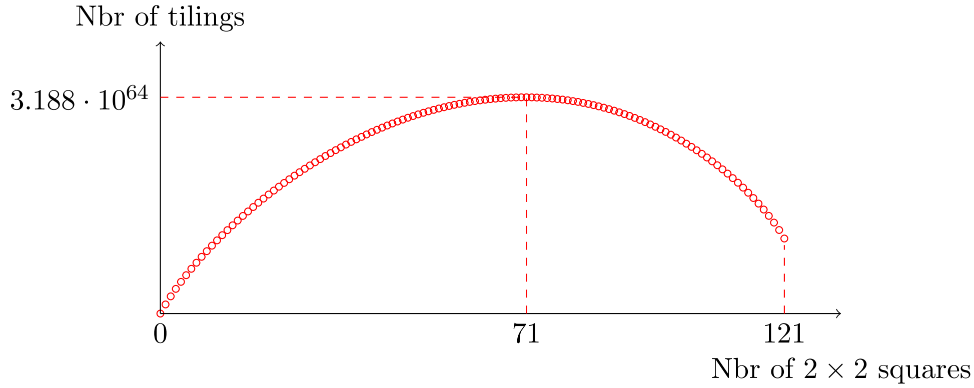


Figure 8: The distribution of the number of tilings of a 23×23 square with given number of 2×2 squares. The y–axis is drawn in a logarithmic scale.

6 Acknowledgment

The author wishes to thank T. Fernique and A. Ugolnikova at Université Paris 13 for our discussions of the problem and for reading drafts of the manuscript. This work was supported by the ANR project QuasiCool (ANR-12-JS02-011-01).

References

[1] N. J. Calkin, K. James, S. Purvis, S. Race, K. Schneider, and M. Yancey, Counting kings: as easy as λ_1, λ_2, λ_3,.... *Proceedings of the Thirty-Seventh Southeastern International Conference on Combinatorics, Graph Theory and Computing.*, Congr. Numer. **183** (2006), 83–95.

[2] S. Race, K. Schneider, and M. Yancey, The kings problem. Manuscript available at http://www4.ncsu.edu/~slrace/gsskings.pdf.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

[4] N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences. Published electronically at http://oeis.org .

[5] H. Wilf, The problem of the kings, *Electronic J. Combinatorics* **2** (1995), #R3. Available at http://www.combinatorics.org/ojs/index.php/eljc/article/view/v2i1r3.

2010 Mathematics Subject Classification: Primary 68R05; Secondary 68Q25.

Keywords: tiling, analysis of algorithms, combinatorics.

(Concerned with sequences A001045, A063443, A193580 and A245013.)

Received April 20 2016; revised version received December 1 2016. Published in *Journal of Integer Sequences*, December 27 2016.

Return to Journal of Integer Sequences home page.