COME ON CODE ON

A blog about programming and more programming.

Archive for the '**Programming**' Category

Modular Multiplicative Inverse

with 28 comments

The modular multiplicative inverse of an integer a modulo m is an integer x such that $a^{-1} \equiv x \pmod{m}$.

That is, it is the multiplicative inverse in the ring of integers modulo m. This is equivalent to $ax \equiv aa^{-1} \equiv 1 \pmod{m}$.

The multiplicative inverse of a modulo m exists if and only if a and m are coprime (i.e., if gcd(a, m) = 1).

Let's see various ways to calculate Modular Multiplicative Inverse:

1. Brute Force
We can calculate the inverse using a brute force approach where we multiply *a* with all possible values *x* and find a *x* such that $ax \equiv 1 \pmod{m}$. Here's a sample C++ code:

```
int modInverse(int a, int m) {
    a %= m;
    for(int x = 1; x < m; x++) {
        if((a*x) % m == 1) return x;
    }
}
```

The time complexity of the above codes is O(m).

2. Using Extended Euclidean Algorithm
We have to find a number x such that a·x = 1 (mod m). This can be written as well as a·x = 1 + m·y, which rearranges into a·x – m·y = 1. Since x and y need not be positive, we can write it as well in the standard form, a·x + m·y = 1.

In number theory, Bézout's identity for two integers a, b is an expression ax + by = d, where x and y are integers (called Bézout coefficients for (a,b)), such that d is a common divisor of a and b. If d is the greatest common divisor of a and b then Bézout's identity ax + by = gcd(a,b) can be solved using Extended Euclidean Algorithm.

The Extended Euclidean Algorithm is an extension to the Euclidean algorithm. Besides finding the greatest common divisor of integers a and b, as the Euclidean algorithm does, it also finds integers x and y (one of which is typically negative) that satisfy Bézout's identity
ax + by = gcd(a,b). The Extended Euclidean Algorithm is particularly useful when a and b are coprime, since x is the multiplicative inverse of a modulo b, and y is the multiplicative inverse of b modulo a.

We will look at two ways to find the result of Extended Euclidean Algorithm.

Iterative Method

This method computes expressions of the form $r_i = ax_i + by_i$ for the remainder in each step i of the Euclidean algorithm. Each successive number r_i can be written as the remainder of the division of the previous two such numbers, which remainder can be expressed using the whole quotient q_i of that division as follows:

$r_i = r_{i-2} - q_i r_{i-1}.$

By substitution, this gives:

$r_i = (ax_{i-2} + by_{i-2}) - q_i(ax_{i-1} + by_{i-1})$, which can be written

$r_i = a(x_{i-2} - q_i x_{i-1}) + b(y_{i-2} - q_i y_{i-1}).$

The first two values are the initial arguments to the algorithm:

$r_1 = a = a \times 1 + b \times 0$

$r_2 = b = a \times 0 + b \times 1.$

So the coefficients start out as $x_1 = 1$, $y_1 = 0$, $x_2 = 0$, and $y_2 = 1$, and the others are given by

$x_i = x_{i-2} - q_i x_{i-1},$

$y_i = y_{i-2} - q_i y_{i-1}.$

The expression for the last non-zero remainder gives the desired results since this method computes every remainder in terms of a and b, as desired.

So the algorithm looks like,

1. Apply Euclidean algorithm, and let qn(n starts from 1) be a finite list of quotients in the division.
2. Initialize x_0, x_1 as 1, 0, and y_0, y_1 as 0,1 respectively.
    1. Then for each i so long as q_i is defined,
    2. Compute $x_{i+1} = x_{i-1} - q_i x_i$
    3. Compute $y_{i+1} = y_{i-1} - q_i y_i$
    4. Repeat the above after incrementing i by 1.
3. The answers are the second-to-last of x_n and y_n.

```
/* This function return the gcd of a and b followed by
    the pair x and y of equation ax + by = gcd(a,b)*/
pair<int, pair<int, int> > extendedEuclid(int a, int b) {
    int x = 1, y = 0;
    int xLast = 0, yLast = 1;
    int q, r, m, n;
    while(a != 0) {
        q = b / a;
        r = b % a;
        m = xLast - q * x;
        n = yLast - q * y;
        xLast = x, yLast = y;
        x = m, y = n;
        b = a, a = r;
    }
    return make_pair(b, make_pair(xLast, yLast));
}

```

```
19 int modInverse(int a, int m) {
20     return (extendedEuclid(a,m).second.first + m) % m;
21 }
```

Recursive Method

This method attempts to solve the original equation directly, by reducing the dividend and divisor gradually, from the first line to the last line, which can then be substituted with trivial value and work backward to obtain the solution.

