COME ON CODE ON

A blog about programming and more programming.

Modular Multiplicative Inverse

with 17 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));
}

int modInverse(int a, int m) {
    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:

```
/* 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) {
    if(a == 0) return make_pair(b, make_pair(0, 1));
    pair<int, pair<int, int> > p;
    p = extendedEuclid(b % a, a);
    return make_pair(p.first, make_pair(p.second.second - p.second.fir
}

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

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:

```
/* This function calculates (a^b)%MOD */
int pow(int a, int b, int MOD) {
int x = 1, y = a;
    while(b > 0) {
        if(b%2 == 1) {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

int modInverse(int a, int m) {
```

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

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:

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

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

Combination

with 12 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)!)

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

long long C(int n, int r)
{
    long long f[n + 1];
    f[0]=1;
    for (int i=1;i<=n;i++)
        f[i]=i*f[i-1];
    return f[n]/f[r]/f[n-r];
}

int main()
{
    int n,r,m;
    while (~scanf("%d%d",&n,&r))
    {
        printf("%lld\n",C(n, min(r,n-r)));
    }
}
```

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

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

/* This function calculates (a^b)%MOD */
long long pow(int a, int b, int MOD)
{
    long long x=1,y=a;
    while(b > 0)
    {
```

```
        if(b%2 == 1)
        {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

/*  Modular Multiplicative Inverse
    Using Euler's Theorem
    a^(phi(m)) = 1 (mod m)
    a^(-1) = a^(m-2) (mod m) */
long long InverseEuler(int n, int MOD)
{
    return pow(n,MOD-2,MOD);
}

long long C(int n, int r, int MOD)
{
    vector<long long> f(n + 1,1);
    for (int i=2; i<=n;i++)
        f[i]= (f[i-1]*i) % MOD;
    return (f[n]*((InverseEuler(f[r], MOD) * InverseEuler(f[n-r], MOD)
}

int main()
{
    int n,r,p;
    while (~scanf("%d%d%d",&n,&r,&p))
    {
        printf("%lld\n",C(n,r,p));
    }
}
```

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).

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

/*
    C(n,r) mod m
    Using recurrence:
    C(i,k) = C(i-1,k-1) + C(i-1,k)
    Time Complexity: O(n*r)
    Space Complexity: O(n*r)
*/

long long C(int n, int r, int MOD)
{
```

```
    vector< vector<long long> > C(n+1,vector<long long> (r+1,0));

    for (int i=0; i<=n; i++)
    {
        for (int k=0; k<=r && k<=i; k++)
            if (k==0 || k==i)
                C[i][k] = 1;
            else
                C[i][k] = (C[i-1][k-1] + C[i-1][k])%MOD;
    }
    return C[n][r];
}
int main()
{
    int n,r,m;
    while (~scanf("%d%d%d",&n,&r,&m))
    {
        printf("%lld\n",C(n, r, m));
    }
}
```

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.

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

/*
    Time Complexity: O(n*r)
    Space Complexity: O(r)
*/
long long C(int n, int r, int MOD)
{
    vector< vector<long long> > C(2,vector<long long> (r+1,0));

    for (int i=0; i<=n; i++)
    {
        for (int k=0; k<=r && k<=i; k++)
            if (k==0 || k==i)
                C[i&1][k] = 1;
            else
                C[i&1][k] = (C[(i-1)&1][k-1] + C[(i-1)&1][k])%MOD;
    }
    return C[n&1][r];
}

int main()
{
    int n,r,m,i,k;
    while (~scanf("%d%d%d",&n,&r,&m))
    {
        printf("%lld\n",C(n, r, m));
    }
}
```

3. Using expansion of nCr
Since

```
C(n,k) = n!/((n-k)!k!)
         [n(n-1)...(n-k+1)][(n-k)...(1)]
       = ------------------------------
           [(n-k)...(1)][k(k-1)...(1)]
```

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

```
    n (n-1)     (n-k+1)
    - ----- ... -------
    k (k-1)       (1)
```

which we might write as:

```
C(n,k) = 1,                  if k = 0
       = (n/k)*C(n-1, k-1), otherwise
```

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

long long C(int n, int r)
{
    if (r==0) return 1;
    else return C(n-1,r-1) * n / r;
}

int main()
{
    int n,r,m;
    while (~scanf("%d%d",&n,&r))
    {
        printf("%lld\n",C(n, min(r,n-r)));
    }
}
```

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.

```
(.   .   .   .)  ( C(i-1,0) )   ( C(i,0) )
(.   .   .   .)  ( C(i-1,1) ) = ( C(i,1) )
(.   .   .   .)  ( C(i-1,2) )   ( C(i,2) )
(.   .   .   .)  ( C(i-1,3) )   ( C(i,3) )
```

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

```
(.   .   .   .)  ( C(i-1,0) )   ( C(i,0) )
(1   1   0   0)  ( C(i-1,1) ) = ( C(i,1) )
(0   1   1   0)  ( C(i-1,2) )   ( C(i,2) )
(0   0   1   1)  ( C(i-1,3) )   ( C(i,3) )
```

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

```
(.   .   .   .)  ( C(i-1,0) )   ( C(i,0) )
(1   1   0   0)  ( C(i-1,1) ) = ( C(i,1) )
(0   1   1   0)  ( C(i-1,2) )   ( C(i,2) )
(0   0   1   1)  ( C(i-1,3) )   ( C(i,3) )
```

And this then leads to the general form:

```
                i
(.   .   .   .)  ( C(0,0) )   ( C(i,0) )
(1   1   0   0)  ( C(0,1) ) = ( C(i,1) )
(0   1   1   0)  ( C(0,2) )   ( C(i,2) )
(0   0   1   1)  ( C(0,3) )   ( C(i,3) )
```

For example if we wan't C(4,3) we just raise the above matrix to the 4th power.

```
               4
(1   0   0   0)  ( 1 )   ( 1 )
(1   1   0   0)  ( 0 ) = ( 4 )
(0   1   1   0)  ( 0 )   ( 6 )
(0   0   1   1)  ( 0 )   ( 4 )
```

Here's a C++ code.

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

/*
    C(n,r) mod m
    Using Matrix Exponentiation
    Time Complexity: O((r^3)*log(n))
    Space Complexity: O(r*r)
*/

long long MOD;

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 );
```

```
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]*A.d
79          }
80      return C;
81  }
82
```

```
template< class T >
const Matrix< T > Matrix< T >::operator^( int P )
{
    if( P == 1 ) return (*this);
    if( P & 1 ) return (*this) * ((*this) ^ (P-1));
    Matrix B = (*this) ^ (P/2);
    return B*B;
}

long long C(int n, int r)
{
    Matrix<long long> M(r+1,r+1);
    for (int i=0;i<(r+1)*(r+1);i++)
        M.data[i]=0;
    M.data[0]=1;
    for (int i=1;i<r+1;i++)
    {
        M.data[i*(r+1)+i-1]=1;
        M.data[i*(r+1)+i]=1;
    }
    return (M^n).data[r*(r+1)];
}

int main()
{
    int n,r;
    while (~scanf("%d%d%lld",&n,&r,&MOD))
    {
        printf("%lld\n",C(n, r));
    }
}
```

5. Using the power of prime p in n factorial
The power of prime p in n factorial is given by

$$\varepsilon_p = \lfloor n/p \rfloor + \lfloor n/p^2 \rfloor + \lfloor n/p^3 \rfloor \ldots$$

If we call the power of p in n factorial, the power of p in nCr is given by
e = countFact(n,i) – countFact(r,i) – countFact(n-r,i)
To get the result we multiply p^e for all p less than n.

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

/* This function calculates power of p in n! */
int countFact(int n, int p)
{
    int k=0;
    while (n>0)
    {
        k+=n/p;
        n/=p;
    }
    return k;
}

```

```
/* This function calculates (a^b)%MOD */
long long pow(int a, int b, int MOD)
{
    long long x=1,y=a;
    while(b > 0)
    {
        if(b%2 == 1)
        {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

long long C(int n, int r, int MOD)
{
    long long res = 1;
    vector<bool> isPrime(n+1,1);
    for (int i=2; i<=n; i++)
        if (isPrime[i])
        {
            for (int j=2*i; j<=n; j+=i)
                isPrime[j]=0;
            int k = countFact(n,i) - countFact(r,i) - countFact(n-r,i)
            res = (res * pow(i, k, MOD)) % MOD;
        }
    return res;
}

int main()
{
    int n,r,m;
    while (scanf("%d%d%d",&n,&r,&m))
    {
        printf("%lld\n",C(n,r,m));
    }
}
```

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

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

and

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

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.

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

```
#include<vector>

long long SmallC(int n, int r, int MOD)
{
    vector< vector<long long> > C(2,vector<long long> (r+1,0));

    for (int i=0; i<=n; i++)
    {
        for (int k=0; k<=r && k<=i; k++)
            if (k==0 || k==i)
                C[i&1][k] = 1;
            else
                C[i&1][k] = (C[(i-1)&1][k-1] + C[(i-1)&1][k])%MOD;
    }
    return C[n&1][r];
}

long long Lucas(int n, int m, int p)
{
    if (n==0 && m==0) return 1;
    int ni = n % p;
    int mi = m % p;
    if (mi>ni) return 0;
    return Lucas(n/p, m/p, p) * SmallC(ni, mi, p);
}

long long C(int n, int r, int MOD)
{
    return Lucas(n, r, MOD);
}

int main()
{

    int n,r,p;
    while (~scanf("%d%d%d",&n,&r,&p))
    {
          printf("%lld\n",C(n,r,p));
    }
}
```

7. Using special n! mod p

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

n!* mod p = 1 * 2 * … * (p-1) * 1 * 2 * … * (p-1) * 2 * 1 * 2 * … * 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 * … * (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 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 * … * (p-1), which is common to all blocks, and multiplication of all *strange indices* i from 1 to 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*.

```
int factMOD(int n, int MOD)
{
    long long res = 1;
    while (n > 1)
    {
        long long cur = 1;
        for (int i=2; i<MOD; ++i)
            cur = (cur * i) % MOD;
        res = (res * powmod (cur, n/MOD, MOD)) % MOD;
        for (int i=2; i<=n%MOD; ++i)
            res = (res * i) % MOD;
        n /= MOD;
    }
    return int (res % MOD);
}
```

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:

```
long long factMOD(int n, int MOD)
{
    long long res = 1;
    while (n > 1)
    {
        res = (res * pow(MOD - 1, n/MOD, MOD)) % MOD;
        for (int i=2, j=n%MOD; i<=j; i++)
            res = (res * i) % MOD;
        n/=MOD;
    }
    return res;
}
```

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:

```
long long factMOD(int n, int MOD)
{
    long long res = 1;
    while (n > 0)
    {
        for (int i=2, m=n%MOD; i<=m; i++)
            res = (res * i) % MOD;
        if ((n/=MOD)%2 > 0)
            res = MOD - res;
    }
    return res;
}
```

Finally the complete code here:

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

/* This function calculates power of p in n! */
int countFact(int n, int p)
{
    int k=0;
    while (n>=p)
    {
        k+=n/p;
        n/=p;
    }
    return k;
}

/* This function calculates (a^b)%MOD */
long long pow(int a, int b, int MOD)
{
    long long x=1,y=a;
    while(b > 0)
    {
        if(b%2 == 1)
        {
            x=(x*y);
            if(x>MOD) x%=MOD;
        }
        y = (y*y);
        if(y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

/*  Modular Multiplicative Inverse
    Using Euler's Theorem
    a^(phi(m)) = 1 (mod m)
    a^(-1) = a^(m-2) (mod m) */
long long InverseEuler(int n, int MOD)
{
    return pow(n,MOD-2,MOD);
}

long long factMOD(int n, int MOD)
{
    long long res = 1;
    while (n > 0)
    {
        for (int i=2, m=n%MOD; i<=m; i++)
            res = (res * i) % MOD;
        if ((n/=MOD)%2 > 0)
            res = MOD - res;
    }
    return res;
}

long long C(int n, int r, int MOD)
```

```
{
    if (countFact(n, MOD) > countFact(r, MOD) + countFact(n-r, MOD))
        return 0;

    return (factMOD(n, MOD) *
            ((InverseEuler(factMOD(r, MOD), MOD) *
            InverseEuler(factMOD(n-r, MOD), MOD)) % MOD)) % MOD;
}

int main()
{
    int n,r,p;
    while (~scanf("%d%d%d",&n,&r,&p))
    {
        printf("%lld\n",C(n,r,p));
    }
}
```

-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

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();
};

template< class T >
Matrix< T >::Matrix( int m, int n )
{
    this->m = m;
    this->n = n;
    data = new T[m*n];
}

template< class T >
Matrix< T >::Matrix( const Matrix< T > &A )
{
    this->m = A.m;
    this->n = A.n;
    data = new T[m*n];
    for( int i = 0; i < m * n; i++ )
        data[i] = A.data[i];
}
```

```
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.data
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 :

```
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
```

```
                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),random.r
        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 $1 \leq a < p$ 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 $2^s.d$. 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 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 :

```python
import random
def modulo(a,b,c):
        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.

