

COME ON CODE ON

A blog about programming and more programming.

Archive for the 'Algorithm' Category

Combination

with 14 comments

In mathematics a combination is a way of selecting several things out of a larger group, where (unlike permutations) order does not matter. More formally a k-combination of a set S is a subset of k distinct elements of S. If the set has n elements the number of k-combinations is equal to the binomial coefficient. In this post we will see different methods to calculate the binomial.

1. Using Factorials

We can calculate nCr directly using the factorials.

$nCr = n! / (r! * (n-r)!)$

```
1  #include<iostream>
2  using namespace std;
3
4  long long C(int n, int r)
5  {
6      long long f[n + 1];
7      f[0]=1;
8      for (int i=1;i<=n;i++)
9          f[i]=i*f[i-1];
10     return f[n]/f[r]/f[n-r];
11 }
12
13 int main()
14 {
15     int n,r,m;
16     while (~scanf("%d%d",&n,&r))
17     {
18         printf("%lld\n",C(n, min(r,n-r)));
19     }
20 }
```

But this will work for only factorial below 20 in C++. For larger factorials you can either write big factorial library or use a language like Python. The time complexity is $O(n)$.

If we have to calculate $nCr \bmod p$ (where p is a prime), we can calculate factorial mod p and then use modular inverse to find $nCr \bmod p$. If we have to find $nCr \bmod m$ (where m is not prime), we can factorize m into primes and then use Chinese Remainder Theorem(CRT) to find $nCr \bmod m$.

```

1  #include<iostream>
2  using namespace std;
3  #include<vector>
4
5  /* This function calculates (a^b)%MOD */
6  long long pow(int a, int b, int MOD)
7  {
8      long long x=1,y=a;
9      while(b > 0)
10     {
11         if(b%2 == 1)
12         {
13             x=(x*y);
14             if(x>MOD) x%=MOD;
15         }
16         y = (y*y);
17         if(y>MOD) y%=MOD;
18         b /= 2;
19     }
20     return x;
21 }
22
23 /* Modular Multiplicative Inverse
24 Using Euler's Theorem
25 a^(phi(m)) = 1 (mod m)
26 a^(-1) = a^(m-2) (mod m) */
27 long long InverseEuler(int n, int MOD)
28 {
29     return pow(n,MOD-2,MOD);
30 }
31
32 long long C(int n, int r, int MOD)
33 {
34     vector<long long> f(n + 1,1);
35     for (int i=2; i<=n;i++)
36         f[i]= (f[i-1]*i) % MOD;
37     return (f[n]*((InverseEuler(f[r], MOD) * InverseEuler(f[n-r],
38 }
39
40 int main()
41 {
42     int n,r,p;
43     while (~scanf("%d%d%d",&n,&r,&p))
44     {
45         printf("%lld\n",C(n,r,p));
46     }
47 }
```

2. Using Recurrence Relation for nCr

The recurrence relation for nCr is $C(i,k) = C(i-1,k-1) + C(i-1,k)$. Thus we can calculate nCr in time complexity $O(n*r)$ and space complexity $O(n*r)$.

```

1  #include<iostream>
2  using namespace std;
3  #include<vector>
4
5  /*
6      C(n,r) mod m
7      Using recurrence:
8      C(i,k) = C(i-1,k-1) + C(i-1,k)
9      Time Complexity: O(n*r)
10     Space Complexity: O(n*r)
11  */
12
13 long long C(int n, int r, int MOD)
14 {
15     vector< vector<long long> > C(n+1,vector<long long> (r+1,0));
16
17     for (int i=0; i<=n; i++)
18     {
19         for (int k=0; k<=r && k<=i; k++)
20             if (k==0 || k==i)
21                 C[i][k] = 1;
22             else
23                 C[i][k] = (C[i-1][k-1] + C[i-1][k])%MOD;
24     }
25     return C[n][r];
26 }
27 int main()
28 {
29     int n,r,m;
30     while (~scanf("%d%d%d",&n,&r,&m))
31     {
32         printf("%lld\n",C(n, r, m));
33     }
34 }

```

We can easily reduce the space complexity of the above solution by just keeping track of the previous row as we don't need the rest rows.

```

1  #include<iostream>
2  using namespace std;
3  #include<vector>
4
5  /*
6      Time Complexity: O(n*r)
7      Space Complexity: O(r)
8  */
9  long long C(int n, int r, int MOD)
10 {
11     vector< vector<long long> > C(2,vector<long long> (r+1,0));
12
13     for (int i=0; i<=n; i++)
14     {
15         for (int k=0; k<=r && k<=i; k++)
16             if (k==0 || k==i)
17                 C[i&1][k] = 1;
18             else
19                 C[i&1][k] = (C[(i-1)&1][k-1] + C[(i-1)&1][k])%MOD;
20     }