Notice that the equation remains unchanged after decomposing the original dividend in terms of the divisor plus a remainder, and then regrouping terms. So the algorithm looks like this:

1. If b = 0, the algorithm ends, returning the solution x = 1, y = 0.
2. Otherwise:
   - Determine the quotient q and remainder r of dividing a by b using the integer division algorithm.
   - Then recursively find coefficients s, t such that bs + rt divides both b and r.
   - Finally the algorithm returns the solution x = t, and y = s − qt.

Here's a C++ implementation:

```
1  /* This function return the gcd of a and b followed by
2      the pair x and y of equation ax + by = gcd(a,b)*/
3  pair<int, pair<int, int> > extendedEuclid(int a, int b) {
4      if(a == 0) return make_pair(b, make_pair(0, 1));
5      pair<int, pair<int, int> > p;
6      p = extendedEuclid(b % a, a);
7      return make_pair(p.first, make_pair(p.second.second - p.second
8  }
9
10 int modInverse(int a, int m) {
11     return (extendedEuclid(a,m).second.first + m) % m;
12 }
```

The time complexity of the above codes is $O(log(m)^2)$.

3. Using Fermat's Little Theorem

Fermat's little theorem states that if m is a prime and a is an integer co-prime to m, then $a^p - 1$ will be evenly divisible by m. That is $a^{m-1} \equiv 1 \pmod{m}$. or $a^{m-2} \equiv a^{-1} \pmod{m}$. Here's a sample C++ code:

```
1  /* This function calculates (a^b)%MOD */
2  int pow(int a, int b, int MOD) {
3  int x = 1, y = a;
4      while(b > 0) {
5          if(b%2 == 1) {
6              x=(x*y);
7              if(x>MOD) x%=MOD;
8          }
9          y = (y*y);
10         if(y>MOD) y%=MOD;
11         b /= 2;
12     }
13     return x;
14 }
15
16 int modInverse(int a, int m) {
```

```
17          return pow(a,m-2,m);
18      }
```

The time complexity of the above codes is O(log(m)).

4. Using Euler's Theorem
Fermat's Little theorem can only be used if m is a prime. If m is not a prime we can use Euler's Theorem, which is a generalization of Fermat's Little theorem. According to Euler's theorem, if a is coprime to m, that is, gcd(a, m) = 1, then $a^{\varphi(m)} \equiv 1 \pmod{m}$, where where φ(m) is Euler Totient Function. Therefore the modular multiplicative inverse can be found directly: $a^{\varphi(m)-1} \equiv a^{-1} \pmod{m}$. The problem here is finding φ(m). If we know φ(m), then it is very similar to above method.

Now lets take a little different question. Now suppose you have to calculate the inverse of first n numbers. From above the best we can do is O(n log(m)). Can we do any better? Yes.

We can use sieve to find a factor of composite numbers less than n. So for composite numbers inverse(i) = (inverse(i/factor(i)) * inverse(factor(i))) % m, and we can use either Extended Euclidean Algorithm or Fermat's Theorem to find inverse for prime numbers. But we can still do better.

a * (m / a) + m % a = m
(a * (m / a) + m % a) mod m = m mod m, or
(a * (m / a) + m % a) mod m = 0, or
(- (m % a)) mod m = (a * (m / a)) mod m.
Dividing both sides by (a * (m % a)), we get
– inverse(a) mod m = ((m/a) * inverse(m % a)) mod m
inverse(a) mod m = (- (m/a) * inverse(m % a)) mod m

Here's a sample C++ code:

```
1   vector<int> inverseArray(int n, int m) {
2       vector<int> modInverse(n + 1,0);
3       modInverse[1] = 1;
4       for(int i = 2; i <= n; i++) {
5           modInverse[i] = (-(m/i) * modInverse[m % i]) % m + m;
6       }
7       return modInverse;
8   }
```

The time complexity of the above code is O(n).

-fR0DDY

Written by fR0DDY

October 9, 2011 at 12:29 AM

Posted in Programming

Tagged with algorithm, C, code, euclidean, Euler, fermat, inverse, little, modular, multiplicative, theorem

Recurrence Relation and Matrix Exponentiation

with 14 comments

Recurrence relations appear many times in computer science. Using recurrence relation and dynamic programming we can calculate the n^{th} term in O(n) time. But many times we need to calculate the n^{th} in O(log n) time. This is where Matrix Exponentiation comes to rescue.