```
void preKMP(string pattern, int f[])
{
        int m = pattern.length(),k;
        f[0] = -1;
        for (int i = 1; i<m; i++)
        {
                k = f[i-1];
                while (k>=0)
```

```
                {
                        if (pattern[k]==pattern[i-1])
                                break;
                        else
                                k = f[k];
                }
                f[i] = k + 1;
        }
}

bool KMP(string pattern, string target)
{
        int m = pattern.length();
        int n = target.length();
        int f[m];

        preKMP(pattern, f);

        int i = 0;
        int k = 0;

        while (i<n)
        {
                if (k==-1)
                {
                        i++;
                        k = 0;
                }
                else if (target[i]==pattern[k])
                {
                        i++;
                        k++;
                        if (k==m)
                                return 1;
                }
                else
                        k=f[k];
        }
        return 0;
}
```

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;
    }
```

```
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.

```
#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]+dis[k][j]);
}

int main()
{
        //input size N
        init(N);
        //set values for dis[i][j]
        floyd_warshall(N);
}
```

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 3 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, $1 \leq m \leq n$. 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

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

or

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

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

Multiply two Numbers Without Using * Operator

with 7 comments

Lets have a fun question. **Multiply two numbers without using the * operator.**
Here's the code in C/C++:

```
main(a,b,m)
{
    while (~scanf("%d%d",&a,&b))
    {
          m=0;
          while (a)
          {
                if (a&1)
                   m+=b;
                a>>=1;
                b<<=1;
          }
          printf("%d\n",m);
    }
}
```

The above code is a implementation of an algorithm better known as the Ethiopian Multiplication or the Russian Peasant Multiplication.
Here's the algorithm :

```
1. Let the two numbers to be multiplied be a and b.
2. If a is zero then break and print the result.
3. If a is odd then add b to the result.
4. Half a, Double b. Goto Step 2.
```

NJOY!
-fR0DDY

Written by fR0DDY

April 8, 2010 at 10:05 AM

Posted in Beautiful Codes

Tagged with ethiopian, multiply, number

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 "2070331676202550000000" 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

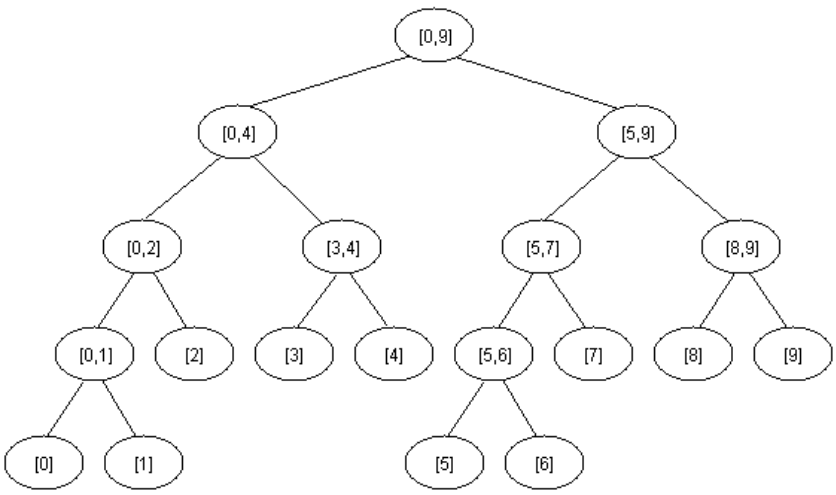
Segment Trees

with 19 comments

A segment tree is a heap-like data structure that can be used for making update/query operations upon array intervals in logarithmical time. We define the segment tree for the interval [i, j] in the following recursive manner:

- the first node will hold the information for the interval [i, j]
- if i<j the left and right son will hold the information for the intervals **[i, (i+j)/2]** and **[(i+j)/2+1, j]**

See the picture below to understand more :



Segment Tree

We can use segment trees to solve Range Minimum/Maximum Query Problems (RMQ). The time complexity is **T(N, log N)** where O(N) is the time required to build the tree and each query takes O(log N) time. Here's a C++ template implementation :

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

template<class T>
class SegmentTree
{
     int *A,size;
     public:
     SegmentTree(int N)
     {
          int x = (int)(ceil(log2(N)))+1;
          size = 2*(int)pow(2,x);
          A = new int[size];
          memset(A,-1,sizeof(A));
     }
     void initialize(int node, int start,
                         int end, T *array)
     {

          if (start==end)
             A[node] = start;
          else
          {
              int mid = (start+end)/2;
              initialize(2*node,start,mid,array);
              initialize(2*node+1,mid+1,end,array);
              if (array[A[2*node]]<=
                     array[A[2*node+1]])
                 A[node] = A[2 * node];
              else
                  A[node] = A[2 * node + 1];
          }
     }
     int query(int node, int start,
                   int end, int i, int j, T *array)
     {
         int id1,id2;
         if (i>end || j<start)
            return -1;

         if (start>=i && end<=j)
            return A[node];

         int mid = (start+end)/2;
```

```
        id1 = query(2*node,start,mid,i,j,array);
        id2 = query(2*node+1,mid+1,end,i,j,array);

        if (id1==-1)
           return id2;
        if (id2==-1)
           return id1;

        if (array[id1]<=array[id2])
           return id1;
        else
            return id2;
    }
};

int main()
{
    int i,j,N;
    int A[1000];
    scanf("%d",&N);
    for (i=0;i<N;i++)
        scanf("%d",&A[i]);

    SegmentTree<int> s(N);
    s.initialize(1,0,N-1,A);
    while (scanf("%d%d",&i,&j)!=EOF)
      printf("%d\n",A[s.query(1,0,N-1,i-1,j-1,A)]);
}
```

Resources:

1. Topcoder Tutorial

NJOY!
-fR0D

Written by fR0DDY

September 15, 2009 at 8:21 PM

Posted in Programming

Tagged with C, code, complexity, data, RMQ, Segment, structure, time, Tree

Subsequence Frequency

with 6 comments

The question is to find the number of times a string appears as a subsequence in an another string. A similar question was asked in Google Code Jam 09. Here is the link.

So how do we do it. If you think the general way the first brute force method which comes to mind is to get a matrix for the general subsequence problem and then backtrack all paths that lead to the subsequence. But the backtracking will be too time consuming. Now the second thought which comes to mind is can't we have a similar dynamic programming approach to find the frequency rather than length. Yes, thankfully we can do that. How?

First lets start with a string of two letters. Suppose we need to find the occurence of ab in abbacb. We can calculate the answer to be four. How did we do it, there are three b's after first 'a' and only one after the second 'a', all in all four. So if somehow we manipulate the table to deal with the frequency of characters we can take out the total numbers of times the string appears as subsequence. If you wan't to try out the algorithm for yourself, now is the right time. What follows next is the algorithm.
We have the table for the above example as :

		a	b	b	a	c	b
	0	1	1	1	1	1	1
a	0	1	1	1	2	2	2
b	0	0	1	2	2	2	4

As seen above we set the row[0] values to 1 and column[0] values to 0. Here's the algorithm.
Let's call the string whose frequency is to be found as str1 and given string str2.

```
for i from 1 to length(str1)
{
    for j from 1 to length(str2)
    {
        if (str1[i-1]==str2[j-1])
           table[i][j]=table[i-1][j]+table[i][j-1];
        else
            table[i][j]=table[i][j-1];
    }
}
```

Here's a C++ code for the Code Jam Question :

```
#include
using namespace std;

char str[1000],cj[]="welcome to code jam";
int M[30][1000]={0};

int main()
{
    int t,i,j,k,l,lcj=strlen(cj);;
    scanf("%d\n",&t);
    for (j=1;j<=1000;j++)
        M[0][j]=1;
    for (j=1;j<=lcj;j++)
        M[j][0]=0;
    for (k=1;k<=t;k++)
    {
        gets(str);
        l=strlen(str);
        for (i=1;i<=lcj;i++)
        {
            M[i][0]=0;
            for (j=1;j<=l;j++)
                if (cj[i-1]==str[j-1])
                M[i][j]=(M[i-1][j]+M[i][j-1])%10000;
                else M[i][j]= M[i][j-1];
        }
        printf("Case #%d: %04d\n",k,M[lcj][l]);
    }
}
```

Happy Coding!
-fR0D

Written by fR0DDY

September 6, 2009 at 3:12 PM

Posted in Algorithm, Programming

Tagged with dynamic, frequency, gcj, number, Programming, subsequence, times

Longest Increasing Subsequence (LIS)

with 32 comments

The longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous.
For example for the sequence -7 10 9 2 3 8 8 1 the longest (strictly) increasing subsequence is -7 2 3 8 of length 4.

There are few methods to solve this question. The very first method is to sort the given sequence and save it into another array and then take out the longest common subsequence of the two arrays. This method has a complexity of $O(n^2)$ which should be good for most question but not all.

We will see here an algorithm which take O(n log k) time where k is the size of the LIS. For explanation on this algorithm see Faster Algorithm on this page. Here's a small code to calculate only the length of the LIS.

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

int LIS(vector<int> A)
{
    int N = A.size(),i;
    set<int> s;
    set<int>::iterator k;
    for (i=0;i<N;i++)
    {
        if (s.insert(A[i]).second)
        {
           k = s.find(A[i]);
           k++;
           if (k!=s.end())
              s.erase(k);
        }
    }
    return s.size();
}
```

To also get the LIS we need to maintain a previous array which stores the index of the previous element in the LIS. Here's a C++ implementation. It returns the LIS as an array.

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

typedef pair < int , int > PI;

vector<int> LIS(vector<int> A)
{
    int N = A.size(),i,j=-1,t;
    vector<int> pre(N,-1),res;
    map<int,int> m;
    map<int,int>::iterator k,l;
    for (i=0;i<N;i++)
    {
        if (m.insert(PI(A[i],i)).second)
        {
           k = m.find(A[i]);
           l = k;
           k++;
           if (l==m.begin())
                  pre[i]=-1;
           else
           {
               l--;
               pre[i]=l->second;
           }
           if (k!=m.end())
              m.erase(k);
        }
    }
    k=m.end();
    k--;
    j = k->second;
    while (j!=-1)
    {
          res.push_back(A[j]);
          j = pre[j];
    }
    reverse (res.begin(),res.end());
    return res;
}
```

Note that if there are more than one LIS the above code prints the last one which occured in the input array. Also to get the LDS we just need to change the lines :

```
set<int> s;
to
set<int,greater<int> > s;
and
map<int,int> m;
to
map<int,int,greater<int> > m;
```
Also little changes need to be made to the above codes if you dont want the LIS to be strictly increasing and rather be Longest Non-Decreasing Subsequence or Longest Non-Increasing Subsequence.

Happy Coding!
-fR0D

Written by fR0DDY

August 12, 2009 at 2:34 PM

Posted in Programming

Tagged with C, code, complexity, decreasing, increasing, lds, lis, longest, subsequence, time

Longest Common Subsequence (LCS)

with 20 comments

The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences (often just two).