```

```

21     return C[n&1][r];
22 }
23
24 int main()
25 {
26     int n,r,m,i,k;
27     while (~scanf("%d%d%d",&n,&r,&m))
28     {
29         printf("%lld\n",C(n, r, m));
30     }
31 }

```

3. Using expansion of nCr

Since

$$\begin{aligned}
 C(n,k) &= \frac{n!}{((n-k)!k!)} \\
 &= \frac{[n(n-1)\dots(n-k+1)][(n-k)\dots(1)]}{[(n-k)\dots(1)][k(k-1)\dots(1)]}
 \end{aligned}$$

We can cancel the terms: $[(n-k)\dots(1)]$ as they appear both on top and bottom, leaving:

$$\frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots(1)}$$

which we might write as:

$$\begin{aligned}
 C(n,k) &= 1, & \text{if } k = 0 \\
 &= (n/k) * C(n-1, k-1), & \text{otherwise}
 \end{aligned}$$

```

1  #include<iostream>
2  using namespace std;
3
4  long long C(int n, int r)
5  {
6      if (r==0) return 1;
7      else return C(n-1,r-1) * n / r;
8  }
9
10 int main()
11 {
12     int n,r,m;
13     while (~scanf("%d%d",&n,&r))
14     {
15         printf("%lld\n",C(n, min(r,n-r)));
16     }
17 }

```

4. Using Matrix Multiplication

In the last post we learned how to use Fast Matrix Multiplication to calculate functions having linear equations in logarithmic time. Here we have the recurrence relation $C(i,k) = C(i-1,k-1) + C(i-1,k)$.

If we take $k=3$ we can write,

$$C(i-1,1) + C(i-1,0) = C(i,1)$$

$$C(i-1,2) + C(i-1,1) = C(i,2)$$

$$C(i-1,3) + C(i-1,2) = C(i,3)$$

Now on the left side we have four variables $C(i-1,0)$, $C(i-1,1)$, $C(i-1,2)$ and $C(i-1,3)$.

On the right side we have three variables $C(i,1)$, $C(i,2)$ and $C(i,3)$.

We need those two sets to be the same, except that the right side index numbers should be one higher than the left side index numbers. So we add $C(i,0)$ on the right side. NOW let's get our all important Matrix.

$$\begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \begin{pmatrix} C(i-1,0) \\ C(i-1,1) \\ C(i-1,2) \\ C(i-1,3) \end{pmatrix} = \begin{pmatrix} C(i,0) \\ C(i,1) \\ C(i,2) \\ C(i,3) \end{pmatrix}$$

The last three rows are trivial and can be filled from the recurrence equations above.

$$\begin{pmatrix} . & . & . & . \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} C(i-1,0) \\ C(i-1,1) \\ C(i-1,2) \\ C(i-1,3) \end{pmatrix} = \begin{pmatrix} C(i,0) \\ C(i,1) \\ C(i,2) \\ C(i,3) \end{pmatrix}$$

The first row, for $C(i,0)$, depends on what is supposed to happen when $k = 0$. We know that $C(i,0) = 1$ for all i when $k=0$. So the matrix reduces to

$$\begin{pmatrix} . & . & . & . \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} C(i-1,0) \\ C(i-1,1) \\ C(i-1,2) \\ C(i-1,3) \end{pmatrix} = \begin{pmatrix} C(i,0) \\ C(i,1) \\ C(i,2) \\ C(i,3) \end{pmatrix}$$

And this then leads to the general form:

$$\begin{pmatrix} . & . & . & . \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}^i \begin{pmatrix} C(0,0) \\ C(0,1) \\ C(0,2) \\ C(0,3) \end{pmatrix} = \begin{pmatrix} C(i,0) \\ C(i,1) \\ C(i,2) \\ C(i,3) \end{pmatrix}$$

For example if we want $C(4,3)$ we just raise the above matrix to the 4th power.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}^4 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \\ 6 \\ 4 \end{pmatrix}$$