We will specifically look at linear recurrences. A linear recurrence is a sequence of vectors defined by the equation $X_{i+1} = M\ X_i$ for some constant matrix M. So our aim is to find this constant matrix M, for a given recurrence relation.

Let's first start by looking at the common structure of our three matrices X_{i+1}, X_i and M. For a recurrence relation where the next term is dependent on last k terms, X_{i+1} and X_{i+1} are matrices of size 1 x k and M is a matrix of size k x k.

```
| f(n+1)   |       | f(n)   |
|  f(n)    |       | f(n-1) |
| f(n-1)   | = M x | f(n-2) |
| ......   |       | ...... |
| f(n-k+1) |       | f(n-k) |
```

Let's look at different type of recurrence relations and how to find M.

1. Let's start with the most common recurrence relation in computer science, The Fibonacci Sequence. $F_{i+1} = F_i + F_{i-1}$.

```
| f(n+1) | = M x | f(n)   |
| f(n)   |       | f(n-1) |
```

Now we know that M is a 2 x 2 matrix. Let it be

```
| f(n+1) | = | a b | x | f(n)   |
| f(n)   |   | c d |   | f(n-1) |
```

Now a*f(n) + b*f(n-1) = f(n+1) and c*f(n) + d*f(n-1) = f(n). Solving these two equations we get a=1,b=1,c=1 and d=0. So,

```
| f(n+1) | = | 1 1 | x | f(n)   |
| f(n)   |   | 1 0 |   | f(n-1) |
```

2. For recurrence relation f(n) = a*f(n-1) + b*f(n-2) + c*f(n-3), we get

```
| f(n+1) | = | a b c | x | f(n)   |
| f(n)   |   | 1 0 0 |   | f(n-1) |
| f(n-1) |   | 0 1 0 |   | f(n-2) |
```

3. What if the recurrence relation is f(n) = a*f(n-1) + b*f(n-2) + c, where c is a constant. We can also add it in the matrices as a state.

```
| f(n+1) | = | a b 1 | x | f(n)   |
| f(n)   |   | 1 0 0 |   | f(n-1) |
| c      |   | 0 0 1 |   | c      |
```

4. If a recurrence relation is given like this f(n) = f(n-1) if n is odd, f(n-2) otherwise, we can convert it to f(n) = (n&1) * f(n-1) + (!(n&1)) * f(n-2) and substitute the value accordingly in the matrix.

5.If there are more than one recurrence relation, g(n) = a*g(n-1) + b*g(n-2) + c*f(n), and, f(n) = d*f(n-1) + e*f(n-2). We can still define the matrix X in following way

```
| g(n+1) |   | a b c 0 |   | g(n)   |
| g(n)   | = | 1 0 0 0 | x | g(n-1) |
| f(n+2) |   | 0 0 d e |   | f(n+1) |
| f(n+1) |   | 0 0 1 0 |   | f(n)   |
```

Now that we have got our matrix M, how are we going to find the nth term.

X_{i+1} = M X_i

(Multiplying M both sides)

M * X_{i+1} = M * M X_i

X_{i+2} = M^2 X_i

..

X_{i+k} = M^k X_i

So all we need now is to find the matrix M^k to find the k-th term. For example in the case of Fibonacci Sequence,

```
M^2 = | 2 1 |
      | 1 1 |
```

Hence F(2) = 2.

Now we need to learn to find M^k in $O(n^3 \log b)$ time. The brute force approach to calculate a^b takes O(b) time, but using a recursive divide-and-conquer algorithm takes only O(log b) time:

- If b = 0, then the answer is 1.
- If b = 2k is even, then $a^b = (a^k)^2$.
- If b is odd, then $a^b = a * a^{b-1}$.

We take a similar approach for Matrix Exponentiation. The multiplication part takes the $O(n^3)$ time and hence the overall complexity is $O(n^3 \log b)$. Here's a sample code in C++ using template class:

```
#include<iostream>
using namespace std;

template< class T >
class Matrix
{
    public:
        int m,n;
        T *data;

        Matrix( int m, int n );
        Matrix( const Matrix< T > &matrix );

        const Matrix< T > &operator=( const Matrix< T > &A );
        const Matrix< T > operator*( const Matrix< T > &A );
        const Matrix< T > operator^( int P );

        ~Matrix();
};