First we look into only finding the length of LCS. We can easily construct an exponential time recursive algorithm to compute the length of the LCS. But using Dynamic Programming (DP) to compute the solution bottom up the same job can be done in O(mn) time where m and n are the lengths of the subsequences. Here is a C++ code to do so. Note that the code also uses very less space.

```
int LCS(string X,string Y)
{
     if (Y.length() > X.length())
        swap(X,Y);
     int m = X.length(),n=Y.length();
     vector< vector<int> > c(2, vector<int>(n+1,0));
     int i,j;
     for (i=1;i<=m;i++)
     {
         for (j=1;j<=n;j++)
         {
             if (X[i-1]==Y[j-1])
                c[1][j]=c[0][j-1]+1;
             else
                 c[1][j]=max(c[1][j-1],c[0][j]);
         }
         for (j=1;j<=n;j++)
                 c[0][j]=c[1][j];
     }
     return (c[1][n]);
}
```

If we also wish to print the subsequence we use a space of size m x n to get the LCS. Here's the code

```
string X,Y;
vector< vector<int> > c(101, vector<int>(101,0));
int m,n,ctr;

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

     int i,j;
     for (i=0;i<=m;i++)
         for (j=0;j<=n;j++)
             c[i][j]=0;

     for (i=1;i<=m;i++)
         for (j=1;j<=n;j++)
         {
             if (X[i-1]==Y[j-1])
                c[i][j]=c[i-1][j-1]+1;
             else
                 c[i][j]=max(c[i][j-1],c[i-1][j]);
         }
}

void printLCS(int i,int j)
{
    if (i==0 || j==0)
       return;
    if (X[i-1]==Y[j-1])
    {
       printLCS(i-1,j-1);
       cout<<X[i-1];
    }
    else if (c[i][j]==c[i-1][j])
         printLCS(i-1,j);
    else
        printLCS(i,j-1);
}

int main()
{
    while(cin>>X>>Y)
    {
        LCS();
        printLCS(m,n);
        cout<<endl ;
    }
}
```

This post is a prequel to posts on similar topics that i wish to write.

NJOY
-fR0D

Written by fR0DDY

August 7, 2009 at 5:29 PM

Posted in Programming

Tagged with C, code, common, DP, LCS, length, longest, subsequence

Program without header file?

with one comment

So continuing in the series of program without, here are two more programs. The first one is just a valid C program without any header file.

```
int main()
{
    return 0;
}
```

The above program will actually compile and run. But you will say it did nothing. So lets have a program without header file which prints something.

```
extern "C"
{
       int printf(const char *format,...);
}

int main()
{
    printf("Hello World");
    return 0;
}
```

The only unexplained feature is extern “C”. For more about it read this.

NJOY!
fR0D

Written by fR0DDY

August 7, 2009 at 2:35 PM

Posted in Beautiful Codes

Tagged with C, code, file, header, print, tricky, without

Programs without semicolons?

with 9 comments

Let’s have a little fun. First of all try to write a C code to print Hello World! without using semicolons. Here’s the answer in C

```
#include <stdio.h>
int main()
{
    if (printf("Hello World!\n")) {}
}
```

or if you are a hardcore C++ fan then

```
#include <iostream>
int main()
{
    if (std::cout << "Hello world!" << std::endl)
    {}
}
```

But the real gem is the following program. A program to find the factorial of a number.

```
#include<iostream>
int main(int n)
{
    if (std::cin>>n)
    {
        if (int f=1 )
        {
            while (n != 1)
            {
                if ( f = n * f )
                {
                   if (--n){}
                }
            }
            if(std::cout<<"fac = "<<f<<std::endl)
            {}
        }
        else
        {}
    }
}
```

Infact any C program can be converted a non-semicolon form.

NJOY!
-fR0D

Written by fR0DDY

July 21, 2009 at 9:35 PM

Posted in Beautiful Codes

Tagged with C, code, fun, semicolon

Counting Problems

with 17 comments

Problem 1: Given a set of n numbers a_i sum up to M, and any K ≤ M, whether there is a subset of the numbers such that they sum up to K?

Also called the **Subset Sum Problem**, in this problem, the number of times we can use a particular number is limited i.e. we can use a number only once. This problem can be done in two ways
1> Binary Representation of Numbers and
2> DP

1> Using Binary Representation of Numbers
First lets discuss the common approach for iterating through all subsets of a set. Mathematics tells us that the total number of subsets of a set is 2^n. THe easier way to do this is in any programmin language is to represent each element of a set by a single bit of the number. So for example, if we have 3 elements in a set then we have 8 subsets. or we can iterate from 0 to 7 to get all subsets as

Number	Binary Representation
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

So if we let 1 in the binary represent that the number is selected and 0 that the number is not selected, we actually have all the subsets of the set. We can use this approach to solve the above problem by iterating through all subsets and finding whether the sum is equal to the required sum. The point to be noted is that this approach is quite slow since for a worst case scenario we iterate through all the 2^n cases. Also

checking each bit takes time. Here's a simple C++ implementation of the above logic. Below is a program which takes as input N numbers and M the required subset sum and outputs Yes if the subset sum M is possible and No otherwise.

```
int a[21];
int main()
{
    int t,N,M,i,j,k,sum;
    bool F;
    scanf("%d",&t);
    while(t--)
    {
        scanf("%d%d",&N,&M);
        for (i=0;i<N;i++)
            scanf("%d",&a[i]);

        F=0;
        j=1<<N;
        for (i=1;i<j && F==0;i++)
        {
            sum=0;
            for (k=0;k<N;k++)
            {
                if ((i & (1<<k)) == (1<<k))
                   sum += a[k];
            }
            if (sum == M)
            {
                F=1;
                printf("Yes\n");
            }

        }
        if (F==0)
           printf("No\n");

    }
}
```

The time complexity of the above code is O(2^n). We also have a O($2^{n/2}$) algorithm, where we divide the input numbers into two equal subsets. Then we calculate the sum of subsets of each of the two subset. We then iterate through the sums of one subset and check whether M-sum is present in the sums of the other subset. If yes, we get our desired result. If there are no such values we print no.

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

int a[21];
int main()
```

```
{
    int t,N,M,i,j,k,sum,x;
    set<int> s1,s2;
    set<int>::iterator l;
    scanf("%d",&t);

    while(t--)
    {
        scanf("%d%d",&N,&M);
        s1.clear();
        s2.clear();
        for (i=0;i<N;i++)
            scanf("%d",&a[i]);


        j=1<<(N/2);
        for (i=0;i<j;i++)
        {
            sum=0;
            for (k=0;k<N/2;k++)
            {
                if ((i & (1<<k)) == (1<<k))
                   sum += a[k];
            }
            s1.insert(sum);
        }

        x=N-N/2;
        j=1<<x;
        for (i=0;i<j;i++)
        {
            sum=0;
            for (k=0;k<x;k++)
            {
                if ((i & (1<<k)) == (1<<k))
                   sum += a[k+N/2];
            }
            s2.insert(sum);
        }

        for (l=s1.begin();l!=s1.end();l++)
        {
            if (binary_search(s2.begin(),s2.end(),M-*l))
            {
               printf("Yes\n");
               break;
            }
        }
        if (l==s1.end())
```

```
            printf("No\n");
    }
}
```

2> DP Approach

This is the other approach to solving this problem. This method takes into account that the possible sum's after including a number, are all previous possible sum's plus the new number. So we only check whether x-a_i is a possible sum, if yes x is also a possible sum. We iterate from M (maximum possible sum) to a_i because we cannot include a number twice. Here's the C++ implementation of the above program using DP approach.

```
int m[30000],a[21];
int main()
{
    int t,N,M,i,j;
    scanf("%d",&t);
    while(t--)
    {
        scanf("%d%d",&N,&M);
        for (i=0;i<N;i++)
            scanf("%d",&a[i]);
        for (i=0;i<=M;i++)
            m[i]=0;
        m[0]=1;
        for(i=0; i<N; i++)
                for(j=M; j>=a[i]; j--)
                        m[j] |= m[j-a[i]];
                if (m[M])
                   printf("Yes\n");
           else
               printf("No\n");
    }
}
```

Problem 2: Given that the a_i's are unlimited i.e you can use the numbers in the subset as many times as you want find whether a particular sum M is possible and if possible how many ways are there to do it?

Also called the **Counting Change Problem**, this problem is a variation of the above problem and we just need to change the direction of the second loop in the above problem. Here's a program to find the number of ways to change a amount of Rs. 100 using the coins and notes of RS. 1,2,5,10,20,50,100.

```
int main()
{
    int coin[7]={1,2,5,10,20,50,100},i,j,x,c;
    x=100;
    vector<long long> ways(x+1,0);
    ways[0]=1;
    for (i=0;i<7;i++)
    {
      c=coin[i];
      for (j=c;j<=x;j++)
          ways[j]+=ways[j-c];
    }
    cout<<ways[100]<<endl;
}
```

More variations of the above problem can be minimizing the number of elements required to get a particular sum.

Problem 3: Find the number of ways of writing a number as the sum of positive integers?

Also called the **Partition Function** P of n, it is denoted by *P(n)*. For example,

5	5
	4+1
	3+2
	3+1+1
	2+2+1
	2+1+1+1
	1+1+1+1+1

Euler invented a generating function which gives rise to a recurrence equation in *P(n)*,

$$P(n) = \sum_{k=1}^{n} (-1)^{k+1} \left[P\left(n - \tfrac{1}{2} k(3k-1)\right) + P\left(n - \tfrac{1}{2} k(3k+1)\right) \right]$$

Below is a program to calculate *P(n)*. It can easily modified to include large numbers.

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

map <int , long long> P;

long long Calculate_Pn(long long n)
{
  if (n < 0)
     return 0;

  long long Pn = P[n];

  if (Pn == 0)
  {
     long long k;
     for (k = 1; k <= n; k++)
     {
         // A little bit of recursion
         long long n1 = n - k * (3 * k - 1) / 2;
         long long n2 = n - k * (3 * k + 1) / 2;

         long long Pn1 = Calculate_Pn(n1);
         long long Pn2 = Calculate_Pn(n2);

         // elements are alternately added
         // and subtracted
         if (k % 2 == 1)
            Pn = Pn + Pn1 + Pn2;
         else
              Pn = Pn - Pn1 - Pn2;
    }
       P[n]=Pn;
  }
  return Pn;
}

int main()
{
    long long i;
    P[0]=1;
    while (scanf("%lld",&i)!=EOF)
          printf("%lld\n",Calculate_Pn(i));
}
```

Happy Coding!!!
-fR0D

Written by fR0DDY

July 21, 2009 at 10:05 AM

Posted in Programming

Tagged with C, change, code, coin, counting, Euler, function, partition, subset, sum

Last Non-zero Digit of Factorial

with 26 comments

If you are not familiar with factorial read this.

Question: Find the last non-zero digit of n!

Approach 1
If you think you can calculate the factorial first and then divide out all the zeroes, you can but only to a certain extent, i mean to the point you can calculate n factorial. What if you had to find the last non-zero digit of a very large number whose factorial you cannot calculate.

Approach 2
This is the most popular technique and quite handy too. We know that n! = 1 x 2 x 3 …x (n-1) x n
or n! = (2^a2)(3^a3)(5^a5)(7^a7)…
or n! = (2^a2)(5^a5)(3^a3)(7^a7)…

Now we know that the number of zeroes in n1 is equal to a5. So n! divided by 10^a5 will be equal to factorial without trailing digits. Lets call this n! without trailing zeroes as (n!)’ .So,
(n!)’ = (2^(a2-a5))(3^a3)(7^a7)…

Let L(k) denote the least significant non-zero decimal digit of the integer k. So from above we can infer that
L(n!) = L((n!)’) = L[(2^(a2-a5))(3^a3)(7^a7)…]
= L[(3^a3)(7^a7)…]*L[(2^(a2-a5))]

This logic is implemented in the C/C++ function below.

```
int last_digit_factorial(int N)
{
    int i,j,ans=1,a2=0,a5=0,a;

    for (i = 1; i <= N; i++)
    {
        j = i;
        //divide i by 2 and 5
        while (j%2==0)
        {
              j /= 2;
              a2++;
        }
        while (j%5==0)
        {
              j/=5;
              a5++;
        }

        ans = (ans*(j%10))%10;
    }

    a=a2-a5;

    for (i = 1; i <= a; i++)
        ans=(ans * 2) %10;

    return ans;
}
```

Approach 3

Fact : The powers of 2 3 7 and 1 have cyclic last digit.
So we divide all the primes into one of these groups and then take out the power accordingly.

power1[4]={1, 1, 1, 1}
power2[4]={6, 2, 4, 8}
power3[4]={1, 3, 9, 7}
power7[4]={1, 7, 9, 3}
power9[4]={1, 9, 1, 9}

So for example, if we have to find last digit in 6!
6! = 1 * 2 * 3 * 4 * 5 * 6
6! = 2^3 * (3^2) * 10
6!' = 2^3 * (3^2)

We know the power of 3 has cyclic last digit, so 3^2 last digit is 9 or power3[2 mod 4]
L(6!) = L(6!') = power2[3 mod 4] * power3[2 mod 4] mod 10
= 32 mod 10 = 2

But why code this method when we have a simpler approach.

