_____ / _____ / _____

Otimização de algoritmos em C++

Seja o problema 12 do magnífico site de matemática e programação `projecteuler.net`. Ele diz: A sequência dos números triangulares é gerada pela adição dos números naturais. Assim, o 7^o número triangular será $1+2+3+4+5+6+7 = 28$. Os primeiros 10 números triangulares são

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...$$

Vai-se listar aqui os divisores dos primeiros sete números triangulares:

```
1:  1
3:  1,3
6:  1,2,3,6
10: 1,2,5,10
15: 1,3,5,15
21: 1,3,7,21
28: 1,2,4,7,14,28
```

Pode-se ver que 28 é o primeiro número triangular que tem mais do que 5 divisores. Qual é o valor do primeiro número triangular que tem mais do que 500 divisores ?

Primeira estratégia

Sem pensar muito, pode-se resolver a questão usando a força bruta. Aqui, geram-se os números triangulares como descrito e depois acham-se todos seus divisores. A busca acaba quando se acha o primeiro com mais do que 500 divisores. (Uma informação: inteiros de 32 bits são suficientes para este problema). Uma solução em C++:

```
#include<iostream>
using namespace std;
int main() {
   int t,a,cnt,i;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      while (i<=t){
         if ((t%i)==0){
            cnt++;
         }
         i++;
      }
   }
   cout<<t;
}
```

A implementação acima é muito demorada (17.75 minutos em uma CPU média, com 2 processadores).

Segunda estratégia

Uma melhora evidente é interromper a busca quando o divisor alcança a raiz quadrada do número triangular: Para cada divisor abaixo da raiz quadrada existe um acima dela. O código fica:

```
#include<iostream>
#include<cmath>
using namespace std;
int main() {
   int t,a,cnt,i,ttx;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      ttx=sqrt(t);
      while (i<=ttx){
         if ((t%i)==0){
            cnt=cnt+2;
         }
         if (t==(ttx*ttx)){
            cnt=cnt-1; // correcao quadrado perfeito
         }
         i++;
      }
      }
      cout<<t;
}
```

Ainda assim, esta segunda estratégia continua demorada.

Terceira estratégia

Vamos nos socorrer de um pouco de matemática. Sabe-se que qualquer inteiro N pode ser expresso de maneira única como

$$N = p_1^{a1} \times p_2^{a2} \times p_3^{a3} \times ...$$

onde p_i é um número primo e a_n seu expoente. Por exemplo, $28 = 2^2 \times 7^1$. Daqui, o número de divisores de N, $D(N)$ de qualquer inteiro pode ser computado como

$$D(N) = (a_1 + 1) \times (a_2 + 1) \times ...$$

onde os a_n são os expoentes dos fatores primos que compõe o número N. No exemplo, os divisores de 28 são $D(28) = (2 + 1) \times (1 + 1) = 3 \times 2 = 6$. Uma tabela de primos é necessária para aplicar esta relação. Uma estratégia para criá-la pode ser (até 65500):

```
#include<iostream>
#include<cmath>
#include<fstream>
#include <stdlib.h>
using namespace std;
int pri[7000];
int primo(int n){
   int f,i,r,j;
   if (n==1){
      return 0;
   }
   if (n<4){
      return 1;
   }
   if (0==(n%2)){
      return 0;
   }
   if (n<9){
      return 1;
   }
   if (0==(n%3)){
      return 0;
   }
   r=floor(sqrt(n));
   f=5;
   while (f<=r){
      if ((n%f)==0){
         return 0;
      }
      if ((n%(f+2))==0){
         return 0;
      }
      f=f+6;
   }
   return 1;
}
int criatabprimos(){
   int i,j;
   pri[0]=2;
   i=1;
   j=3;
   while (j<65500){
      if (primo(j)==1){
         pri[i]=j;
         i=i+1;
      }
      j=j+2;
   }
   ofstream outfile;
   outfile.open("c:/p/n/162/primos.dat");
   j=0;
   while(j<i){
      outfile<<pri[j]<<endl;
      j=j+1;
   }
   outfile.close();
}
int main() {
   int t,a,cnt,tt,i,expoente;
   string valor;
//   criatabprimos();
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (!infile.eof()){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   t=1;
   a=1;
   cnt=1;
   while (cnt<=500){
      cnt=1;
      a=a+1;
      t=t+a;
      tt=t;
      i=0;
      while (i<6540){
          if (pri[i]*pri[i]>tt){
             cnt=cnt*2;
             break;
          }
          expoente=1;
          while ((tt%pri[i])==0){
             expoente=expoente+1;
             tt=tt/pri[i];
          }
          if (expoente>1){
             cnt=cnt*expoente;
             }
          if (tt==1){
             break;
             }
          i=i+1;
      }
   }
   cout<<t;
}
```