```

```
21  template< class T >
22  Matrix< T >::Matrix( int m, int n )
23  {
24      this->m = m;
25      this->n = n;
26      data = new T[m*n];
27  }
28
29  template< class T >
30  Matrix< T >::Matrix( const Matrix< T > &A )
31  {
32      this->m = A.m;
33      this->n = A.n;
34      data = new T[m*n];
35      for( int i = 0; i < m * n; i++ )
36          data[i] = A.data[i];
37  }
38
39  template< class T >
40  Matrix< T >::~Matrix()
41  {
42      delete [] data;
43  }
44
45  template< class T >
46  const Matrix< T > &Matrix< T >::operator=( const Matrix< T > &A )
47  {
48      if( &A != this )
49      {
50          delete [] data;
51          m = A.m;
52          n = A.n;
53          data = new T[m*n];
54          for( int i = 0; i < m * n; i++ )
55              data[i] = A.data[i];
56      }
57      return *this;
58  }
59
60  template< class T >
61  const Matrix< T > Matrix< T >::operator*( const Matrix< T > &A )
62  {
63      Matrix C( m, A.n );
64      for( int i = 0; i < m; ++i )
65          for( int j = 0; j < A.n; ++j )
66          {
67              C.data[i*C.n+j]=0;
68              for( int k = 0; k < n; ++k )
69                  C.data[i*C.n+j] = C.data[i*C.n+j] + data[i*n+k]*A.
70          }
71      return C;
72  }
73
74  template< class T >
75  const Matrix< T > Matrix< T >::operator^( int P )
76  {
77      if( P == 1 ) return (*this);
78      if( P & 1 ) return (*this) * ((*this) ^ (P-1));
79      Matrix B = (*this) ^ (P/2);
```

```
80      return B*B;
81  }
82
83  int main()
84  {
85      Matrix<int> M(2,2);
86      M.data[0] = 1;M.data[1] = 1;
87      M.data[2] = 1;M.data[3] = 0;
88
89      int F[2]={0,1};
90      int N;
91      while (~scanf("%d",&N))
92          if (N>1)
93              printf("%lld\n",(M^N).data[0]);
94          else
95              printf("%d\n",F[N]);
96  }
```

Written by fR0DDY

May 8, 2011 at 5:26 PM

Posted in Programming

Tagged with algorithm, code, complexity, divide-and-conquer, equation, exponentiation, Fibonacci, matrix, recurrence, relation, time

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*x + c. Here's a python implementation :

```
def pollardRho(N):
        if N%2==0:
                return 2
        x = random.randint(1, N-1)
        y = x
        c = random.randint(1, N-1)
        g = 1
        while g==1:
                x = ((x*x)%N+c)%N
                y = ((y*y)%N+c)%N
                y = ((y*y)%N+c)%N
                g = gcd(abs(x-y),N)
        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 :

```
def brent(N):
        if N%2==0:
                return 2
        y,c,m = random.randint(1, N-1),random.randint(1, N-1),rand
        g,r,q = 1,1,1
        while g==1:
                x = y
                for i in range(r):
                        y = ((y*y)%N+c)%N
                k = 0
                while (k<r and g==1):
                        ys = y
                        for i in range(min(m,r-k)):
                                y = ((y*y)%N+c)%N
                                q = q*(abs(x-y))%N
                        g = gcd(q,N)
                        k = k + m
                r = r*2
        if g==N:
                while True:
                        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 ≤ 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 $2^s.d$. 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 ≤ r ≤ 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 ≤ r ≤ 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 probably that a composite number is returned as prime after k itereations 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.d
For each iteration
        Pick a random a in [1,N-1]
        x = a^d mod n
        if x =1 or x = n-1
                Next iteration
        for r = 1 to s-1
                x  = x^2 mod 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
```

```
        y = a
        while b>0:
                if b%2==1:
                        x = (x*y)%c
                y = (y*y)%c
                b = b/2
        return x%c

def millerRabin(N,iteration):
        if N<2:
                return False
        if N!=2 and N%2==0:
                return False

        d=N-1
        while d%2==0:
                d = d/2

        for i in range(iteration):
                a = random.randint(1, N-1)
                temp = d
                x = modulo(a,temp,N)
                while (temp!=N-1 and x!=1 and x!=N-1):
                        x = (x*x)%N
                        temp = temp*2

                if (x!=N-1 and temp%2==0):
                        return False

        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++;
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.

```
bool zAlgorithm(string pattern, string target)
{
    string s = pattern + '$' + target ;
    int n = s.length();
    vector<int> z(n,0);

    int goal = pattern.length();
    int r = 0, l = 0, i;
    for (int k = 1; k<n; k++)
    {
        if (k>r)
        {
            for (i = k; i<n && s[i]==s[i-k]; i++);
            if (i>k)
            {
                z[k] = i - k;
                l = k;
                r = i - 1;
            }
        }
        else
        {
            int kt = k - l, b = r - k + 1;
            if (z[kt]>b)
            {
                for (i = r + 1; i<n && s[i]==s[i-k]; i++);
                z[k] = i - k;
                l = k;
                r = i - 1;
```

```
            }
        }
        if (z[k]==goal)
            return true;
    }
    return false;
}
```

