|  |
| --- |
| import sys |
|  |  |
|  |  |
|  | palindromelist = [] |
|  | for i in range(100, 1000): |
|  | for j in range(100, 1000): |
|  | a = i \* j |
|  | if str(a) == str(a)[::-1] and a not in palindromelist: |
|  | palindromelist.append(a) |
|  | palindromelist.sort() |
|  | length = len(palindromelist) |
|  |  |
|  |  |
|  | if \_\_name\_\_ == '\_\_main\_\_': |
|  | n = int(input()) |
|  | for \_ in range(n): |
|  | a = int(input()) |
|  | for i in range(length - 1, -1, -1): |
|  | if palindromelist[i] < a: |
|  | print(palindromelist[i]) |
|  | break |

#4 A palindromic number reads the same both ways. The smallest 6 digit palindrome made from the product of two 3-digit numbers is .

#5

C++ solution

#include <cmath>

#include <cstdio>

#include <vector>

#include <iostream>

#include <algorithm>

using namespace std;

int main() {

/\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/

int T,N,i,j,k,q,M[40],p;

scanf("%d",&T);

M[0]=1;

M[1]=2;

M[2]=3;

for (i=3;i<40;i++){

q=i+1;

for (j=1;j<i;j++){

if (q%M[j]==0){

q/=M[j];

}

}

M[i]=q;

}

for (i=0;i<T;i++){

scanf("%d",&N);

p=1;

for (j=0;j<N;j++){

p\*=M[j];

}

printf("%d\n",p);

}

return 0;

}

#1 python

def sum(n, k):

    d = n // k

    return k \* (d \* (d+1)) // 2

def euler1(n):

    return sum(n, 3) + sum(n, 5) - sum(n, 15)

t = int(input().strip())

for i in range(t):

    N = int(input().strip())

    print(euler1(N - 1))

#2

def even\_fibonacci\_sum(n):

    fn\_2 = 1 #Fn-2

    fn\_1 = 1 #Fn-1

    sum = 0

    while True :

        fn = fn\_2 + fn\_1 #Fn

        if fn >= n: return sum

        if fn % 2 == 0: sum += fn

        fn\_2, fn\_1 = fn\_1, fn

t = int(input().strip())

for i in range(t):

    n = int(input().strip())

    print(even\_fibonacci\_sum(n))

#3

import math

def max\_prime\_factor(n):

    maxPrime = -1

    # We keep dividing n by 2 to get rid of all the even composite factors.

    while n % 2 == 0:

        maxPrime = 2

        n >>= 1 # equivalent to n //= 2

    # We loop over the possible odd factors.

    # to remove the rest of the composites, and updating maxPrime to the largest factor.

    for i in range(3, int(math.sqrt(n)) + 1, 2):

        while n % i == 0:

            maxPrime = i

            n //= i

    # If at this stage n is is still bigger than 2

    # then n must be prime so we return it, otherwise we return maxPrime

    return n if n > 2 else maxPrime

T = int(input().strip())

for \_ in range(T):

    N = int(input().strip())

    print(max\_prime\_factor(N))

#6 CPP

#include <iostream>

int main()

{

  unsigned int tests;

  std::cin >> tests;

  while (tests--)

  {

    unsigned long long x;

    std::cin >> x;

    unsigned long long sum        = 0; // 1   + 2   + ...

    unsigned long long sumSquared = 0; // 1^2 + 2^2 + ...

    for (unsigned long long i = 1; i <= x; i++)

    {

      sum        += i;

      sumSquared += i\*i;

    }

    // chances are that your compiler (partially) unrolls this simple loop

    // actually we don't need a loop for the sum (and the sum of squares)

    // => see "Alternative" section above

    // we had (1+2+...) instead of (1+2+...)^2

    unsigned long long squaredSum = sum \* sum;

    std::cout << (squaredSum - sumSquared) << std::endl;

  }

  return 0;

}

#7 CPP

#include <iostream>

#include <vector>

int main()