Quarta estratégia

Ainda se pode melhorar mais, sabendo que os números triangulares também podem ser obtidos de acordo com

$$t = \frac{n \times (n + 1)}{2}$$

onde os componentes n e $n+1$ são necessariamente co-primos (isto é, não têm qualquer fator primo comum nem divisor comum). Aqui, o número de divisores $D(t)$ pode ser obtido

$$D(t) = D(n/2) \times D(n + 1) \; se \; n \; par$$

ou

$$D(t) = D(n) \times D((n + 1)/2) \; se \; n + 1 \; par$$

Cada componente é muito menor do que o número triangular. A tabela de primos também é muito menor (contém apenas primos menores do que 1000). Mais, o resultado do componente $n + 1$ pode ser reaproveitado como n no próximo número triangular, sem ser necessário recalculá-lo. Fica:

```
#include<iostream>
#include<stdlib.h>
#include<fstream>
using namespace std;
int pri[168];
int main(){
   int i,n,Dn,xnt,Dn1,n1,expoente,cnt;
   string valor;
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (i<169){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   n=3;
   Dn=2;
   cnt=0;
   while (cnt<=500){
      n=n+1;
      n1=n;
      if ((n1%2)==0) {
          n1=n1/2;
      }
      Dn1=1;
      i=0;
      while (i<169){
         if ((pri[i]*pri[i])>n1){
            Dn1=2*Dn1;
            break;
            }
         expoente=1;
         while ((n1%pri[i])==0){
            expoente=expoente+1;
            n1=n1/pri[i];
         }
         if (expoente>1){
            Dn1=Dn1*expoente;
         }
         if (n1==1){
            break;
         }
         i++;
      }
      cnt=Dn*Dn1;
      Dn=Dn1;
   }
   cout<<((n*(n-1))/2)<<endl;
}
```

Esta implementação demorou perto de $\frac{1}{10}$ de segundo na mesma CPU lá do começo. (0.152 seg para ser exato). Veja-se como vale a pena estudar matemática e algoritmos.

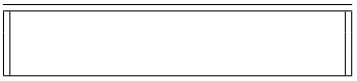
☞ Para você fazer

O objetivo deste exercício é treinar a implementação de algoritmos em C++ e também meditar sobre como modificações na implementação de um mesmo problema a partir de conhecimentos matemáticos (Santa Matemática !) podem reduzir radicalmente os tempos de processamento.

Para ajudar nesta meditação, implemente a alternativa que quiser e informe qual o número inteiro triangular que tem mais do que

```
121
```

divisores. Informe o número achado abaixo:



Otimização de algoritmos em C++

Seja o problema 12 do magnífico site de matemática e programação `projecteuler.net`. Ele diz: A sequência dos números triangulares é gerada pela adição dos números naturais. Assim, o 7^o número triangular será $1+2+3+4+5+6+7 = 28$. Os primeiros 10 números triangulares são

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...$$

Vai-se listar aqui os divisores dos primeiros sete números triangulares:

```
1:  1
3:  1,3
6:  1,2,3,6
10: 1,2,5,10
15: 1,3,5,15
21: 1,3,7,21
28: 1,2,4,7,14,28
```

Pode-se ver que 28 é o primeiro número triangular que tem mais do que 5 divisores. Qual é o valor do primeiro número triangular que tem mais do que 500 divisores ?

Primeira estratégia

Sem pensar muito, pode-se resolver a questão usando a força bruta. Aqui, geram-se os números triangulares como descrito e depois acham-se todos seus divisores. A busca acaba quando se acha o primeiro com mais do que 500 divisores. (Uma informação: inteiros de 32 bits são suficientes para este problema). Uma solução em C++:

```
#include<iostream>
using namespace std;
int main() {
   int t,a,cnt,i;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      while (i<=t){
         if ((t%i)==0){
            cnt++;
         }
         i++;
      }
   }
   cout<<t;
}
```

A implementação acima é muito demorada (17.75 minutos em uma CPU média, com 2 processadores).

Segunda estratégia

Uma melhora evidente é interromper a busca quando o divisor alcança a raiz quadrada do número triangular: Para cada divisor abaixo da raiz quadrada existe um acima dela. O código fica:

```
#include<iostream>
#include<cmath>
using namespace std;
int main() {
   int t,a,cnt,i,ttx;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      ttx=sqrt(t);
      while (i<=ttx){
         if ((t%i)==0){
            cnt=cnt+2;
         }
         if (t==(ttx*ttx)){
            cnt=cnt-1; // correcao quadrado perfeito
         }
         i++;
      }
      }
      cout<<t;
}
```

Ainda assim, esta segunda estratégia continua demorada.