Here's a C++ code.

```

1  #include<iostream>
2  using namespace std;
3
4  /*
5      C(n,r) mod m
6      Using Matrix Exponentiation
7      Time Complexity: O((r^3)*log(n))
8      Space Complexity: O(r*r)
9  */
10
11 long long MOD;
12
13 template< class T >
14 class Matrix
15 {
16     public:
```

```

17         int m,n;
18         T *data;
19
20         Matrix( int m, int n );
21         Matrix( const Matrix< T > &matrix );
22
23         const Matrix< T > &operator=( const Matrix< T > &A );
24         const Matrix< T > operator*( const Matrix< T > &A );
25         const Matrix< T > operator^( int P );
26
27         ~Matrix();
28     };
29
30     template< class T >
31     Matrix< T >::Matrix( int m, int n )
32     {
33         this->m = m;
34         this->n = n;
35         data = new T[m*n];
36     }
37
38     template< class T >
39     Matrix< T >::Matrix( const Matrix< T > &A )
40     {
41         this->m = A.m;
42         this->n = A.n;
43         data = new T[m*n];
44         for( int i = 0; i < m * n; i++ )
45             data[i] = A.data[i];
46     }
47
48     template< class T >
49     Matrix< T >::~~Matrix()
50     {
51         delete [] data;
52     }
53
54     template< class T >
55     const Matrix< T > &Matrix< T >::operator=( const Matrix< T > &A )
56     {
57         if( &A != this )
58         {
59             delete [] data;
60             m = A.m;
61             n = A.n;
62             data = new T[m*n];
63             for( int i = 0; i < m * n; i++ )
64                 data[i] = A.data[i];
65         }
66         return *this;
67     }
68
69     template< class T >
70     const Matrix< T > Matrix< T >::operator*( const Matrix< T > &A )
71     {
72         Matrix C( m, A.n );
73         for( int i = 0; i < m; ++i )
74             for( int j = 0; j < A.n; ++j )
75             {

```

```

76         C.data[i*C.n+j]=0;
77         for( int k = 0; k < n; ++k )
78             C.data[i*C.n+j] = (C.data[i*C.n+j] + (data[i*n+k]
79         )
80     return C;
81 }
82
83 template< class T >
84 const Matrix< T > Matrix< T >::operator^( int P )
85 {
86     if( P == 1 ) return (*this);
87     if( P & 1 ) return (*this) * ((*this) ^ (P-1));
88     Matrix B = (*this) ^ (P/2);
89     return B*B;
90 }
91
92 long long C(int n, int r)
93 {
94     Matrix<long long> M(r+1,r+1);
95     for (int i=0;i<(r+1)*(r+1);i++)
96         M.data[i]=0;
97     M.data[0]=1;
98     for (int i=1;i<r+1;i++)
99     {
100         M.data[i*(r+1)+i-1]=1;
101         M.data[i*(r+1)+i]=1;
102     }
103     return (M^n).data[r*(r+1)];
104 }
105
106 int main()
107 {
108     int n,r;
109     while (~scanf( "%d%d%lld", &n, &r, &MOD))
110     {
111         printf("%lld\n", C(n, r));
112     }
113 }

```

5. Using the power of prime p in n factorial

The power of prime p in n factorial is given by

$$\epsilon_p = \lfloor n/p \rfloor + \lfloor n/p^2 \rfloor + \lfloor n/p^3 \rfloor \dots$$

If we call the power of p in n factorial, the power of p in nCr is given by

$$e = \text{countFact}(n,i) - \text{countFact}(r,i) - \text{countFact}(n-r,i)$$

To get the result we multiply p^e for all p less than n.

```

1  #include<iostream>
2  using namespace std;
3  #include<vector>
4
5  /* This function calculates power of p in n! */
6  int countFact(int n, int p)
7  {
8      int k=0;
9      while (n>0)
10     {
11         k+=n/p;

```

```

12         n/=p;
13     }
14     return k;
15 }
16
17 /* This function calculates (a^b)%MOD */
18 long long pow(int a, int b, int MOD)
19 {
20     long long x=1,y=a;
21     while(b > 0)
22     {
23         if(b%2 == 1)
24         {
25             x=(x*y);
26             if(x>MOD) x%=MOD;
27         }
28         y = (y*y);
29         if(y>MOD) y%=MOD;
30         b /= 2;
31     }
32     return x;
33 }
34
35 long long C(int n, int r, int MOD)
36 {
37     long long res = 1;
38     vector<bool> isPrime(n+1,1);
39     for (int i=2; i<=n; i++)
40         if (isPrime[i])
41         {
42             for (int j=2*i; j<=n; j+=i)
43                 isPrime[j]=0;
44             int k = countFact(n,i) - countFact(r,i) - countFact(n-
45             res = (res * pow(i, k, MOD)) % MOD;
46         }
47     return res;
48 }
49
50 int main()
51 {
52     int n,r,m;
53     while (scanf("%d%d%d",&n,&r,&m))
54     {
55         printf("%lld\n",C(n,r,m));
56     }
57 }