{

  // compute the first 10001 primes

  std::vector<unsigned int> primes;

  primes.reserve(10001);

  primes.push\_back(2);

  for (unsigned int x = 3; primes.size() <= 10000; x += 2)

  {

    bool isPrime = true;

    for (auto p : primes)

    {

      // found a divisor ? => abort

      if (x % p == 0)

      {

        isPrime = false;

        break;

      }

      // no larger prime factors possible ?

      if (p\*p > x)

        break;

    }

    // yes, we have a new prime

    if (isPrime)

      primes.push\_back(x);

  }

  // processing all test cases is now just a plain lookup

  unsigned int tests;

  std::cin >> tests;

  while (tests--)

  {

      unsigned int x;

      std::cin >> x;

      // just look up the x-th prime

      // with a little twist: vector's index is zero-based, therefore "off by one"

      x--;

      if (x < primes.size())

        std::cout << primes[x] << std::endl;

      else

        std::cout << "ERROR" << std::endl;

  }

  return 0;

}

#8 CPP

#include <cmath>

#include <cstdio>

#include <vector>

#include <iostream>

#include <algorithm>

using namespace std;

#define ctoi(c)(int)((c)-'0')

inline unsigned long long prodK(string number, int start, int end){

unsigned long long product=1;

for (int j=start;j<end;j++){

product\*=ctoi(number[j]);

}

return product;

}

int main() {

int T,N,K;

string number;

unsigned long long product,max;

cin>>T;

for (int i=0;i<T;i++){

cin>>N>>K;

cin>>number;

product=1;

product=prodK(number,0,K);

max=product;

for (int l=K;l<N;l++){

if (ctoi(number[l-K])!=0){

product=(product/ctoi(number[l-K]))\*ctoi(number[l]);

}else{

product=prodK(number,l-K+1,l+1);

}

max=(product>max)?product:max;

}

cout<<max<<"\n";

}

return 0;

}

#10 cpp

#include <iostream>

#include <vector>

using namespace std;

const int N=1000000;

bool mark[N+1000];

int main()

{

vector<int> primes;

vector<long long> sum;

for(int i=2;i<=N/2;++i)

if(mark[i]==false)

{

int b=i+i;

while(b<=N)

{

mark[b]=true;

b+=i;

}

}

primes.push\_back(2);

sum.push\_back(2);

for(int i=3;i<=N;++i)

if(mark[i]==false)

{

primes.push\_back(i);

sum.push\_back( sum[sum.size()-1]+i);

}

int t;

cin>>t;

primes.push\_back(10000000);

for(int k=1;k<=t;++k)

{

int n;

cin>>n;

int i;

for(i=0;primes[i]<=n;++i);

cout<<sum[i-1]<<endl;

}

}

#11 cpp

#include <iostream>

int main()

{

  // always a 20x20 matrix

  const unsigned int Size = 20;

  unsigned int matrix[Size][Size];

  // read from console

  for (unsigned int y = 0; y < Size; y++)

    for (unsigned int x = 0; x < Size; x++)

      std::cin >> matrix[x][y];

  unsigned int best = 0;

  // walk through all cells of the matrix

  for (unsigned int y = 0; y < Size; y++)

    for (unsigned int x = 0; x < Size; x++)

    {

      // three more horizontal cells (right)

      if (x + 3 < Size)

      {

        unsigned int current = matrix[x][y] \* matrix[x+1][y] \* matrix[x+2][y] \* matrix[x+3][y];

        if (best < current)

          best = current;

      }

      // three more vertical cells available (down)

      if (y + 3 < Size)

      {

        unsigned int current = matrix[x][y] \* matrix[x][y+1] \* matrix[x][y+2] \* matrix[x][y+3];

        if (best < current)

          best = current;

      }

      // three more diagonal cells (right-down)

      if (x + 3 < Size && y + 3 < Size)

      {

        unsigned int current = matrix[x][y] \* matrix[x+1][y+1] \* matrix[x+2][y+2] \* matrix[x+3][y+3];

        if (best < current)

          best = current;

      }

      // three more diagonal cells (left-down)

      if (x + 3 < Size && y >= 3)