Terceira estratégia

Vamos nos socorrer de um pouco de matemática. Sabe-se que qualquer inteiro N pode ser expresso de maneira única como

$$N = p_1^{a1} \times p_2^{a2} \times p_3^{a3} \times ...$$

onde p_i é um número primo e a_n seu expoente. Por exemplo, $28 = 2^2 \times 7^1$. Daqui, o número de divisores de N, $D(N)$ de qualquer inteiro pode ser computado como

$$D(N) = (a_1 + 1) \times (a_2 + 1) \times ...$$

onde os a_n são os expoentes dos fatores primos que compõe o número N. No exemplo, os divisores de 28 são $D(28) = (2 + 1) \times (1 + 1) = 3 \times 2 = 6$. Uma tabela de primos é necessária para aplicar esta relação. Uma estratégia para criá-la pode ser (até 65500):

```
#include<iostream>
#include<cmath>
#include<fstream>
#include <stdlib.h>
using namespace std;
int pri[7000];
int primo(int n){
   int f,i,r,j;
   if (n==1){
      return 0;
   }
   if (n<4){
      return 1;
   }
   if (0==(n%2)){
      return 0;
   }
   if (n<9){
      return 1;
   }
   if (0==(n%3)){
      return 0;
   }
   r=floor(sqrt(n));
   f=5;
   while (f<=r){
      if ((n%f)==0){
         return 0;
      }
      if ((n%(f+2))==0){
         return 0;
      }
      f=f+6;
   }
   return 1;
}
int criatabprimos(){
   int i,j;
   pri[0]=2;
   i=1;
   j=3;
   while (j<65500){
      if (primo(j)==1){
         pri[i]=j;
         i=i+1;
      }
      j=j+2;
   }
   ofstream outfile;
   outfile.open("c:/p/n/162/primos.dat");
   j=0;
   while(j<i){
      outfile<<pri[j]<<endl;
      j=j+1;
   }
   outfile.close();
}
int main() {
   int t,a,cnt,tt,i,expoente;
   string valor;
//   criatabprimos();
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (!infile.eof()){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   t=1;
   a=1;
   cnt=1;
   while (cnt<=500){
      cnt=1;
      a=a+1;
      t=t+a;
      tt=t;
      i=0;
      while (i<6540){
          if (pri[i]*pri[i]>tt){
             cnt=cnt*2;
             break;
          }
          expoente=1;
          while ((tt%pri[i])==0){
             expoente=expoente+1;
             tt=tt/pri[i];
          }
          if (expoente>1){
             cnt=cnt*expoente;
             }
          if (tt==1){
             break;
             }
          i=i+1;
      }
   }
   cout<<t;
}
```

Quarta estratégia

Ainda se pode melhorar mais, sabendo que os números triangulares também podem ser obtidos de acordo com

$$t = \frac{n \times (n + 1)}{2}$$

onde os componentes n e $n+1$ são necessariamente co-primos (isto é, não têm qualquer fator primo comum nem divisor comum). Aqui, o número de divisores $D(t)$ pode ser obtido

$$D(t) = D(n/2) \times D(n + 1) \; se \; n \; par$$

ou

$$D(t) = D(n) \times D((n + 1)/2) \; se \; n + 1 \; par$$

Cada componente é muito menor do que o número triangular. A tabela de primos também é muito menor (contém apenas primos menores do que 1000). Mais, o resultado do componente $n + 1$ pode ser reaproveitado como n no próximo número triangular, sem ser necessário recalculá-lo. Fica:

```
#include<iostream>
#include<stdlib.h>
#include<fstream>
using namespace std;
int pri[168];
int main(){
   int i,n,Dn,xnt,Dn1,n1,expoente,cnt;
   string valor;
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (i<169){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   n=3;
   Dn=2;
   cnt=0;
   while (cnt<=500){
      n=n+1;
      n1=n;
      if ((n1%2)==0) {
          n1=n1/2;
      }
      Dn1=1;
      i=0;
      while (i<169){
         if ((pri[i]*pri[i])>n1){
            Dn1=2*Dn1;
            break;
            }
         expoente=1;
         while ((n1%pri[i])==0){
            expoente=expoente+1;
            n1=n1/pri[i];
         }
         if (expoente>1){
            Dn1=Dn1*expoente;
         }
         if (n1==1){
            break;
         }
         i++;
      }
      cnt=Dn*Dn1;
      Dn=Dn1;
   }
   cout<<((n*(n-1))/2)<<endl;
}
```

Esta implementação demorou perto de $\frac{1}{10}$ de segundo na mesma CPU lá do começo. (0.152 seg para ser exato). Veja-se como vale a pena estudar matemática e algoritmos.