Approach 4

Theory : Since we know that only a factor 5 brings in zero and we always have more powers of 2 than of 5, so the value of L(n!) is easily computed recursively as L(L(n)L((n-1)!)) unless n is a multiple of 5, in which case we need more information. As a result, the values of L(n!) come in fixed strings of five, as shown below :

	1	5	10	15	20	25
L(n!)	1264	22428	88682	88682	44846	44846

Thus if we know any value of 5n we can get the value of 5n+j. For example if we know value of L(15!) we know the value of L(16!), L(17!) up-to L(19!). But how to get the value of 5n. If we map the starting values of the L(5n!) we get like 2 8 8 4 4 8 2 2 6. They come in another fixed strings of five. So we need to know L(25n!) to know L((25n+5j)!) for j = 0,1,2,3,4. Continuing in this way if we tabulate the values of L((5^t n)!) we get :

L(n!)	1264	22428	88682	88682	44846	44846
L(5n!)	2884	48226	24668	48226	48226	86442
L(25n!)	4244	82622	82622	28488	46866	64244
L(125n!)	8824	68824	26648	68824	42286	26648
L(625n!)	1264	22428	88682	88682	44846	44846

So we see that the pattern for L(625n!) is same as L(n!). Hence we can conclude that the pattern for L((5^j n)!) is the same as for L((5^(j+4) n)!). So we can use a modulo 4 function to get the next results. Also we can see below that each of the four row have a starting digit as 0 , 2 , 4, 6 and 8.

Result :

k mod 4					
0	06264	22428	44846	66264	88682
1	02884	24668	48226	62884	86442
2	04244	28488	46866	64244	82622
3	08824	26648	42286	68824	84462

This is all we need to get the last digit of n! or L(n!). First we convert the given number n into base 5. So we have
n = d_0 + d_1*5 + d_2*5^2 + … + d_h*5^h
where d_0 is the least significant digit.

Now we enter the above table at the row h (mod 4) in the block whose first digit is 0 (because the coefficient of 5^(h+1) is zero), and determine the digit in the (d_h)th position of this block. Let this digit be denoted by s_h. Then we enter the table at row h-1 (mod 4) in the block that begins with s_h, and determine the digit in the (d_(h-1))th position of this block. Let this digit be denote by s_(h-1). We continue in this way down to s_0, which is the least significant non-zero digit of n!.

Example :
To illustrate, consider the case of the decimal number n=1592. In the base 5 this is n=22332. Now we enter the above table at row k=4=0 (mod 4) in the block beginning with 0, which is 06264. The leading

digit of n (in the base 5) is 2, so we check the digit in position 2 of this block to give L((2*5^4)!) = 2. Then we enter the table at row k=3 (mod 4) in the block beginning with 2, which is 26648, to find L((2*5^4 + 2*5^3)!) = 6.

Then in the row k=2 (mod 4), the block beginning with 6 is 64244, and we find L((2*5^4 + 2*5^3 + 3*5^2)!) = 4. From this we know we're in the block 48226 in row k=1 (mod 4), so we have L((2*5^4 + 2*5^3 + 3*5^2 + 3*5)!) = 2. Finally, we enter the row k=0 (mod 4) in block 22428 to find the result

L(1592!) = L((2*5^4 + 2*5^3 + 3*5^2 + 3*5 + 2)!) = 4

Thus if there are k digits in the base 5 representation of n then we only need k lookups in the table to get the last non-zero digit. Finally a C++ program for the algorithm :

```
#include
using namespace std;

int A[4][5][5] = {
{0,6,2,6,4,2,2,4,2,8,4,4,8,4,6,6,6,2,6,4,8,8,6,8,2},
{0,2,8,8,4,2,4,6,6,8,4,8,2,2,6,6,2,8,8,4,8,6,4,4,2},
{0,4,2,4,4,2,8,4,8,8,4,6,8,6,6,6,4,2,4,4,8,2,6,2,2},
{0,8,8,2,4,2,6,6,4,8,4,2,2,8,6,6,8,8,2,4,8,4,4,6,2}
};

char num[200];
char* dto5(int n)
{
     int b=5;
     int j,l;
     register int i=0;
     do
     {
           j=n%b;
           num[i++]=(j<10) ? (j+'0'): ('A'+j-10);
     }while((n/=b)!=0);

     num[i]='';
     l=strlen(num);
     reverse(num,num+l);
     return num;
}

int last_digit_factorial(int N)
{
    char *num=dto5(N);
    int l = strlen(num),s=0,i;
    for (i=0;i<l;i++)
    {
        s=A[(l-1-i)%4][s/2][num[i]-'0'];
    }
    return s;
}

int main()
{
  int N;

  while (scanf("%d", &N) == 1)
  {
        printf("%d\n", last_digit_factorial(N));
  }
  return 0;
}
```

Can we make it even simpler. Yes.

Let's write n as 5k + r

So,
(n)! = (5k + r)!
= 1·2·3…k.(k+1)…(5k-1)·(5k)·(5k+1)…(5k+r)
(Seperating out multiples of 5)
= {5·10·15…(5k)} · {1·2·3·4·6·7…(5k-4)·(5k-3)·(5k-2)·(5k-1)·(5k+1)…(5k+r)}
(Expanding multiples of 5)
= {5·(2·5)·(3·5)…(k·5)} · {1·2·3·4·6·7…(5k-4)·(5k-3)·(5k-2)·(5k-1)·(5k+1)…(5k+r)}
= 5^k {1·2·3…k} · {1·2·3·4·6·7…(5k-4)·(5k-3)·(5k-2)·(5k-1)·(5k+1)…(5k+r)}
= 5^k · k! · {1·2·3·4·6·7…(5k-4)·(5k-3)·(5k-2)·(5k-1)·(5k+1)…(5k+r)}
= 5^k · k! · product{(5·i+1)·(5·i+2)·(5·i+3)·(5·i+4), i = 0 to k – 1} · (5k+1)…(5k+r)
(Multiplying and dividing by 2^k)
= 5^k · 2^k · k! · product((5i+1)·(5i+2)·(5i+3)·(5i+4)/2, i = 0 to k – 1) · (5k+1)…(5k+r)
= 10^k · k! · product((5i+1)·(5i+2)·(5i+3)·(5i+4)/2, i = 0 to k – 1) · (5k+1)…(5k+r)

Now to find the last digit of n! or L(n), we remove the power of 10 first and move to next type.

So,
L(n) = L(5k+r)
L(n) = L(k! · product((5i+1)·(5i+2)·(5i+3)·(5i+4)/2, i = 0 to k – 1))
L(n) = [L(k) · {product((5i+1)·(5i+2)·(5i+3)·(5i+4)/2, i = 0 to k – 1) mod 10} · {(5k+1)…(5k+r) mod 10 }] mod 10

Now let us first find P = (5i+1)·(5i+2)·(5i+3)·(5i+4)/2 mod 10

To calculate this we will have to take (mod 2) value and (mod 5) value and then using chinese remainder take out (mod 10) value.

Clearly P mod 2 = 0.

Modular inverse of 2 (mod 5) is 3.

So,
P mod 5 = 1·2·3·4·3 mod 5
= 72 mod 5
= 2 mod 5

Using the chinese remainder theorem we get,

P mod 10 = 2 mod 10

Hence,

L(n) = [L(k) · {product((5i+1)·(5i+2)·(5i+3)·(5i+4)/2, i = 0 to k – 1) mod 10} · {(5k+1)…(5k+r) mod 10 }] mod 10
= [L(k) · {product(2, i = 0 to k – 1) mod 10} · L(r)] mod 10
= [L(k) · L(r) · {2^k mod 10}] mod 10
= 2^k · L(k) · L(r) mod 10

Also,
2^k mod 10 = 1 when k = 6 when k = 4i
= 2 when k = 4i+1
= 4 when k = 4i+2
= 8 when k = 4i+3

Hence,

L(0) = L(1) = 1
L(2) = 2
L(3) = 6
L(4) = 4

L(n) = L(5k+r) = 2^k·L(k)·L(r) (mod 10)

And here's a simpler code.

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

int P(int K)
{
        int A[]={6,2,4,8};
        if (K<1) return 1;
        return A[K%4];
}

int L(int N)
{
        int A[]={1,1,2,6,4};
        if (N<5) return A[N];
        return (P(N/5)*L(N/5)*L(N%5))%10;
}

int main()
{
        int N;
        while (scanf("%d", &N) == 1)
                printf("%d\n", L(N));
        return 0;
}
```

NJOY!!!
fR0D

Written by fR0DDY

June 20, 2009 at 6:12 PM

Posted in Programming

Tagged with C, code, digit, factorial, last, lookup, non-zero

Factorial

with 6 comments

In mathematics, the factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example, 4! = 1 x 2 x 3 x 4 = 24

In programming, factorial comes to use at many places. Sometimes you simply have to calculate factorials but in C/C++ you are limited by the range of integers. Even long long can store only up-to 20!
.

```
int main()
{
    long long i,j=1;
    for (i=1;i<21;i++)
    {
        j*=i;
        printf("%lld! = %lld\n",i,j);
    }
}
```

So you have to find factorials using strings and multiply the way you used to do in std I or II. If you haven't done so do now, you will learn a lot. Or if you know some language like Python or Haskell, which allow larger numbers you need not worry. For example, a code to accept t number of test cases and then print the factorial of next t numbers(less than 100) in python would be :

```
f=[1]*101
for i in range(2,101):
    f[i]=f[i-1]*i

n=input()
for i in range(0,n):
    x=input()
    print f[x]
```

Sometimes the question can be to find n! as a power of primes. To know the power of prime in a factorial you can use the formula:

$$\varepsilon_p = \lfloor n/p \rfloor + \lfloor n/p^2 \rfloor + \lfloor n/p^3 \rfloor \ldots$$

where ε_p is the power of prime p in factorial n.

Number of trailing zeroes in n Factorial

Finding number of trailing zeroes is very easy. Because you can get a zero only through a multiple of 5. So the number of trailing zeroes is equal to the multiples of five which is :

$$\varepsilon_5 = \lfloor n/5 \rfloor + \lfloor n/5^2 \rfloor + \lfloor n/5^3 \rfloor \ldots$$

A simple c-sharp program to get t test cases and then output the number of trailing zeroes in the factorial of next t numbers would be :

