Building up the recurrence matrix to compute recurrences in O(logN) time

27

9

As these new editorials for SEPT12 were released, the interest in matrix exponentiation grew a lot, and I will try to provide a relatively complete, yet simple tutorial on the most simple recurrence relationships and how they can be written in the matrix form.

- Advantages of using matrix form instead of the recurrence relationship itself

To use matrix exponentiation it's first necessary to understand **why** we would want to use it... After all, methods such as classic DP and/or memoization are available and they are easier to code.

The great advantage of matrix exponentiation is that its running time is simply O(k^3 * logN)(for a matrix of dimensions k*k) which is critical when we are dealing with values as large as 10^15, for example. It is used when the recursive relationship we derived is somehow **entangled**, in the sense that the values it takes depend on more than one of the previous values...Using the base cases of the recurrence relation and a repeated squaring/fast exponentiation algorithm, we have a very efficent way of dealing with large input values :D I will try to illustrate this with an example, the **Tribonacci Numbers**.

- The Tribonacci Numbers

For those of you who had never heard the name before, this sequence of numbers is an "expansion" of the fibonacci sequence that includes a third term on the sum of the previous two, such that the formula looks like:

```
F(n) = F(n-1) + F(n-2) + F(n-3),  F(1) = 1; F(2) = 1; F(3) = 2
```

as stated on WolframMathworld.

Of course, all the problems that arose when we tried to compute the fibonnaci numbers via dp or any other way become a lot more complicated with tribonacci numbers, and for N as large as 10^15, using dp will always be very slow, regardless of the time limit.

- Understanding Matrix exponentiation

The basic idea behind matrix exponentiation, as stated earlier is to use the base cases of the recurrence relationship in order to assemble a matrix which will allow us to compute values fast.

On our case we have:

F(1) = 1

F(2) = 1

F(3) = 2

And we now have a relationship that will go like this:

```
|f(4)|    = MATRIX * |f(1)|
|f(3)|               |f(2)|
|f(2)|               |f(3)|
```

Now all that is left is to assemble the matrix... and that is done based both on the rules of matrix multiplication and on the recursive relationship... Now, on our example we see that to obtain f(4), the 1st line of the matrix needs to be composed only of ones, as f(4) = 1 *f(3) + 1* f(2) + 1* f(1).

Now, denoting the unknown elements as *, we have:

```
|f(4)|   = |1 1 1| * |f(1)|
|f(3)|     |* * *|   |f(2)|
|f(2)|     |* * *|   |f(3)|
```

For the second line, we want to obtain f(3), and the only possible way of doing it is by having:

```
|f(4)|   = |1 1 1| * |f(1)|
|f(3)|     |0 0 1|   |f(2)|
|f(2)|     |* * *|   |f(3)|
```

To get the value of f(2), we can follow the same logic and get the final matrix:

```
|1 1 1|
|0 0 1|
|0 1 0|
```

To end it, we now need to generalize it, and, as we have 3 base cases, all we need to do to compute the Nth tribonacci number in O(logN) time, is to raise the matrix to the power N -3 to get:

```
|f(n)|   =   |1 1 1|^(N-3) * |f(1)|
|f(n-1)|     |0 0 1|         |f(2)|
|f(n-2)|     |0 1 0|         |f(3)|
```

The power of the matrix can now be computed in O(logN) time using repeated squaring applied to matrices and the method is complete... Below is the Python code that does this, computing the number modulo 10000000007.

```
 def matrix_mult(A, B):
  C = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
  for i in range(3):
    for j in range(3):
      for k in range(3):
        C[i][k] = (C[i][k] + A[i][j] * B[j][k]) % 1000000007
  return C

def fast_exponentiation(A, n):
  if n == 1:
    return A
  else:
    if n % 2 == 0:
      A1 = fast_exponentiation(A, n/2)
      return matrix_mult(A1, A1)
    else:
      return matrix_mult(A, fast_exponentiation(A, n - 1))

def solve(n):
    A = [[1,1,1],[0,0,1],[0,1,0]]
    A_n = fast_exponentiation(A,n-3)
    return A_n[0][0] + A_n[0][1] + A_n[0][2]*2
```

I hope you have liked my tutorial!!