☞ Para você fazer

O objetivo deste exercício é treinar a implementação de algoritmos em C++ e também meditar sobre como modificações na implementação de um mesmo problema a partir de conhecimentos matemáticos (Santa Matemática !) podem reduzir radicalmente os tempos de processamento.

Para ajudar nesta meditação, implemente a alternativa que quiser e informe qual o número inteiro triangular que tem mais do que

490

divisores. Informe o número achado abaixo:

_____ / _____ / _____

Otimização de algoritmos em C++

Seja o problema 12 do magnífico site de matemática e programação `projecteuler.net`. Ele diz: A sequência dos números triangulares é gerada pela adição dos números naturais. Assim, o 7^o número triangular será $1+2+3+4+5+6+7 = 28$. Os primeiros 10 números triangulares são

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...$$

Vai-se listar aqui os divisores dos primeiros sete números triangulares:

```
1:  1
3:  1,3
6:  1,2,3,6
10: 1,2,5,10
15: 1,3,5,15
21: 1,3,7,21
28: 1,2,4,7,14,28
```

Pode-se ver que 28 é o primeiro número triangular que tem mais do que 5 divisores. Qual é o valor do primeiro número triangular que tem mais do que 500 divisores ?

Primeira estratégia

Sem pensar muito, pode-se resolver a questão usando a força bruta. Aqui, geram-se os números triangulares como descrito e depois acham-se todos seus divisores. A busca acaba quando se acha o primeiro com mais do que 500 divisores. (Uma informação: inteiros de 32 bits são suficientes para este problema). Uma solução em C++:

```
#include<iostream>
using namespace std;
int main() {
   int t,a,cnt,i;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      while (i<=t){
         if ((t%i)==0){
            cnt++;
         }
         i++;
      }
   }
   cout<<t;
}
```

A implementação acima é muito demorada (17.75 minutos em uma CPU média, com 2 processadores).

Segunda estratégia

Uma melhora evidente é interromper a busca quando o divisor alcança a raiz quadrada do número triangular: Para cada divisor abaixo da raiz quadrada existe um acima dela. O código fica:

```
#include<iostream>
#include<cmath>
using namespace std;
int main() {
   int t,a,cnt,i,ttx;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      ttx=sqrt(t);
      while (i<=ttx){
         if ((t%i)==0){
            cnt=cnt+2;
         }
         if (t==(ttx*ttx)){
            cnt=cnt-1; // correcao quadrado perfeito
         }
         i++;
      }
      }
      cout<<t;
}
```

Ainda assim, esta segunda estratégia continua demorada.

Terceira estratégia

Vamos nos socorrer de um pouco de matemática. Sabe-se que qualquer inteiro N pode ser expresso de maneira única como

$$N = p_1^{a1} \times p_2^{a2} \times p_3^{a3} \times ...$$

onde p_i é um número primo e a_n seu expoente. Por exemplo, $28 = 2^2 \times 7^1$. Daqui, o número de divisores de N, $D(N)$ de qualquer inteiro pode ser computado como

$$D(N) = (a_1 + 1) \times (a_2 + 1) \times ...$$

onde os a_n são os expoentes dos fatores primos que compõe o número N. No exemplo, os divisores de 28 são $D(28) = (2 + 1) \times (1 + 1) = 3 \times 2 = 6$. Uma tabela de primos é necessária para aplicar esta relação. Uma estratégia para criá-la pode ser (até 65500):

```
#include<iostream>
#include<cmath>
#include<fstream>
#include <stdlib.h>
using namespace std;
int pri[7000];
int primo(int n){
   int f,i,r,j;
   if (n==1){
      return 0;
   }
   if (n<4){
      return 1;
   }
   if (0==(n%2)){
      return 0;
   }
   if (n<9){
      return 1;
   }
   if (0==(n%3)){
      return 0;
   }
   r=floor(sqrt(n));
   f=5;
   while (f<=r){
      if ((n%f)==0){
         return 0;
      }
      if ((n%(f+2))==0){
         return 0;
      }
      f=f+6;
   }
   return 1;
}
int criatabprimos(){
   int i,j;
   pri[0]=2;
   i=1;
   j=3;
   while (j<65500){
      if (primo(j)==1){
         pri[i]=j;
         i=i+1;
      }
      j=j+2;
   }
   ofstream outfile;
   outfile.open("c:/p/n/162/primos.dat");
   j=0;
   while(j<i){
      outfile<<pri[j]<<endl;
      j=j+1;
   }
   outfile.close();
}
int main() {
   int t,a,cnt,tt,i,expoente;
   string valor;
//   criatabprimos();
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (!infile.eof()){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   t=1;
   a=1;
   cnt=1;
   while (cnt<=500){
      cnt=1;
      a=a+1;
      t=t+a;
      tt=t;
      i=0;
      while (i<6540){
          if (pri[i]*pri[i]>tt){
             cnt=cnt*2;
             break;
          }
          expoente=1;
          while ((tt%pri[i])==0){
             expoente=expoente+1;
             tt=tt/pri[i];
          }
          if (expoente>1){
             cnt=cnt*expoente;
             }
          if (tt==1){
             break;
             }
          i=i+1;
      }
   }
   cout<<t;
}
```