```

6. Using Lucas Theorem

For non-negative integers m and n and a prime p , the following congruence relation holds:

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{n_i} \pmod{p},$$

where

$$m = m_k p^k + m_{k-1} p^{k-1} + \dots + m_1 p + m_0,$$

and

$$n = n_k p^k + n_{k-1} p^{k-1} + \dots + n_1 p + n_0$$

are the base p expansions of m and n respectively.

We only need to calculate nCr only for small numbers (less than equal to p) using any of the above methods.

```

1  #include<iostream>
2  using namespace std;
3  #include<vector>
4
5  long long SmallC(int n, int r, int MOD)
6  {
7      vector< vector<long long> > C(2,vector<long long> (r+1,0));
8
9      for (int i=0; i<=n; i++)
10     {
11         for (int k=0; k<=r && k<=i; k++)
12             if (k==0 || k==i)
13                 C[i&1][k] = 1;
14             else
15                 C[i&1][k] = (C[(i-1)&1][k-1] + C[(i-1)&1][k])%MOD;
16     }
17     return C[n&1][r];
18 }
19
20 long long Lucas(int n, int m, int p)
21 {
22     if (n==0 && m==0) return 1;
23     int ni = n % p;
24     int mi = m % p;
25     if (mi>ni) return 0;
26     return Lucas(n/p, m/p, p) * SmallC(ni, mi, p);
27 }
28
29 long long C(int n, int r, int MOD)
30 {
31     return Lucas(n, r, MOD);
32 }
33
34 int main()
35 {
36
37     int n,r,p;
38     while (~scanf("%d%d%d",&n,&r,&p))
39     {
40         printf("%lld\n",C(n,r,p));
41     }
42 }

```

7. Using special $n! \bmod p$

We will calculate n factorial $\bmod p$ and similarly inverse of $r! \bmod p$ and $(n-r)! \bmod p$ and multiply to find the result. But while calculating factorial $\bmod p$ we remove all the multiples of p and write

$$n! \bmod p = 1 * 2 * \dots * (p-1) * 1 * 2 * \dots * (p-1) * 2 * 1 * 2 * \dots * n.$$

We took the usual factorial, but excluded all factors of p (1 instead of p , 2 instead of $2p$, and so on). Lets call this *strange factorial*.

So *strange factorial* is really several blocks of construction:

$$1 * 2 * 3 * \dots * (p-1) * i$$

where i is a 1-indexed index of block taken again without factors p .

The last block could be *not* full. More precisely, there will be $\text{floor}(n/p)$ full blocks and some tail (its result we can compute easily, in $O(P)$).

The result in each block is multiplication $1 * 2 * \dots * (p-1)$, which is common to all blocks, and multiplication of all *strange indices* i from 1 to $\text{floor}(n/p)$.

But multiplication of all *strange indices* is really a strange factorial again, so we can compute it recursively. Note, that in recursive calls n reduces exponentially, so this is rather fast algorithm.

Here's the algorithm to calculate *strange factorial*.

```

1  int factMOD(int n, int MOD)
2  {
3      long long res = 1;
4      while (n > 1)
5      {
6          long long cur = 1;
7          for (int i=2; i<MOD; ++i)
8              cur = (cur * i) % MOD;
9          res = (res * powmod (cur, n/MOD, MOD)) % MOD;
10         for (int i=2; i<=n%MOD; ++i)
11             res = (res * i) % MOD;
12         n /= MOD;
13     }
14     return int (res % MOD);
15 }
```

But we can still reduce our complexity.

