COME ON CODE ON

A blog about programming and more programming.

Archive for the '**Maths**' Category

Number of Binary Trees

with 4 comments

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

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

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

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

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

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

are $t_{n-m,h-1}\sum_{i=0}^{h-2} t_{m-1,i}$.

Hence we get the recurrent relation

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

or

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

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

Here's a sample C++ code.

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

#define MAXN 35

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

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

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

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

Note :

1. The total number of binary trees with n nodes is $\frac{2n!}{n!(n+1)!}$, also known as Catalan numbers or Segner numbers.
2. $t_{n,n} = 2^{n-1}$.
3. The number of trees with height not lower than h is $\sum_{i=h}^{n} t_{n,i}$.
4. There are other recurrence relation as well such as
   $t_{n,h}=\sum_{i=0}^{h-1}\left(t_{n-1,h-1-i}\left(2*\sum_{j=0}^{n-2}t_{j,i}+t_{n-1,i}\right)\right)$.

NJOY!
-fR0DDY

Written by fR0DDY

April 13, 2010 at 11:44 AM

Posted in Algorithm, Maths, Programming

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

Convert a number from decimal base to any Base

with 6 comments

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

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

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

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

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

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

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

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

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


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

NJOY!
-fR0DDY

Written by fR0DDY

February 17, 2010 at 6:06 PM

Posted in Algorithm, Maths, Programming

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

Number of zeores and digits in N Factorial in Base B

with 2 comments

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

For example,
20! can be written as 2432902008176640000 in decimal number system while it is equal to "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

Programming Multiplicative Functions

with 9 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 …, then f(n) = f(p^a) f(q^b) …

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

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

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);
}
```

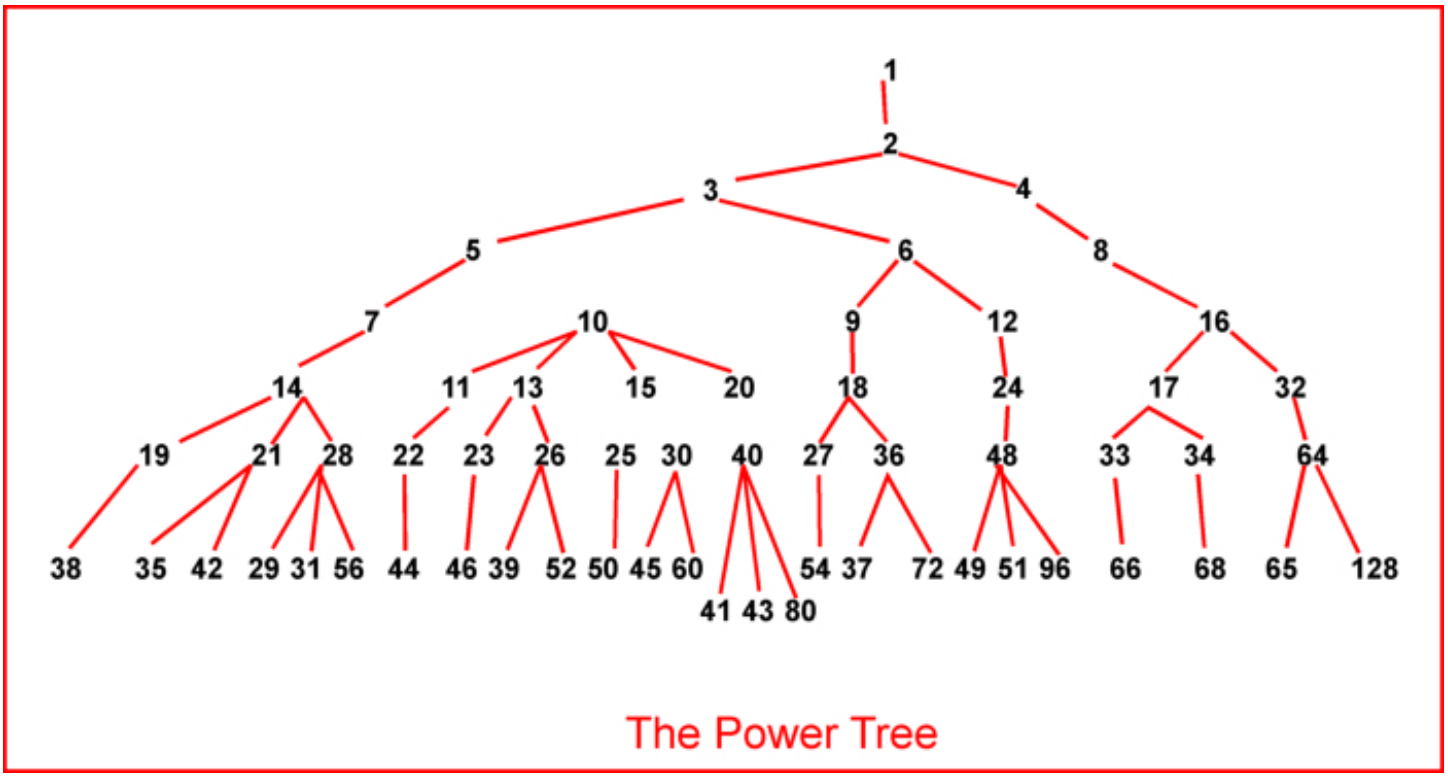
But does this method give the minimum number of multiplications. The answer is **NO**.
The smallest counterexample is n=15. The binary method takes six multiplications but the best method can do it in five multiplications. We can calculate y = x^3 in two multiplications and $x^{15} = y^5$ in three more, 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 = $x^5 = x^4x = (x^2)^2x$; then we form $y^{11} = y^{10}y = (y^2)^5y$. 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 $l(n) = \min(l(n-1)+1 , l_n) - \delta_n$, 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 $\delta_n = 1$ 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

COME ON CODE ON