Quarta estratégia

Ainda se pode melhorar mais, sabendo que os números triangulares também podem ser obtidos de acordo com

$$t = \frac{n \times (n + 1)}{2}$$

onde os componentes n e $n+1$ são necessariamente co-primos (isto é, não têm qualquer fator primo comum nem divisor comum). Aqui, o número de divisores $D(t)$ pode ser obtido

$$D(t) = D(n/2) \times D(n + 1) \; se \; n \; par$$

ou

$$D(t) = D(n) \times D((n + 1)/2) \; se \; n + 1 \; par$$

Cada componente é muito menor do que o número triangular. A tabela de primos também é muito menor (contém apenas primos menores do que 1000). Mais, o resultado do componente $n + 1$ pode ser reaproveitado como n no próximo número triangular, sem ser necessário recalculá-lo. Fica:

```
#include<iostream>
#include<stdlib.h>
#include<fstream>
using namespace std;
int pri[168];
int main(){
   int i,n,Dn,xnt,Dn1,n1,expoente,cnt;
   string valor;
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (i<169){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   n=3;
   Dn=2;
   cnt=0;
   while (cnt<=500){
      n=n+1;
      n1=n;
      if ((n1%2)==0) {
          n1=n1/2;
      }
      Dn1=1;
      i=0;
      while (i<169){
         if ((pri[i]*pri[i])>n1){
            Dn1=2*Dn1;
            break;
            }
         expoente=1;
         while ((n1%pri[i])==0){
            expoente=expoente+1;
            n1=n1/pri[i];
         }
         if (expoente>1){
            Dn1=Dn1*expoente;
         }
         if (n1==1){
            break;
         }
         i++;
      }
      cnt=Dn*Dn1;
      Dn=Dn1;
   }
   cout<<((n*(n-1))/2)<<endl;
}
```

Esta implementação demorou perto de $\frac{1}{10}$ de segundo na mesma CPU lá do começo. (0.152 seg para ser exato). Veja-se como vale a pena estudar matemática e algoritmos.

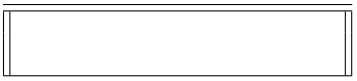
☞ Para você fazer

O objetivo deste exercício é treinar a implementação de algoritmos em C++ e também meditar sobre como modificações na implementação de um mesmo problema a partir de conhecimentos matemáticos (Santa Matemática !) podem reduzir radicalmente os tempos de processamento.

Para ajudar nesta meditação, implemente a alternativa que quiser e informe qual o número inteiro triangular que tem mais do que

156

divisores. Informe o número achado abaixo:



_____ / _____ / _____

Otimização de algoritmos em C++

Seja o problema 12 do magnífico site de matemática e programação `projecteuler.net`. Ele diz: A sequência dos números triangulares é gerada pela adição dos números naturais. Assim, o 7^o número triangular será $1+2+3+4+5+6+7 = 28$. Os primeiros 10 números triangulares são

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...$$

Vai-se listar aqui os divisores dos primeiros sete números triangulares:

```
1:  1
3:  1,3
6:  1,2,3,6
10: 1,2,5,10
15: 1,3,5,15
21: 1,3,7,21
28: 1,2,4,7,14,28
```

Pode-se ver que 28 é o primeiro número triangular que tem mais do que 5 divisores. Qual é o valor do primeiro número triangular que tem mais do que 500 divisores ?

Primeira estratégia

Sem pensar muito, pode-se resolver a questão usando a força bruta. Aqui, geram-se os números triangulares como descrito e depois acham-se todos seus divisores. A busca acaba quando se acha o primeiro com mais do que 500 divisores. (Uma informação: inteiros de 32 bits são suficientes para este problema). Uma solução em C++:

```
#include<iostream>
using namespace std;
int main() {
   int t,a,cnt,i;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      while (i<=t){
         if ((t%i)==0){
            cnt++;
         }
         i++;
      }
   }
   cout<<t;
}
```

A implementação acima é muito demorada (17.75 minutos em uma CPU média, com 2 processadores).