```
using System;


class Program
{
    public static void Main()
    {
        int T, N, sum, x;
        T = int.Parse(Console.ReadLine());
        while (T>0)
        {
            N = int.Parse(Console.ReadLine());
            x = 5;
            sum = 0;
            while (N / x > 0)
            {
                sum += N / x;
                x *= 5;
            }
            Console.WriteLine(sum);
            T--;
        }
    }
}
```

I will be writing soon on how to find the last non-zero digit and if possible also last k non zero digits.

ciao
fR0D

Written by fR0DDY

June 20, 2009 at 12:39 PM

Posted in Programming

Tagged with C, code, factorial, large, number, power, prime, Python, trailing, zero

Range Minimum Query

with 12 comments

RANGE QUERY

Given a (big) array r[0..n-1], and a lot of queries of certain type. We may want to pre-process the data so that each query can be performed fast. In this section, we use T (f, g) to denote the running time for an algorithm is O(f(n)) for pre-processing, and O(g(n)) for each query.

If the queries are of type getsum(a, b), which asks the sum of all the elements between a and b, inclusive, we have a T (n, 1) algorithm: Compute s[i] to be the sum from r[0] to r[i-1], inclusive, then getsum(a,b) simply returns s[b+1]-s[a].

RANGE MINIMUM QUERY

For the queries of the form getmin(a,b) asks the minimum elements between r[a] and r[b], inclusive, the task is little more hard. The idea is always to get the min in some big ranges, so in the queries we may try to use these big ranges to compute fast. One simple algorithm is T (n,sqrt(n)): Break the n numbers into sqrt(n) regions, each of size sqrt(n). Compute the champion for each region. For each query getmin(a,b), we go from a towards right to the nearest station (at most sqrt(n) steps), then go by at most sqrt(n) stations (big regions) to the nearest station before b, and from there go to b.

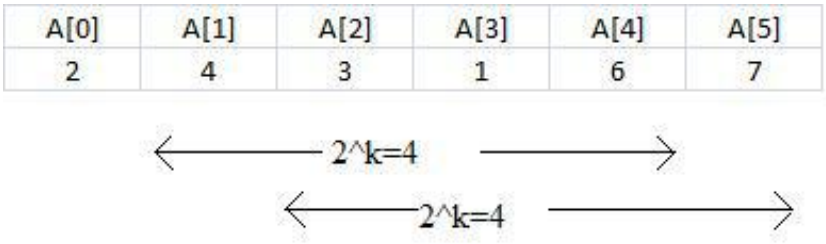
Sparse Table (ST) Algorithm

A better approach is to preprocess **RMQ** for sub arrays of length $\mathbf{2^k}$ using dynamic programming. We will keep an array **preProcess[0, N-1][0, logN]** where **preProcess[i][j]** is the index of the minimum value in the sub array starting at **i** having length $\mathbf{2^j}$. For example :

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
2	4	3	1	6	7

For the above array the preProcess[1][0] = 1, preProcess[1][1]=2,preProcess[1][2]=3 and so on. Specifically, we find the minimum in a block of size 2^j by comparing the two minima of its two constituent blocks of size 2^{j-1}. More formally, preProcess[i, j] = preProcess[i, j -1] if A[preProcess[i, j -1]] <= A[preProcess[i+2^{j-1}, j -1]] and preProcess[i, j] = preProcess[i+2^{j-1}, j -1] otherwise.

Once we have these values preprocessed, let's show how we can use them to calculate RMQ(i, j). The idea is to select two blocks that entirely cover the interval [i..j] and find the minimum between them. We select two overlapping blocks that entirely cover the subrange: let 2k be the size of the largest block that fits into the range from i to j, that is let k = log(j – i). Then rmq(i, j) can be computed by comparing the minima of the following two blocks: i to i + 2^k – 1 (preProcess(i, k)) and j – 2^k + 1 to j (preProcess(j –2^k +1, k)). These values have already been computed, so we can find the RMQ in constant time.



So as shown above if we need to calculate minimum between A[1] and A[5] we will take two sections of size 4 and compare their minimum values to get the answer. Below is a C++ template class implementation :

```
int preProcess[1000][10];
template
class RMQMin
{
  T *A;
  public:
    RMQMin(int N,T *array):A(array)
    {
        int i,j;
        for (i=0;i<N;i++)
            preProcess[i][0]=i;
        for (j=1; (1<<j)<=N; j++)
            for (i=0; i+(1<<j)-1<N; i++)
                preProcess[i][j]=
                A[preProcess[i][j-1]]<=
                A[preProcess[i+(1<<(j-1))][j-1]]?
                preProcess[i][j-1]
                :preProcess[i+(1<<(j-1))][j-1];
    }

    int query(int start,int end)
    {
        int diff=end-start;
        diff=31 - __builtin_clz(diff+1);
        return A[preProcess[start][diff]]
          <=A[preProcess[end-(1<<diff)+1][diff]]?
          preProcess[start][diff]
          :preProcess[end-(1<<diff)+1][diff];
    }
};
```

You can simply use the above class for any type of variable(numeric ofcourse). Just keep in mind that you have to declare a preProcess array of size N x log N . So it turns out that the space complexity of the algorithm is **O(N log N)** and time complexity **T(NlogN, 1)**.

-fR0D

Written by fR0DDY

April 18, 2009 at 6:54 AM

Posted in Algorithm, Programming

Tagged with C, code, complexity, minimum, query, range, RMQ, space, time

Builtin Functions

leave a comment »

There are some useful bulitin function in gcc. Here are some of them :

— Built-in Function: int __builtin_ffs (unsigned int x)

Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.

— Built-in Function: int __builtin_clz (unsigned int x)

Returns the number of leading 0-bits in x, starting at the most significant bit position. If x is 0, the result is undefined.

— Built-in Function: int __builtin_ctz (unsigned int x)

Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined.

— Built-in Function: int __builtin_popcount (unsigned int x)

Returns the number of 1-bits in x.

— Built-in Function: int __builtin_parity (unsigned int x)

Returns the parity of x, i.e. the number of 1-bits in x modulo 2.

Try them out yourself. Here's a sample program.

```
int main()
{
    int x=30;
    printf("Index of least significant bit:%d\n",
    __builtin_ffs(x)-1);
    printf("Number of leading zeroes :%d\n",
    __builtin_clz(x));
    printf("Index of highest set bit:%d\n",
    31-__builtin_clz(x));
    printf("Number of trailing zeroes :%d\n",
    __builtin_ctz(x));
    printf("Number of 1-bits:%d\n",
    __builtin_popcount(x));
    printf("Parity:%d\n",__builtin_parity(x));


    printf("File Name:%s\n",__FILE__);
    printf("DATE:%s\n",__DATE__);
    printf("TIME:%s\n",__TIME__);
}
```

Also many times in graph algorithms we require +ve INFINITY and -ve INFINITY in our programs. There you can use :

```
int main()
{
    printf("+ve INFINTY: %d\n",INT_MAX);
    printf("+ve INFINTY: %d\n",(1<<31)-1);
    printf("-ve INFINTY: %d\n",INT_MIN);
    printf("-ve INFINTY: %d\n",1<<31);
}
```

PLAY WITH BITS!!!If you know others do share.

-fR0D

Written by fR0DDY

April 17, 2009 at 1:10 PM

Posted in Programming

Tagged with bits, builtin, code, functions, gcc, infinity, parity

C++ Templates

leave a comment »

Templates are very useful when implementing generic constructs like vectors, stacks, lists, queues which can be used with any arbitrary type. C++ templates provide a way to re-use source code as opposed to inheritance and composition which provide a way to re-use object code. Templates in C++ are of two types : function templates and class templates.

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

A classic example can be:

```
template <class myType>
myType GetMax (myType a, myType b)
{
	return (a>b?a:b);
}

int main ()
{
    long long a=10183273822013LL,b=10000011010021LL;
    printf("%d\n",GetMax<int>(10,12));
    printf("%lld\n",GetMax<long long>(a,b));
    return 0;
}
```

Other better example can be:

```
template< class T >
T gcd( T a, T b )
{
	return( b == 0 ? a : gcd( b, a % b ));
}

int main ()
{
    long long a=10183273823013LL,b=10000011010121LL;
    printf("%d\n",gcd<int>(10,12));
    printf("%lld\n",gcd<long long>(a,b));
    return 0;
}
```

Class templates can have members that use template parameters as types.For example:

```
template <class T>
class mypair
{
      T a, b;
      public:
             mypair (T first, T second)
             {
                  a=first; b=second;
             }
             T getmax ()
             {
                     return a>b? a : b;
             }
};

int main ()
{
    mypair<long long> p(1018333013LL,1000100121LL);
    printf("%lld\n",p.getmax());
    return 0;
}
```

The Standard Template Library generic algorithms have been implemented as function templates, and the containers have been implemented as class templates. Some of the basic algorithms which can be used are :

```
int main ()
{
    long long a=10183278223013LL,b=10000011000121LL;
    printf("Max :%lld\n",max(a,b));
    printf("Min :%lld\n",min(a,b));
    printf("a=%lld b=%lld\n",a,b);
    swap(a,b);
    printf("a=%lld b=%lld\n",a,b);
    return 0;
}
```

I will cover them in details in another post. Templates are very useful and a good programmer should always have this in his kitty.

-fR0D

Written by fR0DDY

April 16, 2009 at 5:26 PM

Posted in Programming

Tagged with C, class, code, function, generic, STL, templates

Programming Multiplicative Functions

with 8 comments

In number theory, a multiplicative function is an arithmetic function f(n) of the positive integer n with the property that f(1) = 1 and whenever a and b are coprime, then f(ab) = f(a)f(b).

This type of functions can be programmed in quick time using the multiplicative formmula. Examples of multiplicative functions include Euler's totient function, Möbius function and divisor function.

A multiplicative function is completely determined by its values at the powers of prime numbers, a consequence of the fundamental theorem of arithmetic. Thus, if n is a product of powers of distinct primes, say $n = p^a q^b \ldots$, then $f(n) = f(p^a) f(q^b) \ldots$

Lets take an example. If we need to calculate the divisor of 12 then $div(12) = div(2^2 3^1)$ or $div(2^2)div(3^1)$. If we are calculating it for a range then it becomes even more easier. We only need to calculte the $div(2^2)$part because the $div(3^1)$ part will already be present in the array. So lets look at an sample code for calculating the divisor of all numbers from 1 to 10000000.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int A[10000000]={0};
int main()
{

    int isprime[3163],d,n,e,p,k,i;

    for (n=2;n<3163;n++)
        isprime[n]=1;

    //Sieve for Eratosthenes for Prime
    //Storing the smallest prime which divides n.
    //If A[n]=0 it means it is prime number.
    for(d=2;d<3163;d++)
    {
        if(isprime[d])
        {
            for (n=d*d;n<3163;n+=d)
            {
                isprime[n]=0;
                A[n]=d;
            }
            for (;n<=10000000;n+=d)
```

```
                A[n]=d;
        }
    }

    //Applying the formula
    //Divisor(N)=Divisors(N/p^f(N,p))*(f(N,p)+1)
    A[1]=1;
    for(n=2;n<=10000000;n++)
    {
        if (A[n]==0)
           A[n]=2;
        else
        {
           p=A[n],k=n/p,e=2;
           while (k%p==0)
                 k/=p,e++;
           A[n]=A[k]*e;
        }
    }
    printf("time=%.3lf sec.\n",
    (double) (clock())/CLOCKS_PER_SEC);
    while (scanf("%d",&i),i)
    {
          printf("%d\n",A[i]);
    }
    return 0;
}
```

If you run the above program you will be able to see how fast have we made this program. Similar techniques can be used to calculate the divisor function or the sum of squares of divisors which is also called sigma2. The general formula for calculating sigma function is :

$$\sigma_x(n) = \sum_{d|n} d^x.$$

sigma_x

or

$$\sigma_x(n) = \prod_{i=1}^{r} \frac{p_i^{(a_i+1)x} - 1}{p_i^x - 1}$$

-fR0DDY

Written by fR0DDY

April 13, 2009 at 6:31 AM

Posted in Maths, Programming

Tagged with code, Divisor function, Euler, Euler's totient function, Möbius function, Multiplicative functions, prime, sigma

Maximum Subarray in 1-D and 2-D Array

with one comment

This is a well known problem wherein we have to find the subarray whose sum is maximum. For 1-D array the fastest time that this can be done is in O(n). The algorithm was given by by Jay Kadane of Carnegie-Mellon University. It can be found here.
Its C++ implementation would look like this.

```
int MaxSum1D(vector M)
{
    int N=M.size(),i,t=0,S=1<<31;
    for (i=0;i<N;i++)
    {
        t=t+M[i];
        S=max(t,S);
        if (t<0)
           t=0;
    }
    return S;
}
```

For finding the maximum subarray in a 2-D array the brute force method would take O(n^6) time. The trivial solution to it would take O(n^4) time which should be good enough for most questions on online judges. Its C++ implementation would be:

```
int main()
{
    int N,maxsum=1<<31,i,j,k,l,m,sum,p;
    scanf("%d",&N);
    vector< vector > M(N+1,vector(N+1,0));
    vector< vector > S(N+1,vector(N+1,0));
    for (i=1;i<N+1;i++)
        for (j=1;j<N+1;j++)
            scanf("%d",&M[i][j]);

    for (i=1;i<N+1;i++)
        for (j=1;j<N+1;j++)
            S[i][j]=M[i][j]+
            S[i-1][j]+S[i][j-1]-S[i-1][j-1];

    maxsum=-128;
    for (i=1;i<N+1;i++)
        for (j=1;j<N+1;j++)
            for (k=i;k<N+1;k++)
                for (l=j;lmaxsum)
                        maxsum=sum;
                }
    printf("%d\n",maxsum);
}
```

But kadane gave a O(n^3) solution for it too. In this algorithm we calculate the prefix sum for all possible row combination in O(n^2) and then take out their maximum contigous sum in O(n) time. Thus doing the task in O(n^3) time. C++ implementation would be :

```
int MaxSum2D(vector< vector > M)
{
    int S=1<<31,k,j,i,t,s;
    int N=M.size();

    for (i=0;i<N;i++)
    {
        vector pr(N,0);
        for (j=i;j<N;j++)
        {
            t=0;s=1<<31;
            for (k=0;k<N;k++)
            {
                pr[k]=pr[k]+M[j][k];
                t=t+pr[k];
                s=max(t,s);
                if (t<0)
                   t=0;
            }
            S=max(S,s);
        }
    }
    return S;
}
```

For finding the minimum and the position of the subarrays slight changes need to be made to the codes.