By Wilson's Theorem, we know $(n-1)! \equiv -1 \pmod{n}$ for all primes n . SO our method reduces to:

```

1  long long factMOD(int n, int MOD)
2  {
3      long long res = 1;
4      while (n > 1)
5      {
6          res = (res * pow(MOD - 1, n/MOD, MOD)) % MOD;
7          for (int i=2, j=n%MOD; i<=j; i++)
8              res = (res * i) % MOD;
9          n/=MOD;
10     }
11     return res;
12 }
```

Now in the above code we are calculating $(-1)^{(n/p)}$. If (n/p) is even what we are multiplying by 1, so we can skip that. We only need to consider the case when (n/p) is odd, in which case we are multiplying result by $(-1)\%MOD$, which ultimately is equal to $MOD-res$. SO our method again reduces to:

```

1  long long factMOD(int n, int MOD)
2  {
3      long long res = 1;
4      while (n > 0)
5      {
6          for (int i=2, m=n%MOD; i<=m; i++)
7              res = (res * i) % MOD;
8          if ((n/=MOD)%2 > 0)
9              res = MOD - res;
10     }
11     return res;
12 }
```

Finally the complete code here:

```

1  #include<iostream>
2  using namespace std;
3  #include<vector>
4
5  /* This function calculates power of p in n! */
6  int countFact(int n, int p)
7  {
8      int k=0;
9      while (n>=p)
10     {
11         k+=n/p;
12         n/=p;
13     }
14     return k;
15 }
16
17 /* This function calculates (a^b)%MOD */
18 long long pow(int a, int b, int MOD)
19 {
20     long long x=1,y=a;
21     while(b > 0)
22     {
23         if(b%2 == 1)
24         {
25             x=(x*y);
26             if(x>MOD) x%=MOD;
27         }
28         y = (y*y);
29         if(y>MOD) y%=MOD;
30         b /= 2;
31     }
32     return x;
33 }
34
35 /* Modular Multiplicative Inverse
36 Using Euler's Theorem
37  $a^{(\phi(m))} = 1 \pmod{m}$ 
38  $a^{-1} = a^{(m-2)} \pmod{m}$  */
39 long long InverseEuler(int n, int MOD)
40 {
41     return pow(n,MOD-2,MOD);
42 }
43
44 long long factMOD(int n, int MOD)
45 {
46     long long res = 1;
47     while (n > 0)
48     {
49         for (int i=2, m=n%MOD; i<=m; i++)
50             res = (res * i) % MOD;
51         if ((n/=MOD)%2 > 0)
52             res = MOD - res;
53     }
54     return res;
55 }
56
57 long long C(int n, int r, int MOD)

```

```

58 {
59     if (countFact(n, MOD) > countFact(r, MOD) + countFact(n-r, MOD)
60         return 0;
61
62     return (factMOD(n, MOD) *
63         ((InverseEuler(factMOD(r, MOD), MOD) *
64             InverseEuler(factMOD(n-r, MOD), MOD)) % MOD)) % MOD;
65 }
66
67 int main()
68 {
69     int n, r, p;
70     while (~scanf("%d%d%d", &n, &r, &p))
71     {
72         printf("%lld\n", C(n, r, p));
73     }
74 }

```

-fR0D

Written by fR0DDY

July 31, 2011 at 5:30 PM

Posted in [Algorithm](#)

Tagged with [algorithm](#), [binomial](#), [code](#), [combination](#), [Euler](#), [factorial](#), [inverse](#), [Lucas](#), [matrix](#), [multiplication](#), [recurrence](#), [theorem](#), [wilson](#)

Pollard Rho Brent Integer Factorization

with 10 comments

Pollard Rho is an integer factorization algorithm, which is quite fast for large numbers. It is based on Floyd's cycle-finding algorithm and on the observation that two numbers x and y are congruent modulo p with probability 0.5 after $1.177\sqrt{p}$ numbers have been randomly chosen.

Algorithm

Input : A number N to be factorized

Output : A divisor of N

```

If x mod 2 is 0
    return 2

```

Choose random x and c

y = x

g = 1

while g=1

x = f(x)

y = f(f(y))

g = gcd(x-y, N)

return g

Note that this algorithm may not find the factors and will return failure for composite n. In that case, use a different f(x) and try again. Note, as well, that this algorithm does not work when n is a prime number, since, in this case, d will be always 1. We choose $f(x) = x^2 + c$. Here's a python implementation :

```

1  def pollardRho(N):
2      if N%2==0:
3          return 2
4      x = random.randint(1, N-1)
5      y = x
6      c = random.randint(1, N-1)
7      g = 1
8      while g==1:
9          x = ((x*x)%N+c)%N
10         y = ((y*y)%N+c)%N
11         y = ((y*y)%N+c)%N
12         g = gcd(abs(x-y), N)
13     return g