      {

        unsigned int current = matrix[x][y] \* matrix[x+1][y-1] \* matrix[x+2][y-2] \* matrix[x+3][y-3];

        if (best < current)

          best = current;

      }

    }

  std::cout << best << std::endl;

  return 0;

}

#12 cpp

#include <iostream>

#include <vector>

int main()

{

  // find the smallest number with at least 1000 divisors

  // (due to Hackerrank's input range)

  const unsigned int MaxDivisors = 1000;

  // store [divisors] => [smallest number]

  std::vector<unsigned int> smallest;

  smallest.push\_back(0); // 0 => no divisors

  // for index=1 we have triangle=1

  // for index=2 we have triangle=3

  // for index=3 we have triangle=6

  // ...

  // for index=7 we have triangle=28

  // ...

  unsigned int index    = 0;

  unsigned int triangle = 0; // same as index\*(index+1)/2

  while (smallest.size() < MaxDivisors)

  {

    // next triangle number

    index++;

    triangle += index;

    // performance tweak (5x faster):

    // I observed that the "best" numbers with more than 300 divisors end with a zero

    // that's something I cannot prove right now, I just "saw" that debugging my code

    if (smallest.size() > 300 && triangle % 10 != 0)

      continue;

    // find all divisors i where i\*j=triangle

    // it's much faster to assume i < j, which means i\*i < triangle

    // whenever we find i then there is a j, too

    unsigned int divisors = 0;

    unsigned int i        = 1;

    while (i\*i < triangle)

    {

      // divisible ? yes, we found i and j, that's two divisors

      if (triangle % i == 0)

        divisors += 2;

      i++;

    }

    // if i=j then i^2=triangle and we have another divisor

    if (i\*i == triangle)

      divisors++;

    // fill gaps:

    // e.g. 10 is the smallest number with 4 divisors

    //      28 is the smallest number with 6 divisors

    // there is no number between 10 and 28 with 5 divisors

    // therefore 28 is the smallest number with AT LEAST 5 divisors, too

    while (smallest.size() <= divisors)

      smallest.push\_back(triangle);

  }

  unsigned int tests;

  std::cin >> tests;

  while (tests--)

  {

    unsigned int minDivisors;

    std::cin >> minDivisors;

    // problem setting asks for "over" x divisors => "plus one"

    std::cout << smallest[minDivisors + 1] << std::endl;

  }

  return 0;

}

#13 cpp

include <string>

#include <vector>

#include <iostream>

int main()

{

  // store each digit separately

  // input has 50 digits

  // highest digits might overflow and require a few extra digits

  // (I believe +2 would suffice, too)

  const unsigned int MinDigits = 50 + 10;

  // all digits are initially zero, least significant has index 0

  std::vector<unsigned int> sum(MinDigits, 0);

  // the resulting number will be sum[0] + 10\*sum[1] + 100\*sum[2] + ...

  unsigned int numbers = 100;

//#define ORIGINAL

#ifndef ORIGINAL

  std::cin >> numbers;

#endif

  while (numbers--)

  {

    // read a single number as a string

    std::string strAdd;

    std::cin >> strAdd;

    // convert to digits

    std::vector<unsigned int> add;

    // process string in reverse: least significant digits first

    for (auto i = strAdd.rbegin(); i != strAdd.rend(); i++)

      add.push\_back(\*i - '0'); // convert from ASCII

    // fill high/unused positions with zeros

    add.resize(sum.size(), 0);

    // add all digits

    for (unsigned int i = 0; i < add.size(); i++)

    {

      sum[i] += add[i];

      // overflow ? => sum[i] is 10 .. 18

      if (sum[i] >= 10)

      {

        sum[i + 1]++; // sum[i + 1] = sum[i] % 10

        sum[i] -= 10; // sum[i]    %= 10

      }

    }

  }

  // skip high zeros

  auto i = sum.rbegin();

  while (\*i == 0)

    i++;

  // print first ten digits

  unsigned int numDigits = 10;

  while (numDigits-- > 0)

    std::cout << \*i++;

  return 0;

}