-fR0D

Written by fR0DDY

April 7, 2009 at 1:52 PM

Posted in Algorithm, Programming

Tagged with 1-D, 2-D, array, C, code, complexity, Kadane, Maximum, minimum, sum

Pell Equation

leave a comment »

Pell's equation is any Diophantine equation of the form

$$x^2 - Dy^2 = 1$$

where D is a nonsquare integer and x and y are integers. Lagrange proved that for any natural number D that is not a perfect square there are x and $y > 0$ that satisfy Pell's equation. Moreover, infinitely many such solutions of this equation exist.

There are many ways to get the solutiont to these equations some of which are given on Wikipedia, MathWorld and probably a good explanation here.

Though at first read it may seem difficult to understand the solution. The basic aim here is to get the convergents of Sqrt(D). Sqrt(D) = $[a_0,a_1,a_2,\ldots,a_r]$ such that $a_{r+1}=2a_0$.

Very important point is that if r is even then solution is p_{2r+1} where p_n is the numerator in nth convergent.

The following function is a direct application of the logic given on Mathworld and works for all D<=1000.

```
double PellSolution(double D)
{
	double x1,y1;
	double Pn[100],Qn[100],a[100],p[100],q[100];
	long long n,r;
	if (sqrt(D)!=floor(sqrt(D)))
	{
	   //Initialization
	   Pn[0]=0;Qn[0]=1;
	   a[0]=floor(sqrt(D));
	   p[0]=a[0];q[0]=1;

	   n=1;
	   Pn[1]=a[0];Qn[1]=D-a[0]*a[0];
	   a[1]=floor((a[0]+Pn[1])/Qn[1]);
	   p[1]=a[0]*a[1]+1.0;
	   q[1]=a[1];

	   while (a[n]!=2.0*a[0])
	   {
		n=n+1;
		Pn[n]=a[n-1]*Qn[n-1]-Pn[n-1];
		Qn[n]=(D-Pn[n]*Pn[n])/Qn[n-1];
		a[n]=floor((a[0]+Pn[n])/Qn[n]);
		p[n]=a[n]*p[n-1]+p[n-2];
		q[n]=a[n]*q[n-1]+q[n-2];
	   }
	   r=n-1;
	   if (r%2==0)
	   {
	      for (n=r+2;n<=2*r+1;n++)
	      {
		  Pn[n]=a[n-1]*Qn[n-1]-Pn[n-1];
		  Qn[n]=(D-Pn[n]*Pn[n])/Qn[n-1];
		  a[n]=floor((a[0]+Pn[n])/Qn[n]);
		  p[n]=a[n]*p[n-1]+p[n-2];
		  q[n]=a[n]*q[n-1]+q[n-2];
	      }
	      x1=p[2*r+1];
```

```
            y1=q[2*r+1];
        }
        else
        {
            x1=p[r];
            y1=q[r];
        }
        return x1;
    }
    else
        return 0;
}
```

Note that the above program only returns the value of x for fundamental solution. To get fundamental y you can return y1. Also to get the nth solution u can use the following recurrence relation
x_nth=((x1+y1*Sqrt_D)^nth + (x1-y1*Sqrt_D)^nth)/2
y_nth=((x1+y1*Sqrt_D)^nth – (x1-y1*Sqrt_D)^nth)/(2*Sqrt_D)

-fR0D

Written by fR0DDY

April 5, 2009 at 9:36 AM

Posted in Maths, Programming

Tagged with C, code, Pell

Prime Numbers

with 7 comments

Suppose that you have to find all prime numbers till say some value N or let’s say sum of all prime numbers till N. How would you proceed. Let’s play around a bit. A grade 5 student would tell you that a prime has only two divisors 1 and itself and would code it like this :

```
void basic1(int N)
{
     int i,j,k,c;
     long long s=0;
     for (i=1;i<=N;i++)
     {
         c=0;
         for (j=1;j<=i && c<3;j++)
             if (i%j==0)
                c++;
         if (c==2)
            s+=i;
     }
     cout<<s<<endl;
}
```

A little smarter kid would tell you that a prime number has no divisor between 2 and N/2 and would code it like this :

```
void basic2(int N)
{
     int i,j,k,F;
     long long s=0;
     for (i=2;i<=N;i++)
     {
         F=1;
         for (j=2;j<=i/2 && F==1;j++)
             if (i%j==0)
                F=0;
         if (F)
            s+=i;
     }
     cout<<s<<endl;
}
```

An even smarter kid will tell you that a prime number has no divisor between 2 and its root and would code it like this :

```
void moderate1(int N)
{
     vector<int> p(N/2,0);
     int i,j,k,F,c=0;
     long long s=0;
     for (i=2;i<=N;i++)
     {
         F=1;
         for (j=2;j*j<=i && F==1;j++)
             if (i%j==0)
                F=0;
         if (F)
         {
            p[c++]=i;
            s+=i;
         }
     }
     cout<<s<<endl;
}
```

A good maths student can tell you that a prime number has no prime divisors between 2 and its root.

```
void moderate2(int N)
{
     vector<int> p(N/2,0);
     p[0]=2;
     int i,j,F,c=1;
     long long s=2;
     for (i=3;i<=N;i+=2)
     {
         F=1;
         for (j=0; p[j]*p[j]<=i && F==1; j++)
             if (i%p[j] == 0)
                F=0;
         if (F)
         {
            p[c++]=i;
            s+=i;
         }
     }
     cout<<s<<endl;
}
```

A good programmer will tell you that Sieve of Eratosthenes is the best way to find the list of prime numbers.

Its algorithm is as follows :

1. Create a contiguous list of numbers from two to some highest number n.
2. Strike out from the list all multiples of two (4, 6, 8 etc.).
3. The list's next number that has not been struck out is a prime number.
4. Strike out from the list all multiples of the number you identified in the previous step.

5. Repeat steps 3 and 4 until you reach a number that is greater than the square root of n (the highest number in the list).
6. All the remaining numbers in the list are prime.

And the C++ implementation would be :

```
void sieve(int N)
{
     int x=sqrt(N),i,j;
     vector<bool> S(N+1,0);
     for (i=4;i<=N;i+=2)
         S[i]=1;
     for (i=3;i<=x;i+=2)
         if (!S[i])
            for (j=i*i;j<=N;j+=2*i)
                S[j]=1;

     long long s=0;
     for (i=2;i<=N;i++)
         if (!S[i])
            s+=i;

     printf("%lld\n",s);
}
```

But then we can optimize even the Sieve of Eratosthenes. If you look closer you will realise that apart from 2 all the other even numbers are of no use and just waste our time and memory. So we should get rid of them. We can do that by sieving only odd numbers. Let an element with index i correspond to number 2*i+1. So in the sieve we only need to go upto (N-1)/2. Also if p = 2*i+1 p^2 = 4*i^2 + 4*i + 1 which is represented by index 2*i*(i+1). Also the step will be of order 2*i+1. So the code would look something like this:

```
void improved_sieve(int N)
{
     int M=(N-1)/2;
     int x=(floor(sqrt(N))-1)/2,i,j;
     vector<bool> S(M+1,0);
     for (i=1;i<=x;i++)
         if (!S[i])
            for (j=2*i*(i+1);j<=M;j+=(2*i+1))
                S[j]=1;

     long long s=2;
     for (i=1;i<=M;i++)
         if (!S[i])
            s+=(2*i+1);

     printf("%lld\n",s);
}
```

and since i am learning Python. Here's the python code as well :

```
import math
def improved_sieve(N):
    M=(N-1)/2
    x=(int(math.sqrt(N))-1)/2
    S=[]
    for i in range(M+1):
        S.append(False);

    for i in range (1,x+1):
        if (S[i]==False):
            for j in range (2*i*(i+1),M+1,2*i+1):
                S[j]=True

    s=2;
    for i in range (1,M+1):
        if (S[i]==False):
            s+=(2*i+1)

    print s

N=input()
improved_sieve(N)
```

Lets compare all these codes on runtime :

Limit10^N where N=	basic1	basic2	moderate1	moderate2	sieve	improved_sieve
4	0.079	0.026	0.003	0.007	0.002	0.001
5	7.562	2.611	0.050	0.117	0.020	0.011
6	–	–	1.160	2.218	0.208	0.121
7	–	–	26.350	44.946	2.183	1.269

All time are in seconds. If you notice that though algorithm 4 seems to be better than 3 it takes more time because of the array refrences that are made several times. Choose the best.

Happy Coding!
-fR0D

Written by fR0DDY

March 10, 2009 at 3:34 PM

Posted in Maths, Programming

Tagged with C, eratosthenes, prime, Python, sieve

Google Interview Questions

with 3 comments

All the below questions are to be done in O(n) time complexity.

1>Given an array of size n-1 whose entries are integers in the range [1, n], find an integer that is missing. Use only O(1) space and treat the input as read-only.

2>Given an array of size n-2 whose entries are integers in the range [1, n], find two integer that are missing.

3>There is an array A[N] of N integers. You have to compose an array B[N] such that Output[i] will be equal to the product of all the elements of A[] except A[i].

$$B_i = \prod_{j \neq i} A_j$$

or

Example:
INPUT:[4, 3, 2, 1, 2]
OUTPUT:[12, 16, 24, 48, 24]

Solve it without division operator and in O(n).

Solution :

1>Let the missing number be M. We know that the sum of first N natural numbers is N*(N+1)/2. Traverse through the array once and calculate the sum. This is the sum of First N natural numbers – M or S=N*(N+1)/2 – M. Therefore M = N*(N+1)/2 – S.

2>Similar approach to the first one. Traverse the array once and calculate its sum and multiplication. Let the sum be S and multiplication be M. Let the missing numbers be P and Q. From above we know that P+Q = N*(N+1)/2 – S. Also P*Q = N!/M. So we can get P and Q from these two equations.

3> Lets first see the C++ solution.

```
void solution(vector<int> A)
{
    int N=A.size(),i,j;
    vector<int> L(N,0);
    vector<int> R(N,0);
    vector<int> B(N,0);
    for (i=0,j=N-1; i<N && j>=0 ;i++, j--)
    {
        L[i] = i==0? 1 : A[i-1] * L[i-1];
        R[j] = j==N-1? 1 : R[j+1] * A[j+1];
    }

    for (i=0; i<N ; i++)
    {
        B[i] = L[i] * R[i];
        printf("%d ",B[i]);
    }
}
```

Most is clear from the program. Anyways, through the L and R we calculate the multiplication of terms to the left and right of i-th term. then finally we multiply it to get the result.

-fR0D

Written by fR0DDY

March 5, 2009 at 3:35 PM

Posted in Programming

Tagged with C, complexity, Google, Interview, Questions, time

Just Another Question

with 8 comments

Here's an interesting Question :
Look at the following code. Its to print '*' 20 times.

```
#include<iostream>
using namespace std;
int main()
{
    int i,n=20;
    for (i=0;i<n;i--)
        cout<<"*";
    return 0;
}
```

Obviously the code above is wrong. You have to find three ways of correcting the program either by replacing a single character in the code with another character or adding a single character to the given code.
Try it. It's fun.

Answer 1: Change i<n to -i<n
Answer 2: Change i- – to n- –
Answer 3: Change i<n to i+n

Credit for this question goes to my roommate who gave me this(though he too got it from somewhere else).

-fR0D

Written by fR0DDY

March 5, 2009 at 6:09 AM

Posted in Uncategorized

Tagged with C, fun

Evaluation of Powers

with 2 comments

This is a very interesting problem with a lots of history. Anyways we will not wonder into it. We shall see how fast can we calculate x^n, given x and n where n is a positive integer. The brute force method would be to run a loop from 2 to n and calculate in n-1 steps. We will discuss three methods to do it quickly.