```

In 1980, Richard Brent published a faster variant of the rho algorithm. He used the same core ideas as Pollard but a different method of cycle detection, replacing Floyd's cycle-finding algorithm with the related Brent's cycle finding method. It is quite faster than pollard rho. Here's a python implementation :

```

1  def brent(N):
2      if N%2==0:
3          return 2
4      y,c,m = random.randint(1, N-1), random.randint(1, N-1), random
5      g,r,q = 1,1,1
6      while g==1:
7          x = y
8          for i in range(r):
9              y = ((y*y)%N+c)%N
10         k = 0
11         while (k<r and g==1):
12             ys = y
13             for i in range(min(m, r-k)):
14                 y = ((y*y)%N+c)%N
15                 q = q*(abs(x-y))%N
16             g = gcd(q, N)
17             k = k + m
18         r = r*2
19     if g==N:
20         while True:

```

10/1/2015

Algorithm | COME ON CODE ON

```
21  
22  
23  
24  
25  
26  
ys = ((ys*ys)%N+c)%N  
g = gcd(abs(x-ys), N)  
if g>1:  
    break  
return g
```

-fR0DDY

Written by fR0DDY

September 18, 2010 at 11:51 PM

Posted in [Algorithm](#), [Programming](#)

Tagged with [algorithm](#), [brent](#), [cycle](#), [integer](#), [pollard](#), [rho](#)

Miller Rabin Primality Test

with 15 comments

Miller Rabin Primality Test is a probabilistic test to check whether a number is a prime or not. It relies on an equality or set of equalities that hold true for prime values, then checks whether or not they hold for a number that we want to test for primality.

Theory

- 1> Fermat's little theorem states that if p is a prime and $1 \leq a < p$ then $a^{p-1} \equiv 1 \pmod{p}$.
- 2> If p is a prime and $x^2 \equiv 1 \pmod{p}$ or $(x-1)(x+1) \equiv 0 \pmod{p}$ then $x \equiv 1 \pmod{p}$ or $x \equiv -1 \pmod{p}$.
- 3> If n is an odd prime then $n-1$ is an even number and can be written as $2^s \cdot d$. By Fermat's Little Theorem either $a^d \equiv 1 \pmod{n}$ or $a^{2^r \cdot d} \equiv -1 \pmod{n}$ for some $0 \leq r \leq s-1$.
- 4> The Miller-Rabin primality test is based on the contrapositive of the above claim. That is, if we can find an a such that $a^d \not\equiv 1 \pmod{n}$ and $a^{2^r \cdot d} \not\equiv -1 \pmod{n}$ for all $0 \leq r \leq s-1$ then a is witness of compositeness of n and we can say n is not a prime. Otherwise, n may be a prime.
- 5> We test our number N for some random a and either declare that N is definitely a composite or probably a prime. The probability that a composite number is returned as prime after k iterations is 4^{-k} .

Algorithm

Input :A number N to be tested and a variable iteration-the number of 'a' for which algorithm will test N.

Output :0 if N is definitely a composite and 1 if N is probably a prime.

Write N as $2^s \cdot d$

For each iteration

Pick a random a in [1,N-1]

$x = a^d \bmod n$

if $x = 1$ or $x = n-1$

Next iteration

for $r = 1$ to $s-1$

$x = x^2 \bmod n$

if $x = 1$

return false

if $x = N-1$

Next iteration

return false

return true

Here's a python implementation :

```

1  import random
2  def modulo(a,b,c):
3      x = 1
4      y = a
5      while b>0:
6          if b%2==1:
7              x = (x*y)%c
8              y = (y*y)%c
9              b = b/2
10     return x%c
11
12  def millerRabin(N,iteration):
13     if N<2:
14         return False
15     if N!=2 and N%2==0:
16         return False
17
18     d=N-1
19     while d%2==0:
20         d = d/2
21
22     for i in range(iteration):
23         a = random.randint(1, N-1)
24         temp = d
25         x = modulo(a,temp,N)
26         while (temp!=N-1 and x!=1 and x!=N-1):
27             x = (x*x)%N
28             temp = temp*2
29
30         if (x!=N-1 and temp%2==0):
31             return False
32
33     return True

```

-fR0DDY

Written by fR0DDY

September 18, 2010 at 12:23 PM

Posted in [Algorithm](#), [Programming](#)

Tagged with [algorithm](#), [code](#), [fermat](#), [little](#), [miller](#), [primality](#), [prime](#), [probability](#), [Python](#), [rabin](#), [test](#), [theorem](#)

Knuth–Morris–Pratt Algorithm (KMP)

with one comment

Knuth–Morris–Pratt algorithm is the most popular linear time algorithm for string matching. It is little difficult to understand and debug in real time contests. So most programmer's have a precoded KMP in their kitty.