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.

```
#define SIZE 31
#define INF 1e8
double dis[SIZE][SIZE];
void init(int N)
{
    for (k=0;k<N;k++)
        for (i=0;i<N;i++)
            dis[i][j]=INF;
}
void floyd_warshall(int N)
{
        int i,j,k;
        for (k=0;k<N;k++)
                for (i=0;i<N;i++)
                        for (j=0;j<N;j++)
                                dis[i][j]=min(dis[i][j],dis[i][k]+
}

int main()
{
    //input size N
    init(N);
    //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 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<<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 very 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] + the number of simple paths from l to i 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.

```
#include<iostream>
using namespace std;

#define SIZE 20

bool map[SIZE][SIZE],F;
long long f[1<<SIZE][SIZE],res=0;

int main()
{
    int n,m,i,j,k,l,x,y;
    scanf("%d%d",&n,&m);
    for (i=0;i<m;i++)
    {
        scanf("%d%d",&x,&y);
        x--;y--;
        if (x>y)
           swap(x,y);
        map[x][y]=map[y][x]=1;
        f[(1<<x)+(1<<y)][y]=1;
    }

    for (i=7;i<(1<<n);i++)
    {
        F=1;
        for (j=0;j<n;j++)
            if (i&(1<<j) && f[i][j]==0)
            {
               if (F)
               {
                  F=0;
                  k=j;
                  continue;
               }
               for (l=k+1;l<n;l++)
               {
                   if (i&(1<<l) && map[j][l])
                      f[i][j]+=f[i-(1<<j)][l];
               }
               if (map[k][j])
                  res+=f[i][j];
            }
    }
    printf("%lld\n",res/2);
}
```

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

Longest Common Increasing Subsequence (LCIS)

with 4 comments

Given 2 sequences of integers, we need to find a longest sub-sequence which is common to both the sequences, and the numbers of such a sub-sequence are in strictly increasing order.
For example,
2 3 1 6 5 4 6
1 3 5 6
the LCIS is 3 5 6.

The sequence $a_1, a_2, \ldots, a_n$ is called increasing, if $a_i<a_{i+1}$ for $i<n$.The sequence $s_1, s_2, \ldots, s_k$ is called the subsequence of the sequence $a_1, a_2, \ldots, a_n$, if there exist such a set of indexes $1 \le i_1 < i_2 < \ldots < i_k \le n$ that $a_{i_j} = s_j$. In other words, the sequence s can be derived from the sequence a by crossing out some elements.

A nice tutorial on the algorithm is given on CodeChef blog here. If you read the blog you can see that instead of looking for LCIS in a candidate matrix, we can keep an array that stores the length of LCIS ending at that particular element. Also we keep a lookup previous array that gives the index of the previous element of the LCIS, which is used to reconstruct the LCIS.

For every element in the two arrays
1> If they are equal we check whether the LCIS formed will be bigger than any previous such LCIS ending at that element. If yes we change the data.
2> If the jth element of array B is smaller than ith element of array A, we check whether it has a LCIS greater than current LCIS length, if yes we store it as previous value and it’s LCIS length as current LCIS length.

Here’s a C++ code.

```
#include<iostream>
using namespace std;
#include<vector>

void LCIS(vector<int> A, vector<int> B)
{
    int N=A.size(),M=B.size(),i,j;
    vector<int> C(M,0);
    vector<int> prev(M,0);
    vector<int> res;

    for (i=0;i<N;i++)
    {
        int cur=0,last=-1;
        for (j=0;j<M;j++)
        {
            if (A[i]==B[j] && cur+1>C[j])
            {
               C[j]=cur+1;
               prev[j]=last;
            }
            if (B[j]<A[i] && cur<C[j])
            {
               cur=C[j];
               last=j;
            }
        }
    }

    int length=0,index=-1;
    for (i=0;i<M;i++)
        if (C[i]>length)
        {
           length=C[i];
           index=i;
        }
    printf("The length of LCIS is %d\n",length);
    if (length>0)
    {
       printf("The LCIS is \n");
       while (index!=-1)
       {
             res.push_back(B[index]);
             index=prev[index];
       }
       reverse(res.begin(),res.end());
       for (i=0;i<length;i++)
           printf("%d%s",res[i],i==length-1?"\n":" ");
    }
}
```