Binary Method
This the most common method used in programs today. It is also called the Dynamic Programming method. Here is an C++ implementation :

```
int binary(int x,int n)
{
    if (n==1)
       return x;
    if (n%2==0)
    {
       int t=binary(x,n/2);
       return t*t;
    }
    else
        return x*binary(x,n-1);
}
```

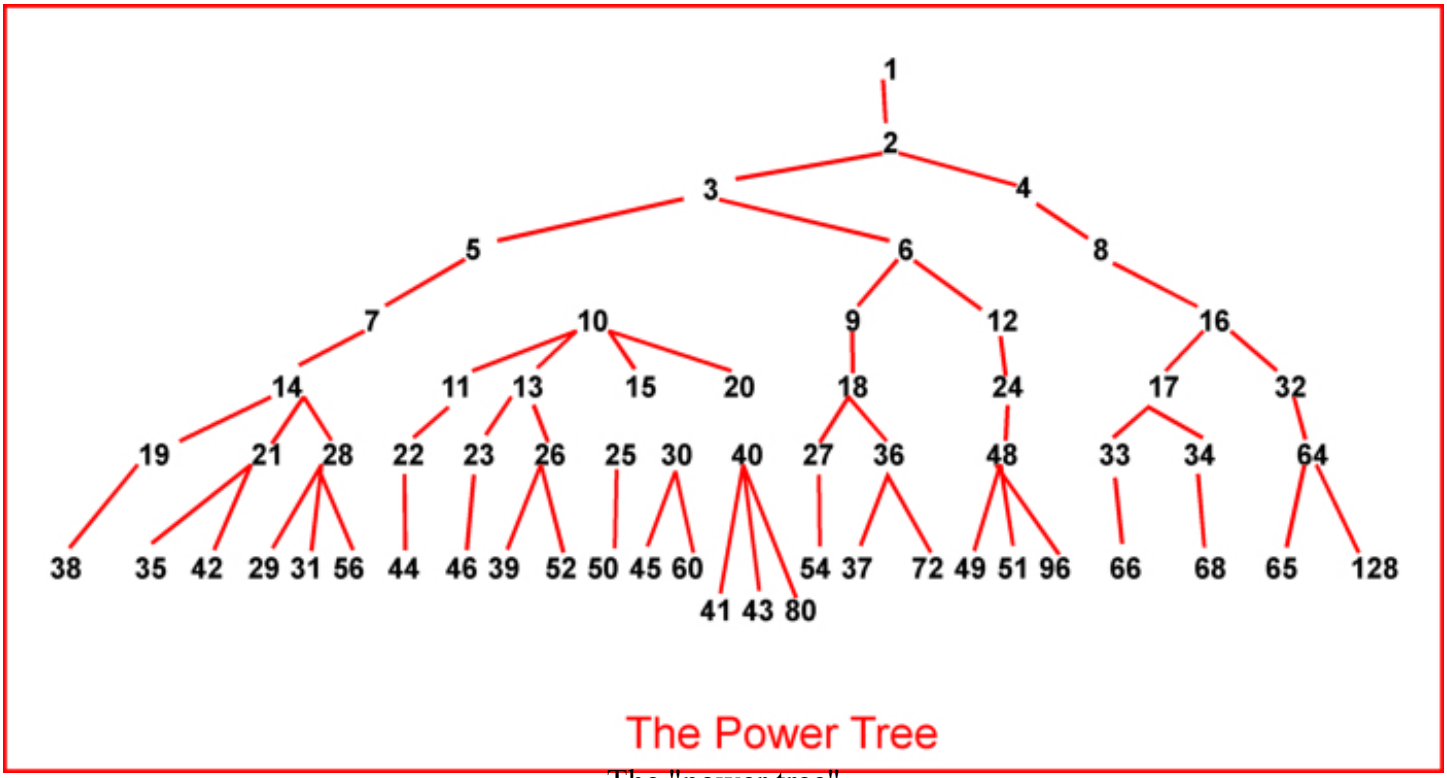
But does this method give the minimum number of multiplications. The answer is **NO**.
The smallest counterexample is n=15. The binary method takes six multiplications but the best method can do it in five multiplications. We can calculate y = x^3 in two multiplications and $x^{15} = y^5$ in three more, needing only five multiplications in total.

Factor Method

Let n=pq where p is smallest prime factor of n. We can calculate n by first calculating p and then raising this quantity to q-th power. If n is prime we calculate x^{n-1} and multiply by x. For example to calculate x^{55}, we first calculate y = $x^5 = x^4$x = $(x^2)^2$x; then we form $y^{11} = y^{10}$y = $(y^2)^5$y. The whole process takes eight multiplications.
But does this method give the minimum number of multiplications. The answer is **NO**.
The smallest counterexample is n=33. The factor method takes seven multiplications but the best method(binary method) can do it in six multiplications.

The "Power Tree Method"

Lets first see the power tree.



The "power tree"

Figure above shows the first few levels of the “power tree.” The (k + 1)-st level of this tree is defined as follows, assuming that the first k levels have been constructed: Take each node n of the lath level, from left to right in turn, and attach below it the nodes

$$n + 1, n + a_1, n + a_2, \ldots, n + a_{k-1} = 2n$$

(in this order), where $1, a_1, a_2, \ldots, a_2$ is the path from the root of the tree to n;but discard any node that duplicates a number that has already appeared in the tree. The C++ implementation of the above can be :

```
/** LINKU[j], LINKR[j] for 0 <= j <= 2^r;
   point upwards and to the right, respectively,
   if j is a number in the tree
**/
int LINKU[2050]={0},k=0,LINKR[2050]={0},q,s,nm,m,i,n;
LINKR[0]=1;
LINKR[1]=0;
//11 being number of level
while (k < 11)
{
    n=LINKR[0];m=0;
    do
    {
        q=0,s=n;
        do
        {
            if (LINKU[n+s]==0)
            {
                if (q==0)
                   nm=n+s;
                LINKR[n+s]=q;
                LINKU[n+s]=n;
                q=n+s;
            }
            s=LINKU[s];
        }while (s!=0);
        if (q!=0)
        {
            LINKR[m]=q;
            m=nm;
        }
        n=LINKR[n];
    }while (n!=0);
    LINKR[m]=0;
    k=k+1;
}
```

But does this method give the minimum number of multiplications. The answer is **NO**.
The smallest counterexample is n=77.Other examples are 154 and 233.

Some Analysis

- The first case for which the power tree is superior to both the binary method and the factor method is

23.

- The first case for which the factor method beats the power tree method is 19879 = 103*193. Such cases are rare, only 6 for n<10^5.
- Power tree never gives more multiplications for the computation of x^n than the binary method.

If You are not interested in Maths stop reading here.

Let l(n) denote the minimum number of multiplications for a given number n.
λ(n) = floor(lg n), where floor(x) is the largest integer not greater than x.
v(n) = number of 1s in the binary representation of n.

They follow the following recurrence relations :

λ(1)=0, λ(2n) = λ(2n+1) = λ(n) + 1;
v(1)=1, v(2n)=v(n), v(2n+1) = v(n)+1.

The binary method requires λ(n) + v(n) – 1 steps.

So we have following theorems

- l(n)<= λ(n) + v(n) -1.
- l(n)>= ceiling(lg n), where ceiling(x) is the smallest integer not less than x.
- l(2^A) = A.
- l($2^A + 2^B$) = A+1 if A>B.
- l($2^A + 2^B + 2^C$) = A+2 if A>B>C.

Conjecture

- l(2n) = l(n) +1; It fails since l(191) = l(382) = 11.
- The smallest four values for which l(2n) = l(n) are n = 191,701,743,1111.

Conclusion

The table of l(n) may be prepared for 2<=n<=1000 by using the formula l(n) = min(l(n-1)+1 , l_n) – δ_n, where $l_n = \infty$ if n is prime, otherwise l_n = l(p)+l(n/p) if p is the smallest prime dividing n; and δ_n = 1 for n in Table below and 0 otherwise.

23	163	229	319	371	413	453	553	599	645	707	741	813	849	903
43	165	233	323	373	419	455	557	611	659	709	749	825	863	905
59	179	281	347	377	421	457	561	619	667	711	759	835	869	923
77	203	283	349	381	423	479	569	623	669	713	779	837	887	941
83	211	293	355	382	429	503	571	631	677	715	787	839	893	947
107	213	311	359	395	437	509	573	637	683	717	803	841	899	955
149	227	317	367	403	451	551	581	643	691	739	809	845	901	983

Phew!!! That was long.

-fR0D
(notes from TAOCP written by Donald E Knuth)

Links : http://www.research.att.com/~njas/sequences/a003313.txt http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html

Written by fR0DDY

March 2, 2009 at 3:54 PM

Posted in Maths, Programming

Tagged with Binary, code, factor, Knuth, Maths, power, TAOCP

Farey Sequence/Series

with 2 comments

Farey Sequence of order **N** is the ascending sequnce of all reduced fractions between **0** and **1** that have denominators **<=N.** For example, the Farey Series of order 4 is

$0/1, 1/4, 1/3, 1/2, 2/3, 3/4, 1/1$

This series can be generated using the following recurrence relation

$x_0=0 \quad y_0=1$
$x_1=1 \quad y_1=N$
$x_{k+2}= \text{floor}((y_k+N)/y_{k+1})x_{k+1}- x_k$
$y_{k+2}= \text{floor}((y_k+N)/y_{k+1})y_{k+1}- y_k$

The C++ Implementation would look something like this :

```
x1=0,y1=1,x2=1,y2=N;
x=1;y=N;
printf("%.0lf/%.0lf, %.0lf/%.0lf",x1,y1,x2,y2);
while (y!=1.0)
{
    x=floor((y1+N)/(y2))*x2-x1;
    y=floor((y1+N)/(y2))*y2-y1;
    printf(", %.0lf/%.0lf",x,y);
    x1=x2,x2=x,y1=y2,y2=y;
}
printf("\n");
```

Note that the number of terms in the Farey Series is given by :

$$|F_n| = 1 + \sum_{m=1}^{n} \varphi(m)$$

where phi(m) is the Euler Toteint Function.

Did You Know? 77 x 999 = 777 x 99 and 112 x 112 = 12544 and its back order 211 x 211 = 44521

NJOY!!!
-fR0D

Written by fR0DDY

March 1, 2009 at 10:04 AM

Posted in Maths, Programming

Tagged with C, Euler, Farey, floor, length, Maths

Binary GCD Algorithm

with one comment

Most of us know that to find the GCD of two numbers, Euclid Algorithm is the best algorithm. But thanks to **Josef Stein**, there is the Binary GCD Algorithm which is touch faster than the Euclid's Algorithm.

The Euclids Algorithm implemented is :

```
long long gcd(long long a, long long b)
{
    if(a==0) return(b);
    return(gcd(b%a, a));
}
```

The Binary GCD Algorithm as given in the The Art of Computer Programming written by D.E.Knuth is as follows :

```
Given positive integers u and v,
this algorithm finds their greatest common divisor.
A1. [Find power of 2.] Set k←0, and then repeatedly set k← k + 1,
u ← u/2,v ← v/2, zero or more times until u and v are not both even.
A2. [Initialize.] (Now the original values of u and v have
been divided by 2^k,and at least one of their present values is odd.)
If u is odd, set t  ← -v, and go to A4. Otherwise set t ← u.
A3. [Halve t.] (At this point, t is even, and nonzero.) Set t ← t/2.
A4. [Is t even?] If t is even, go back to A3.
A5. [Reset max(u, v).] If t > 0, set u ← t; otherwise set v←  -t.
(The larger of u and v has been replaced by |t|,
except perhaps during the first time this step is performed.)
A6. [Subtract.] Set t ← u-v. If t != 0, go back to A3.
Otherwise the algorithm terminates with u* 2^k as the output.
```

Implementation in C++ would look something like this :

```
int binarygcd(int u,int v)
{
    int k=0,t=0,i;
    while (!(u&1) && !(v&1))
    {
          k++;
          u>>=1;
          v>>=1;
    }
    if (u&1)
       t=u;
    else
        t=-v;
    do
    {
        while (!(t&1))
              t>>=1;
        if (t>0)
           u=t;
        else
            v=-t;
        t=u-v;
    }while (t);
    for (i=0;i<k;i++)
        u<<=1;
    return(u);
}
```

Though this program may seem to be longer and have more iterations but still its faster because it requires no division instruction, it depends only on subtraction,parity checking and halving using bit shifting.