To understand the algorithm, you can either read it from Introduction to Algorithms (CLRS) or from the wikipedia [page](#). Here's a sample C++ code.

```
1  void preKMP(string pattern, int f[])
2  {
3      int m = pattern.length(), k;
4      f[0] = -1;
5      for (int i = 1; i < m; i++)
6      {
7          k = f[i-1];
8          while (k >= 0)
9          {
10             if (pattern[k] == pattern[i-1])
11                 break;
12             else
13                 k = f[k];
14         }
15         f[i] = k + 1;
16     }
17 }
18
19 bool KMP(string pattern, string target)
20 {
21     int m = pattern.length();
22     int n = target.length();
23     int f[m];
24
25     preKMP(pattern, f);
26
27     int i = 0;
28     int k = 0;
29
30     while (i < n)
31     {
32         if (k == -1)
33         {
34             i++;
```


10/1/2015

Algorithm | COME ON CODE ON

```
35 k = 0;
36 }
37 else if (target[i]==pattern[k])
38 {
39     i++;
40     k++;
41     if (k==m)
42         return 1;
43 }
44 else
45     k=f[k];
46 }
47 return 0;
48 }
```

NJOY!

-fR0DDY

Written by fR0DDY

August 29, 2010 at 12:20 PM

Posted in [Algorithm](#), [Programming](#)

Tagged with [C](#), [clrs](#), [code](#), [Knuth](#), [linear](#), [matching](#), [morris](#), [pratt](#), [string](#), [time](#)

The Z Algorithm

with one comment

In this post we will discuss an algorithm for linear time string matching. It is easy to understand and code and is usefull in contests where you cannot copy paste code.

Let our string be denoted by S .

$z[i]$ denotes the length of the longest substring of S that starts at i and is a prefix of S .

α denotes the substring.

r denotes the index of the last character of α and l denotes the left end of α .

To find whether a pattern(P) of length n is present in a target string(T) of length m , we will create a new string $S = P\$T$ where $\$$ is a character present neither in P nor in T . The space taken is $n+m+1$ or $O(m)$. We will compute $z[i]$ for all i such that $0 < i < n+m+1$. If $z[i]$ is equal to n then we have found a occurrence of P at position $i - n - 1$. So we can all the occurrence of P in T in $O(m)$ time. To calculate $z[i]$ we will use the z algorithm.

The Z Algorithm can be read from the section 1.3-1.5 of book [Algorithms on strings, trees, and sequences](#) by Gusfield. Here is a sample C++ code.

```
1 bool zAlgorithm(string pattern, string target)
2 {
3     string s = pattern + '$' + target ;
4     int n = s.length();
```

```

5      vector<int> z(n,0);
6
7      int goal = pattern.length();
8      int r = 0, l = 0, i;
9      for (int k = 1; k<n; k++)
10     {
11         if (k>r)
12         {
13             for (i = k; i<n && s[i]==s[i-k]; i++);
14             if (i>k)
15             {
16                 z[k] = i - k;
17                 l = k;
18                 r = i - 1;
19             }
20         }
21         else
22         {
23             int kt = k - l, b = r - k + 1;
24             if (z[kt]>b)
25             {
26                 for (i = r + 1; i<n && s[i]==s[i-k]; i++);
27                 z[k] = i - k;
28                 l = k;
29                 r = i - 1;
30             }
31         }
32         if (z[k]==goal)
33             return true;
34     }
35     return false;
36 }

```

NJOY!

-fR0D

Written by fR0DDY

August 29, 2010 at 12:03 AM

Posted in [Algorithm](#), [Programming](#)

Tagged with [algorithm](#), [C](#), [code](#), [complexity](#), [linear](#), [matching](#), [string](#), [time](#)

All Pair Shortest Path (APSP)

with one comment

Question : Find shortest paths between all pairs of vertices in a graph.