Segunda estratégia

Uma melhora evidente é interromper a busca quando o divisor alcança a raiz quadrada do número triangular: Para cada divisor abaixo da raiz quadrada existe um acima dela. O código fica:

```
#include<iostream>
#include<cmath>
using namespace std;
int main() {
   int t,a,cnt,i,ttx;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      ttx=sqrt(t);
      while (i<=ttx){
         if ((t%i)==0){
            cnt=cnt+2;
         }
         if (t==(ttx*ttx)){
            cnt=cnt-1; // correcao quadrado perfeito
         }
         i++;
      }
      }
      cout<<t;
}
```

Ainda assim, esta segunda estratégia continua demorada.

Terceira estratégia

Vamos nos socorrer de um pouco de matemática. Sabe-se que qualquer inteiro N pode ser expresso de maneira única como

$$N = p_1^{a1} \times p_2^{a2} \times p_3^{a3} \times ...$$

onde p_i é um número primo e a_n seu expoente. Por exemplo, $28 = 2^2 \times 7^1$. Daqui, o número de divisores de N, $D(N)$ de qualquer inteiro pode ser computado como

$$D(N) = (a_1 + 1) \times (a_2 + 1) \times ...$$

onde os a_n são os expoentes dos fatores primos que compõe o número N. No exemplo, os divisores de 28 são $D(28) = (2 + 1) \times (1 + 1) = 3 \times 2 = 6$. Uma tabela de primos é necessária para aplicar esta relação. Uma estratégia para criá-la pode ser (até 65500):

```
#include<iostream>
#include<cmath>
#include<fstream>
#include <stdlib.h>
using namespace std;
int pri[7000];
int primo(int n){
   int f,i,r,j;
   if (n==1){
      return 0;
   }
   if (n<4){
      return 1;
   }
   if (0==(n%2)){
      return 0;
   }
   if (n<9){
      return 1;
   }
   if (0==(n%3)){
      return 0;
   }
   r=floor(sqrt(n));
   f=5;
   while (f<=r){
      if ((n%f)==0){
         return 0;
      }
      if ((n%(f+2))==0){
         return 0;
      }
      f=f+6;
   }
   return 1;
}
int criatabprimos(){
   int i,j;
   pri[0]=2;
   i=1;
   j=3;
   while (j<65500){
      if (primo(j)==1){
         pri[i]=j;
         i=i+1;
      }
      j=j+2;
   }
   ofstream outfile;
   outfile.open("c:/p/n/162/primos.dat");
   j=0;
   while(j<i){
      outfile<<pri[j]<<endl;
      j=j+1;
   }
   outfile.close();
}
int main() {
   int t,a,cnt,tt,i,expoente;
   string valor;
//   criatabprimos();
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (!infile.eof()){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   t=1;
   a=1;
   cnt=1;
   while (cnt<=500){
      cnt=1;
      a=a+1;
      t=t+a;
      tt=t;
      i=0;
      while (i<6540){
          if (pri[i]*pri[i]>tt){
             cnt=cnt*2;
             break;
          }
          expoente=1;
          while ((tt%pri[i])==0){
             expoente=expoente+1;
             tt=tt/pri[i];
          }
          if (expoente>1){
             cnt=cnt*expoente;
             }
          if (tt==1){
             break;
             }
          i=i+1;
      }
   }
   cout<<t;
}
```

Quarta estratégia

Ainda se pode melhorar mais, sabendo que os números triangulares também podem ser obtidos de acordo com

$$t = \frac{n \times (n + 1)}{2}$$

onde os componentes n e $n+1$ são necessariamente co-primos (isto é, não têm qualquer fator primo comum nem divisor comum). Aqui, o número de divisores $D(t)$ pode ser obtido

$$D(t) = D(n/2) \times D(n + 1) \; se \; n \; par$$

ou

$$D(t) = D(n) \times D((n + 1)/2) \; se \; n + 1 \; par$$

Cada componente é muito menor do que o número triangular. A tabela de primos também é muito menor (contém apenas primos menores do que 1000). Mais, o resultado do componente $n + 1$ pode ser reaproveitado como n no próximo número triangular, sem ser necessário recalculá-lo. Fica:

```
#include<iostream>
#include<stdlib.h>
#include<fstream>
using namespace std;
int pri[168];
int main(){
   int i,n,Dn,xnt,Dn1,n1,expoente,cnt;
   string valor;
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (i<169){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   n=3;
   Dn=2;
   cnt=0;
   while (cnt<=500){
      n=n+1;
      n1=n;
      if ((n1%2)==0) {
          n1=n1/2;
      }
      Dn1=1;
      i=0;
      while (i<169){
         if ((pri[i]*pri[i])>n1){
            Dn1=2*Dn1;
            break;
            }
         expoente=1;
         while ((n1%pri[i])==0){
            expoente=expoente+1;
            n1=n1/pri[i];
         }
         if (expoente>1){
            Dn1=Dn1*expoente;
         }
         if (n1==1){
            break;
         }
         i++;
      }
      cnt=Dn*Dn1;
      Dn=Dn1;
   }
   cout<<((n*(n-1))/2)<<endl;
}
```