```
int main()
{
    int n,m,i;
    scanf ("%d", &n);
    vector<int> A(n,0);
    for (i = 0; i < n; i++)
        scanf ("%d", &A[i]);
    scanf ("%d", &m);
    vector<int> B(m,0);
    for (i = 0; i < m; i++)
        scanf ("%d", &B[i]);
    LCIS(A,B);
}
```

NJOY!
-fR0DDY

Written by fR0DDY

June 1, 2010 at 12:47 PM

Posted in Algorithm, Programming

Tagged with C, code, common, increasing, lcis, longest, subsequence

Number of Binary Trees

with 4 comments

Question : What is the number of rooted plane binary trees with n nodes and height h?

Solution :
Let $t_{n,h}$ denote the number of binary trees with n nodes and height h.

Lets take a binary search tree of n nodes with height equal to h. Let the number written at its root be the m^{th}number in sorted order, $1 \le m \le n$. Therefore, the left subtree is a binary search tree on m-1 nodes, and the right subtree is a binary search tree on n-m nodes. The maximum height of the left and the right subtrees must be equal to h-1. Now there are two cases :

1. The height of left subtree is h-1 and the height of right subtree is less than equal to h-1 i.e from 0 to h-1.
2. The height of right subtree is h-1 and the height of left subtree is less than equal to h-2 i.e from 0 to h-2, since we have considered the case when the left and right subtrees have the same height h-1 in case 1.

Therefore $t_{n,h}$ is equal to the sum of number of trees in case 1 and case 2. Let’s find the number of trees in case 1 and case 2.

1. The height of the left subtree is equal to h-1. There are $t_{m-1,h-1}$ such trees. The right subtree can have any height from 0 to h-1, so there are $\sum_{i=0}^{h-1} t_{n-m,i}$ such trees. Therefore the total number of such tress are $t_{m-1,h-1}\sum_{i=0}^{h-1} t_{n-m,i}$.
2. The height of the right subtree is equal to h-1. There are $t_{n-m,h-1}$ such trees. The left subtree can have any height from 0 to h-2, so there are $\sum_{i=0}^{h-2} t_{m-1,i}$ such trees. Therefore the total number of such tress are $t_{n-m,h-1}\sum_{i=0}^{h-2} t_{m-1,i}$.

Hence we get the recurrent relation

$$t_{n,h}=\sum_{m=1}^{n}\left(t_{m-1,h-1}\sum_{i=0}^{h-1}t_{n-m,i}\right)+\sum_{m=1}^{n}\left(t_{n-m,h-1}\sum_{i=0}^{h-2}t_{m-1,i}\right)$$

or

$$t_{n,h}=\sum_{m=1}^{n}\left(t_{m-1,h-1}\sum_{i=0}^{h-1}t_{n-m,i}+t_{n-m,h-1}\sum_{i=0}^{h-2}t_{m-1,i}\right)$$

where $t_{0,0}=1$ and $t_{0,i}=t_{i,0}=0$ for i>0.

Here's a sample C++ code.

```
#include<iostream>
using namespace std;

#define MAXN 35

int main()
{
    long long t[MAXN+1][MAXN+1]={0},n,h,m,i;

    t[0][0]=1;
    for (n=1;n<=MAXN;n++)
        for (h=1;h<=n;h++)
            for (m=1;m<=n;m++)
            {
                for (i=0;i<h;i++)
                    t[n][h]+=t[m-1][h-1]*t[n-m][i];

                for (i=0;i<h-1;i++)
                    t[n][h]+=t[n-m][h-1]*t[m-1][i];
            }

    while (scanf("%lld%lld",&n,&h))
          printf("%lld\n",t[n][h]);
}
```

Note :

1. The total number of binary trees with n nodes is $\frac{2n!}{n!(n+1)!}$, also known as Catalan numbers or Segner

numbers.

2. $t_{n,n} = 2^{n-1}$.
3. The number of trees with height not lower than h is $\sum_{i=h}^{n} t_{n,i}$.
4. There are other recurrence relation as well such as $t_{n,h} = \sum_{i=0}^{h-1} (t_{n-1,h-1-i} (2 * \sum_{j=0}^{n-2} t_{j,i} + t_{n-1,i}))$.

NJOY!
-fR0DDY

Written by fR0DDY

April 13, 2010 at 11:44 AM

Posted in Algorithm, Maths, Programming

Tagged with Binary, C, code, height, node, Tree

Convert a number from decimal base to any Base

with 6 comments

Convert a given decmal number to any other base (either positive or negative).

For example, 100 in Base 2 is 1100100 and in Base -2 is 110100100.

Here's a simple algorithm to convert numbers from Decimal to Negative Bases :