-fR0D

Written by fR0DDY

February 26, 2009 at 12:08 PM

Posted in Maths, Programming

Tagged with Binary, GCD, Knuth, TOACP

Sum of First n m-th Powers

with 2 comments

This is going to be long even without the proof.
The sum of the first n m-th powers is given by the formula :

$$T(m,n) = \sum_{k=1}^{m} S2(m,k)*f(n+1,k+1)/(k+1).$$

sum of first n m-th powers

where S2 is Stirling numbers of the Second Kind :

m\k| 1 2 3 4 5 6
———————————-
1 | 1
2 | 1 1
3 | 1 3 1
4 | 1 7 6 1
5 | 1 15 25 10 1
5 | 1 31 90 65 15 1

given by the recurrence relation :

for m >= 1,

S2(m,0) = 0,
S2(m,1) = 1,
S2(m,k) = 0 for all all k > m,
S2(m+1,k) = S2(m,k-1) + k*S2(m,k).

and **f(x,k) = x*(x-1)*(x-2)*…*(x-k+1).**

Let us take two examples :

T(2,n) = S2(2,1)*f(n+1,2)/2 + S2(2,2)*f(n+1,3)/3,
= 1*(n+1)*n/2 + 1*(n+1)*n*(n-1)/3,
= n*(n+1)*(1/2 + [n-1]/3),
= n*(n+1)*(1/2 + n/3 – 1/3),
= n*(n+1)*(2*n+1)/6.
and

T(4,n) = S2(4,1)*f(n+1,2)/2 + S2(4,2)*f(n+1,3)/3 +
S2(4,3)*f(n+1,4)/4 + S2(4,4)*f(n+1,5)/5,
= 1*(n+1)*n/2 + 7*(n+1)*n*(n-1)/3 +
6*(n+1)*n*(n-1)*(n-2)/4 +
1*(n+1)*n*(n-1)*(n-2)*(n-3)/5,
= n*(n+1)*(1/2 + 7*[n-1]/3 + 3*[n-1]*[n-2]/2 +
[n-1]*[n-2]*[n-3]/5),
= n*(n+1)*(1/2 + 7*[n-1]/3 + 3*[n^2-3*n+2]/2 +
[n^3-6*n^2+11*n-6]/5),
= n*(n+1)*(1/2 + 7*n/3 – 7/3 + 3*n^2/2 – 9*n/2 + 3 +
n^3/5 – 6*n^2/5 + 11*n/5 – 6/5),
= n*(n+1)*(n^3/5 + 3*n^2/10 + n/30 – 1/30),
= n*(n+1)*(6*n^3+9*n^2+n-1)/30,
= n*(n+1)*(2*n+1)*(3*n^2+3*n-1)/30.

Try higher powers for yourself.
-fR0D

Written by fR0DDY

February 26, 2009 at 9:32 AM

Posted in Maths

Tagged with Maths, power, stirling, sum

Fibonacci Numbers

with 5 comments

Fibonacci Numbers are one of the most common sequence in maths and computer science. One of the question is how fast can you calculate Fib(N). The obvious time complexity to calculate this would be O(N) but thanks to late Prof. Edsgar W Dijkstra there exists an O(log N) solution. He gave this two equations :

```
F(2n-1) = F(n-1)^2 + F(n)^2
F(2n) = ( 2 F(n-1) + F(n) ) F(n)
```

I have written a program which implements it in time O(log N) but for simplicity reasons the space complexity is O(N). Also the program is capable of calculating the answers only till index 92 which is within the **long long** range. It sure can be implemented for big integers library.

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

long long F[93];
long long Fib(int N)
{
    if (N==1)
       return 1;
    if (N==0)
       return 0;
    else
    {
        if (F[(N+1)/2-1]==0)
           F[(N+1)/2-1]=Fib((N+1)/2-1);
        if (F[(N+1)/2]==0)
           F[(N+1)/2]=Fib((N+1)/2);
        if (N%2==0)
           return
           ((2*F[(N+1)/2-1]+F[(N+1)/2])*F[(N+1)/2]);
        else
            return
            ((F[(N+1)/2-1]*F[(N+1)/2-1]+
            F[(N+1)/2]*F[(N+1)/2]));
    }
}

int main()
{
    int N;
    while (scanf("%d",&N),N>=0&&N<93)
    {
          printf("%lld\n",Fib(N));
    }
}
```

-fR0D

Written by fR0DDY

February 24, 2009 at 1:56 PM

Posted in Maths, Programming

Tagged with C, Fibonacci, Maths

Number System Conversion

with 3 comments

I gave this question in one of the contests to interconvert between decimal,octal and hexdecimal numbers **without traversing the digits**. The numbers could be random like 12 0xA 15 o12. You have to print the number in other two systems except from the one entered. The program would be terminated by a negative number. The solution i gave uses sscanf and was as following :

```
int main()
{
    char str[20];
    int dec,l,i,n;
    while (1)
    {
          scanf("%s",str);
          if (str[0]=='-')
             break;
          else if (str[1]=='x')
          {
               str[0]=str[1]=' ';
               sscanf(str,"%X",&amp;n);
               printf("%d o%o\n",n,n);
          }
          else if (str[0]=='o')
          {
               str[0]=' ';
               sscanf(str,"%o",&amp;n);
               printf("%d 0x%X\n",n,n);
          }
          else
          {
              n=atoi(str);
              printf("0x%X o%o\n",n,n);
          }
    }
}
```

If you have a better solution please post it in the comments.

-fR0D

Written by fR0DDY

February 23, 2009 at 12:55 PM

Posted in Maths, Programming

Tagged with C, conversion

Bit Play

with 2 comments

The first one is to add two integers using bit operators.

```
int add(int a,int b)
{
    if(!a)
          return b;
    else
        return(add((a&b)<<1,a^b));
}
```

The next is to find maximum and minimum of two numbers :

```
r = y + ((x - y) & -(x < y)); // min(x, y)
r = x - ((x - y) & -(x < y)); // max(x, y)
where r is the result.
```

NJOY!!!
-fR0D

Written by fR0DDY

February 23, 2009 at 5:13 AM

Posted in Beautiful Codes

Tagged with bit, C

Program without main() ?

with one comment

See this code first :

```
#include"stdio.h"
#define decode(s,t,u,m,p,e,d) m##s##u##t
#define begin decode(a,n,i,m,a,t,e)

int begin()
{
    printf(" hello ");
}
```

Compile and run this. It works fine but how?Actually its a very good example of the **preprocessor directive #define** and **token pasting or token merging operator '##'**. Kool program isn't it?

-fR0D

Written by fR0DDY

February 18, 2009 at 12:54 PM

Posted in Beautiful Codes

Tagged with C, main()

Size Contest

with 7 comments

Looks like i am having some affair with shorter codes. Anyways, there is this question on SPOJ. You have to write the shortest code for finding the sum of all positive integers,given a set of T integers followed by T.

As always i am far away from optimal result :

```
#include"iostream"
int a,t,s;
main()
{
      for(std::cin>>t;t--;std::cin>>a,s+=a>0?a:0);
      std::cout<<s;
}
```

And Python Solution :

```
t=input()
s=0
while t:
    x=input()
    s+=x*(x>0)
    t-=1
print s
```

If you have a shorter C++ or Python code, help me out. Between try this line to see the path of your file.

```
printf("The source file name is %s\n",__FILE__);
```

-fR0D

Written by fR0DDY

February 17, 2009 at 5:18 PM

Posted in Beautiful Codes, Programming

Tagged with C, SPOJ

Binary Palindromes

with 4 comments

There is this question from a programming competition which requires you to find whether the binary representation of a number is palindrome or not. The input begins with integer T followed by T cases.

The obvious code would be(my solution) :

```
main(v,r,t,x)
{scanf("%d",&t);while(t--){
for(scanf("%d",&v),x=v,r=0;v;r<<=1,r|=v&1,v>>=1){}
puts(r-x?"NO":"YES");}}
```

But there were better(shorter) solutions which used recursive main functions :

```
a,t;main(c,b)
{c?main(c/2,b*2+c%2):
t++/2&&puts(a-b?"NO":"YES"),
~scanf("%d",&a)&&main(a,0);}
```

-fR0D

Written by fR0DDY

February 13, 2009 at 9:48 AM

Posted in Beautiful Codes

Tagged with C

My Digital Signature

with 6 comments

One day while surfing i reached this page. After looking at the quote there i thought why not create one for myself. So i set on to make my signature but it wasn't easy. I learned new things while making it though. Anyways here's my digital signature :

```
double m[]={4910923072864513.0,819};
main(){1[m]--?*m*=2.,main():printf((char*)m);}
```

This was a learning process which started with this :

```
long long x=94838554181959LL;int main(){printf((char*)&x);}
```

went on to this :

```
int x=5570631,y=5636178,f=1;
main(){f?f^=f,x+=(65<<8),printf((char*)&x),main():
 (y+=(65<<8),printf((char*)&y));}
```

And finally reached my digital signature above. Hope to create an even more complex one when i am free next time. Again you need to compile them using gcc, g++ won't work.

-fR0D

Written by fR0DDY

February 11, 2009 at 4:48 PM

Posted in Uncategorized

Power of 2

with 2 comments

There was an online competition which required to find whether an integer(<2^31) entered is a power of 2 or not. The condition was that code should be as small as possible and should not use any semicolons.The input is terminated by end of file. What me and my room mate could get to was this :

main(int N){while(scanf("%d",&N)+1&&puts(N&N-1?"no":"yes")){}}

Though the best solution turned out to be this which was five character less than our solution :

main(i){while(~scanf("%d",&i)&&puts(i&i-1?"no":"yes")){}}

Both are C programs, so you need to compile it using gcc(g++ won't work).

-fR0D

Written by fR0DDY

February 10, 2009 at 6:23 PM

Posted in Beautiful Codes

Tagged with C

Tower of Hanoi

with 3 comments

This is a C program to solve the very famous Tower of Hanoi problem. It neither uses recursion nor stack. Moreover, it solves the problem in just 6 statements. See the indenting done to make it look like a tower.

```
main(
){int
z,y,n
;scanf("%d",&n);
for(y=1;(1<<n)-y
;y<<=z-1,printf(
"Move disk %i from %i to %i.\n"
,z,(y&y-1)%3+1,((y|y-1)+1)%3)+1,y
++)for(z=1;!(y&1);z++,y>>=1);}/****/
```

Found it while googling.

-fR0D

Written by fR0DDY

February 10, 2009 at 9:31 AM

Posted in Beautiful Codes

Tagged with C, Tower of Hanoi

SUDOKU

with one comment

SUDOKU's are one of the most engrossing pen and paper puzzles i have solved. I love them and so one odd day i thought why not make a program to solve them. I knew that many such programs already existed but i wanted to write my own version. SUDOKU's are known to be NP complete problems (a type of exact cover problem). What i did to optimize it is that i don't start the backtracking right away. First i try to solve it logically till the point you are left with no option but to guess among choices. This alone is sufficient to solve easy and some medium level problems. Once logically filling is over i start to fill it recursively using backtracking which is required for hard and fiendish puzzles. This isn't a very good implementation and may contain bugs. You can have a look at it here . Corrently it takes input from file named inputsudoku.txt but you can comment it out and give input from command line. Put 0 in places where no number is present in the sudoku.

-fR0D

Written by fR0DDY

February 9, 2009 at 5:29 PM

Posted in Programming

Tagged with Sudoku

OEIS

with one comment

OEIS or The **On-Line Encyclopedia of Integer Sequences** is a site containing more than **150000** sequences. I had few sequences in mind since some time but didn't know where to look for them. So few days back when i came to know about this site i searched them on it. To my surprise i found most of them already present there but one was missing. I played a little with it and added two sequences there. The sequences i contributed are **A156317** and **A156316** .

Have you ever come across a number sequence in your work (or play) – such as 1, 1, 2, 3, 6, 11, 23, 47, … and wanted to find out what was known about it (or even simply the next term)? This is the place to find out. I would recommend this site for all programming freaks and math maniacs.

-fR0D

Written by fR0DDY

February 9, 2009 at 8:56 AM

Posted in Maths

Tagged with OEIS

"Hello World!"

leave a comment »

Hi everyone,

I am Gaurav.I am a junior undergraduate at IT-BHU with keen interest in computers specially programming and maths.

Through this blog i wish to write mainly on technical stuffs but often my thoughts about life and other things may creep in.

So hang On! And Happy Coding!
-fR0D

Written by fR0DDY

February 9, 2009 at 8:20 AM

Posted in Uncategorized