Esta implementação demorou perto de $\frac{1}{10}$ de segundo na mesma CPU lá do começo. (0.152 seg para ser exato). Veja-se como vale a pena estudar matemática e algoritmos.

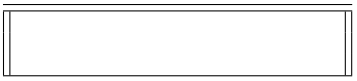
☞ Para você fazer

O objetivo deste exercício é treinar a implementação de algoritmos em C++ e também meditar sobre como modificações na implementação de um mesmo problema a partir de conhecimentos matemáticos (Santa Matemática !) podem reduzir radicalmente os tempos de processamento.

Para ajudar nesta meditação, implemente a alternativa que quiser e informe qual o número inteiro triangular que tem mais do que

354

divisores. Informe o número achado abaixo:



_____ / _____ / _____

Otimização de algoritmos em C++

Seja o problema 12 do magnífico site de matemática e programação `projecteuler.net`. Ele diz: A sequência dos números triangulares é gerada pela adição dos números naturais. Assim, o 7^o número triangular será $1+2+3+4+5+6+7 = 28$. Os primeiros 10 números triangulares são

$$1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...$$

Vai-se listar aqui os divisores dos primeiros sete números triangulares:

```
1:  1
3:  1,3
6:  1,2,3,6
10: 1,2,5,10
15: 1,3,5,15
21: 1,3,7,21
28: 1,2,4,7,14,28
```

Pode-se ver que 28 é o primeiro número triangular que tem mais do que 5 divisores. Qual é o valor do primeiro número triangular que tem mais do que 500 divisores ?

Primeira estratégia

Sem pensar muito, pode-se resolver a questão usando a força bruta. Aqui, geram-se os números triangulares como descrito e depois acham-se todos seus divisores. A busca acaba quando se acha o primeiro com mais do que 500 divisores. (Uma informação: inteiros de 32 bits são suficientes para este problema). Uma solução em C++:

```
#include<iostream>
using namespace std;
int main() {
   int t,a,cnt,i;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      while (i<=t){
         if ((t%i)==0){
            cnt++;
         }
         i++;
      }
   }
   cout<<t;
}
```

A implementação acima é muito demorada (17.75 minutos em uma CPU média, com 2 processadores).

Segunda estratégia

Uma melhora evidente é interromper a busca quando o divisor alcança a raiz quadrada do número triangular: Para cada divisor abaixo da raiz quadrada existe um acima dela. O código fica:

```
#include<iostream>
#include<cmath>
using namespace std;
int main() {
   int t,a,cnt,i,ttx;
   t=1;
   a=1;
   cnt=0;
   while (cnt<=500){
      cnt=0;
      a=a+1;
      t=t+a;
      i=1;
      ttx=sqrt(t);
      while (i<=ttx){
         if ((t%i)==0){
            cnt=cnt+2;
         }
         if (t==(ttx*ttx)){
            cnt=cnt-1; // correcao quadrado perfeito
         }
         i++;
      }
      }
      cout<<t;
}
```

Ainda assim, esta segunda estratégia continua demorada.

Terceira estratégia

Vamos nos socorrer de um pouco de matemática. Sabe-se que qualquer inteiro N pode ser expresso de maneira única como

$$N = p_1^{a1} \times p_2^{a2} \times p_3^{a3} \times ...$$

onde p_i é um número primo e a_n seu expoente. Por exemplo, $28 = 2^2 \times 7^1$. Daqui, o número de divisores de N, $D(N)$ de qualquer inteiro pode ser computado como