```
def tonegativeBase(N,B):
   digits = []
   while i != 0:
        i, remainder = divmod (i, B)
        if (remainder < 0):
            i, remainder = i + 1, remainder + B*-1
        digits.insert (0, str (remainder))
   return ''.join (digits)
```

We can just tweak the above algorithm a bit to convert a decimal to any Base. Here's a sample code :

```
#include<iostream>
using namespace std;

void convert10tob(int N,int b)
{
     if (N==0)
        return;

     int x = N%b;
     N/=b;
     if (x<0)
        N+=1;

     convert10tob(N,b);
     printf("%d",x<0?x+(b*-1):x);
     return;
}


int main()
{
    int N,b;
    while (scanf("%d%d",&N,&b)==2)
    {
          if (N!=0)
          {
              convert10tob(N,b);
              printf("\n");
          }
          else
              printf("0\n");
    }
}
```

NJOY!
-fR0DDY

Written by fR0DDY

February 17, 2010 at 6:06 PM

Posted in Algorithm, Maths, Programming

Tagged with base, C, code, decimal, negative, number

Number of zeores and digits in N Factorial in Base B

with 2 comments

Question : What is the number of trailing zeroes and the number of digits in N factorial in Base B.

For example,
20! can be written as 2432902008176640000 in decimal number system while it is equal to "2070331676202550000 00" in octal number system and "21C3677C82B40000" in hexadecimal number system. That means that 10 factorial has 4 trailing zeroes in Base 10 while it has 6 trailing zeroes in Base 8 and 4 trailing zeroes in Base 16. Also 10 factorial has 19 digits in Base 10 while it has 21 digits in Base 8 and 16 digits in Base 16. Now the question remains how to find it?

Now we can break the Base B as a product of primes :
B = $a^{p1} * b^{p2} * c^{p3} * \ldots$
Then the number of trailing zeroes in N factorial in Base B is given by the formulae
min{1/p1(n/a + n/(a*a) + ….), 1/p2(n/b + n/(b*b) + ..), ….}.

And the number of digits in N factorial is :
floor (ln(n!)/ln(B) + 1)

Here's a sample C++ code :

```
#include<iostream>
using namespace std;
#include<math.h>

int main()
{
    int N,B,i,j,p,c,noz,k;
    while (scanf("%d%d",&N,&B)!=EOF)
    {
          noz=N;
          j=B;
          for (i=2;i<=B;i++)
          {
              if (j%i==0)
              {
                 p=0;
                 while (j%i==0)
                 {
                       p++;
                       j/=i;
                 }
                 c=0;
                 k=N;
                 while (k/i>0)
                 {
                       c+=k/i;
                       k/=i;
                 }
                 noz=min(noz,c/p);
              }
          }
          double ans=0;
          for (i=1;i<=N;i++)
          {
              ans+=log(i);
          }
          ans/=log(B);ans+=1.0;
          ans=floor(ans);
          printf("%d %.0lf\n",noz,ans);
    }
}
```

NJOY!
-fR0DDY

Written by fR0DDY

February 17, 2010 at 5:47 PM

Posted in Maths, Programming

Tagged with base, C, code, digits, factorial, number, zeroes

Number of Distinct LCS

with 2 comments

This problem appeared in the November edition of CodeChef Monthly Contest. The problem is to find the number of distinct LCS that can be formed from the given two strings. A nice tutorial about it is given on the CodeChef site itself. Also there is a research paper available here. Here's my solution to the CodeChef problem.