Floyd-Warshall Algorithm

It is one of the easiest algorithms, and just involves simple dynamic programming. The algorithm can be read from [this](#) wikipedia page.

```

1  #define SIZE 31
2  #define INF 1e8
3  double dis[SIZE][SIZE];
4  void init(int N)
5  {
6      for (k=0;k<N;k++)
7          for (i=0;i<N;i++)
8              dis[i][j]=INF;
9  }
10 void floyd_warshall(int N)
11 {
12     int i,j,k;
13     for (k=0;k<N;k++)
14         for (i=0;i<N;i++)
15             for (j=0;j<N;j++)
16                 dis[i][j]=min(dis[i][j],dis[i][k]+
17                               dis[k][j]);
18 }
19 int main()
20 {
21     //input size N
22     init(N);
23     //set values for dis[i][j]
24     floyd_warshall(N);
25 }
```

We can also use the algorithm to

1. find the shortest path
 - we can use another matrix called predecessor matrix to construct the shortest path.
2. find negative cycles in a graph.
 - If the value of any of the diagonal elements is less than zero after calling the floyd-warshall algorithm then there is a negative cycle in the graph.
3. find transitive closure
 - to find if there is a path between two vertices we can use a boolean matrix and use and-& and or-| operators in the floyd_warshall algorithm.
 - to find the number of paths between any two vertices we can use a similar algorithm.

NJOY!!

-fR0DDY

Written by fR0DDY

August 7, 2010 at 1:53 PM

Posted in [Algorithm](#), [Programming](#)

Tagged with [all](#), [closure](#), [code](#), [cycle](#), [DP](#), [dynamic](#), [floyd](#), [graph](#), [negative](#), [pair](#), [path](#), [Programming](#), [shortest](#), [theory](#), [transitive](#), [warshall](#)

Number of Cycles in a Graph

with 2 comments

Question : Find the number of simple cycles in a simple graph.

Simple Graph – An undirected graph that has no loops and no more than one edge between any two different vertices.

Simple Cycle – A closed (simple) path, with no other repeated vertices or edges other than the starting and ending vertices.

Given a graph of N vertices and M edges, we will look at an algorithm with time complexity $O(2^N N^2)$. We will use dynamic programming to do so. Let there be a matrix map, such that $\text{map}[i][j]$ is equal to 1 if there is a edge between i and j and 0 otherwise. Let there be another array $f[1 \leq i \leq N][N]$ which denotes the number of simple paths.

Let,

i denote a subset S of our vertices

k be the smallest set bit of i

then $f[i][j]$ is the number of simple paths from j to k that contains vertices only from the set S .

In our algorithm first we will find $f[i][j]$ and then check if there is a edge between k and j , if yes, we can complete every simple path from j to k into a simple cycle and hence we add $f[i][j]$ to our result of total number of simple cycles. Now how to find $f[i][j]$.

For every subset i we iterate through all edges j . Once we have set k , we look for all vertices l that can be neighbors of j in our subset S . So if l is a vertex in subset S and there is edge from j to l then $f[i][j] = f[i][j] + \text{the number of simple paths from } l \text{ to } i \text{ in the subset } \{S - j\}$. Since a simple graph is undirected or bidirectional, we have counted every cycle twice and so we divide our result by 2. Here's a sample C++ code which takes N , M and the edges as input.

```

1  #include<iostream>
2  using namespace std;
3
4  #define SIZE 20
5
6  bool map[SIZE][SIZE],F;
7  long long f[1<=SIZE][SIZE],res=0;
8
9  int main()
10 {
11     int n,m,i,j,k,l,x,y;
12     scanf("%d%d",&n,&m);
13     for (i=0;i<m;i++)
14     {
15         scanf("%d%d",&x,&y);
16         x--;y--;
17         if (x>y)
18             swap(x,y);
19         map[x][y]=map[y][x]=1;
20         f[(1<=x)+(1<=y)][y]=1;

```

```
21     }
22
23     for (i=7;i<(1<<n);i++)
24     {
25         F=1;
26         for (j=0;j<n;j++)
27             if (i&(1<<j) && f[i][j]==0)
28             {
29                 if (F)
30                 {
31                     F=0;
32                     k=j;
33                     continue;
34                 }
35                 for (l=k+1;l<n;l++)
36                 {
37                     if (i&(1<<l) && map[j][l])
38                         f[i][j]+=f[i-(1<<j)][l];
39                 }
40                 if (map[k][j])
41                     res+=f[i][j];
42             }
43     }
44     printf("%lld\n",res/2);
45 }
```

NJOY!

-fR0DDY

Written by fR0DDY

June 7, 2010 at 1:14 PM

Posted in [Algorithm](#), [Programming](#)

Tagged with [code](#), [cycle](#), [dynamic](#), [graph](#), [Programming](#), [simple](#), [theory](#)

COME ON CODE ON

Create a free website or blog at WordPress.com. The Journalist v1.9 Theme.

1 Follow

Follow “COME ON CODE ON”

Build a website with WordPress.com