$$D(N) = (a_1 + 1) \times (a_2 + 1) \times ...$$

onde os a_n são os expoentes dos fatores primos que compõe o número N. No exemplo, os divisores de 28 são $D(28) = (2 + 1) \times (1 + 1) = 3 \times 2 = 6$. Uma tabela de primos é necessária para aplicar esta relação. Uma estratégia para criá-la pode ser (até 65500):

```
#include<iostream>
#include<cmath>
#include<fstream>
#include <stdlib.h>
using namespace std;
int pri[7000];
int primo(int n){
   int f,i,r,j;
   if (n==1){
      return 0;
   }
   if (n<4){
      return 1;
   }
   if (0==(n%2)){
      return 0;
   }
   if (n<9){
      return 1;
   }
   if (0==(n%3)){
      return 0;
   }
   r=floor(sqrt(n));
   f=5;
   while (f<=r){
      if ((n%f)==0){
         return 0;
      }
      if ((n%(f+2))==0){
         return 0;
      }
      f=f+6;
   }
   return 1;
}
int criatabprimos(){
   int i,j;
   pri[0]=2;
   i=1;
   j=3;
   while (j<65500){
      if (primo(j)==1){
         pri[i]=j;
         i=i+1;
      }
      j=j+2;
   }
   ofstream outfile;
   outfile.open("c:/p/n/162/primos.dat");
   j=0;
   while(j<i){
      outfile<<pri[j]<<endl;
      j=j+1;
   }
   outfile.close();
}
int main() {
   int t,a,cnt,tt,i,expoente;
   string valor;
//   criatabprimos();
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (!infile.eof()){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   t=1;
   a=1;
   cnt=1;
   while (cnt<=500){
      cnt=1;
      a=a+1;
      t=t+a;
      tt=t;
      i=0;
      while (i<6540){
          if (pri[i]*pri[i]>tt){
             cnt=cnt*2;
             break;
          }
          expoente=1;
          while ((tt%pri[i])==0){
             expoente=expoente+1;
             tt=tt/pri[i];
          }
          if (expoente>1){
             cnt=cnt*expoente;
             }
          if (tt==1){
             break;
             }
          i=i+1;
      }
   }
   cout<<t;
}
```

Quarta estratégia

Ainda se pode melhorar mais, sabendo que os números triangulares também podem ser obtidos de acordo com

$$t = \frac{n \times (n + 1)}{2}$$

onde os componentes n e $n+1$ são necessariamente co-primos (isto é, não têm qualquer fator primo comum nem divisor comum). Aqui, o número de divisores $D(t)$ pode ser obtido

$$D(t) = D(n/2) \times D(n + 1) \; se \; n \; par$$

ou

$$D(t) = D(n) \times D((n + 1)/2) \; se \; n + 1 \; par$$

Cada componente é muito menor do que o número triangular. A tabela de primos também é muito menor (contém apenas primos menores do que 1000). Mais, o resultado do componente $n + 1$ pode ser reaproveitado como n no próximo número triangular, sem ser necessário recalculá-lo. Fica:

```
#include<iostream>
#include<stdlib.h>
#include<fstream>
using namespace std;
int pri[168];
int main(){
   int i,n,Dn,xnt,Dn1,n1,expoente,cnt;
   string valor;
   ifstream infile;
   infile.open("c:/p/n/162/primos.dat");
   i=0;
   while (i<169){
      getline(infile,valor);
      pri[i]=atoi(valor.c_str());
      i=i+1;
      }
   infile.close();
   n=3;
   Dn=2;
   cnt=0;
   while (cnt<=500){
      n=n+1;
      n1=n;
      if ((n1%2)==0) {
          n1=n1/2;
      }
      Dn1=1;
      i=0;
      while (i<169){
         if ((pri[i]*pri[i])>n1){
            Dn1=2*Dn1;
            break;
            }
         expoente=1;
         while ((n1%pri[i])==0){
            expoente=expoente+1;
            n1=n1/pri[i];
         }
         if (expoente>1){
            Dn1=Dn1*expoente;
         }
         if (n1==1){
            break;
         }
         i++;
      }
      cnt=Dn*Dn1;
      Dn=Dn1;
   }
   cout<<((n*(n-1))/2)<<endl;
}
```

Esta implementação demorou perto de $\frac{1}{10}$ de segundo na mesma CPU lá do começo. (0.152 seg para ser exato). Veja-se como vale a pena estudar matemática e algoritmos.

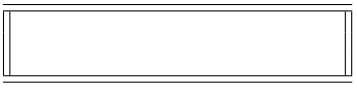
☞ Para você fazer

O objetivo deste exercício é treinar a implementação de algoritmos em C++ e também meditar sobre como modificações na implementação de um mesmo problema a partir de conhecimentos matemáticos (Santa Matemática !) podem reduzir radicalmente os tempos de processamento.

Para ajudar nesta meditação, implemente a alternativa que quiser e informe qual o número inteiro triangular que tem mais do que

313

divisores. Informe o número achado abaixo:



```
===== 31/05/2018 17:19:16.4 =====E=PL162c
    1              157080
    2            76576500
    3              749700
    4            17907120
    5             2162160
```