Cheers, Bruno Oliveira

Edit: As practice problems, CROWD, CSUMD and the problem Plants on the codeforces website are good starting places

numbers tribonacci matrix-expo

This question is marked "community wiki".

edited **22 Sep '12, 14:04**

asked **12 Sep '12, 04:05**
kuruma
16.6k●72●143●208
accept rate: 8%

2 Nice tutorial @kuruma......it helped me understand in simple terms...now i can implement it in C as your python code is pretty neat & clean.

smaurya73 (12 Sep '12, 14:15)

1 I am glad I can help this wondeful community which has also given a lot to me in terms of learning computer programming techniques!!! It's a pleasure :D

kuruma (12 Sep '12, 14:17)

1 Awesome tutorial . @kuruma it was really good and easy to learn .

phanindhar (12 Sep '12, 15:00)

---

6 Answers:

oldest newest **most voted**

8

That should be `M^(n-3)` for all n>3 where M is the base Matrix, because you already have the first 3 values of f(n).

Also, writing matrix F as:

```
|f(3)|
|f(2)|
|f(1)|
```

would be sequentially more consistent with the values of f(n) in F.

link | award points

edited **12 Sep '12, 14:31**

answered **12 Sep '12, 05:27**
rushilpaul
284●1●6●11
accept rate: 10%

I like your way writing ...It is actually very good to understand..

| 1 1 1|^(N-3) |f(3)| | 0 1 0| |f(2)| | 0 0 1| |f(1)|

thanks a lot........

rcsldav2017 (25 Sep, 02:48)

8

"The great advantage of matrix exponentiation is that its running time is simply `O(logN)`" should be "The great advantage of matrix exponentiation is that its running time is simply `O(k^3 * logN)` where matrix is of size `k x k`"

link | award points

answered **12 Sep '12, 10:57**
svm11
409●3●7●9
accept rate: 12%

2

Also, and although it requires a bit more mathematics, it is possible to reduce some recurrences to a closed form, which allow the direct computation of the N-th term of a recurrence... That is also useful when the values of N are very large and one has doubts about the matrix assemblation...As this was mentioned on the comments for the problem CROWD, I recommend everyone who is interested in reading more about "closing" recurrences, the following link:

http://hcmop.wordpress.com/2012/04/20/using-characteristic-equation-to-solve-general-linear-recurrence-relations/

Bruno

link | award points edited 12 Sep '12, 20:34 answered 12 Sep '12, 20:33
kuruma
16.6k●72●143●208
accept rate: 8%

I doubt that would work in a programming contest. Errors would arise while computing the value of a closed form for n (since there are irrational numbers involved). As n gets larger, the error will increase. Therefore using closed forms is not recommended.

rushilpaul (14 Sep '12, 10:04)

Yes, I understand that, but it was just left here as a curiosity, plus, some forms are totally accurate :D

kuruma (14 Sep '12, 13:18)

1

Thank you very much, the power is corrected... I guess I got confused starting with value 1...

link | award points answered 12 Sep '12, 05:31
kuruma
16.6k●72●143●208
accept rate: 8%

1

Thank you everyone for your corrections... As this article was inspired by some tutorials on specific problems and a mixture of my knowledge, it's natural that some things were not that accurate, so thank you for clearing them out both for me in particular and for everyone who reads the post!!

Much appreciated!!!

link | award points answered 12 Sep '12, 13:57
kuruma
16.6k●72●143●208
accept rate: 8%

0

I will also add this tutorial to that list of links we are now compiling on that post :)

@admin, it would be nice if you could pin that post :D Then we would be able to make a really good "reference book" to all these important topics!

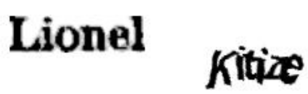
Best regards,

Bruno

link | award points answered 12 Aug '13, 23:11
kuruma
16.6k●72●143●208
accept rate: 8%

Your answer

[hide preview] community wiki



Type the text
Privacy & Terms

Post Your Answer