```
#include<iostream>
using namespace std;

int L[1005][1005];
int D[1005][1005];

int LCS(string X,string Y)
{
     int m = X.length(),n=Y.length();

     int i,j;
     for (i=0;i<=m;i++)
     {
         for (j=0;j<=n;j++)
         {
             if (i==0 || j==0)
                L[i][j]=0;
             else
             {
                 if (X[i-1]==Y[j-1])
                    L[i][j]=L[i-1][j-1]+1;
                 else
                     L[i][j]=max(L[i][j-1],L[i-1][j]);
             }
         }
     }
     return (L[m][n]);
}

int NLCS(string X,string Y)
{
    int m = X.length(),n=Y.length();
    int i,j;
    for (i=0;i<=m;i++)
    {
        for (j=0;j<=n;j++)
        {
             if (i==0 || j==0)
                D[i][j]=1;
             else
             {
                 D[i][j]=0;
                 if (X[i-1]==Y[j-1])
                 {
                    D[i][j]=D[i-1][j-1];
                 }
                 else
                 {
                     if (L[i-1][j]==L[i][j])
                        D[i][j]=(D[i][j]+D[i-1][j])%23102009;
```

```
                        if (L[i][j-1]==L[i][j])
                           D[i][j]=(D[i][j]+D[i][j-1])%23102009;
                        if (L[i-1][j-1]==L[i][j])
                        {
                           if (D[i][j]<D[i-1][j-1])
                              D[i][j]+=23102009;
                           D[i][j]=(D[i][j]-D[i-1][j-1]);
                        }
                     }
                  }
               }
            }
            return (D[m][n]);
}

int main()
{
    string X,Y;
    int T,A,B;
    scanf("%d",&T);
    while (T--)
    {
          cin>>X>>Y;
          A=LCS(X,Y);
          B=NLCS(X,Y);
          printf("%d %d\n",A,B);
    }
}
```

NJOY!
-fR0DDY

Written by fR0DDY

November 13, 2009 at 1:06 PM

Posted in Programming

Tagged with C, code, distinct, dynamic, LCS, number, Programming

Binary Indexed Tree (BIT)

with 20 comments

In this post we will discuss the Binary Indexed Trees structure. According to Peter M. Fenwick, this structure was first used for data compression. Let’s define the following problem: We have n boxes. Possible queries are

1. Add marble to box i
2. Return sum of marbles from box k to box l

The naive solution has time complexity of O(1) for query 1 and O(n) for query 2. Suppose we make m queries. The worst case (when all queries are 2) has time complexity O(n * m). Binary Indexed Trees are easy to code and have worst time complexity O(m log n).

The two major functions are

- update (idx,val) : increases the frequency of index idx with the value val
- read (idx) : reads the cumulative frequency of index idx

Note : tree[idx] is sum of frequencies from index (idx – 2^r + 1) to index idx where r is rightmost position of 1 in the binary notation of idx, f is frequency of index, c is cumulative frequency of index, tree is value stored in tree data structure.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
f	1	0	2	1	1	3	0	4	2	5	2	2	3	1	0	2
c	1	1	3	4	5	8	8	12	14	19	21	23	26	27	27	29
tree	1	1	2	4	1	4	0	12	2	7	2	11	3	4	0	29

Table 1.1

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
tree	1	1..2	3	1..4	5	5..6	7	1..8	9	9..10	11	9..12	13	13..14	15	1..16

Table 1.2 – table of responsibility

Here’s a C++ template code :

```
#include<iostream>
using namespace std;

template<class T>
class BIT
{
      T *tree;
      int maxVal;
      public:
      BIT(int N)
      {
              tree = new T[N+1];
              memset(tree,0,sizeof(T)*(N+1));
              maxVal = N;
      }
      void update(int idx, T val)
      {
           while (idx <= maxVal)
           {
                 tree[idx] += val;
                 idx += (idx & -idx);
           }
      }
      //Returns the cumulative frequency of index idx
      T read(int idx)
      {
        T sum=0;
        while (idx>0)
        {
              sum += tree[idx];
              idx -= (idx & -idx);
        }
        return sum;
      }
};

int main()
{
    int a[100],cur=1,mul=2,add=19,MAX=65536,x,i;
    //Initialize the size by the
    //maximum value the tree can have
    BIT<int> B(MAX);
    for (i=0;i<50;i++)
    {
        a[i] = cur;
        B.update(a[i],1);
        cur = ((cur * mul + add) % MAX);
    }
    while (cin>>x)
    {
```

```
            cout<<B.read(x)<<endl;
    }

}
```

Resources:

1. Topcoder Tutorial

NJOY!
-fR0D

Written by fR0DDY

September 17, 2009 at 11:40 PM

Posted in Programming

Tagged with advanced, Binrary, bit, C, code, complexity, data, Indexed, logarithmic, strucutre, time, Tree

COME ON CODE ON