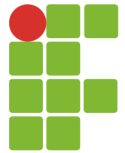
Funções

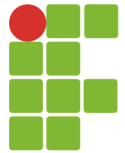
Algoritmos e Programação de Computadores – ABI/LFI/TAI

Prof. Daniel Saad Nogueira Nunes

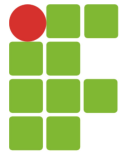
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

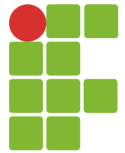


Sumário



Introdução

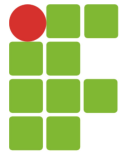
- Quando temos um problema muito complexo de ser resolvido, uma possível estratégia para atacá-lo é dividi-lo em partes menores, mais simples de serem resolvidas.
- Códigos grandes também são difíceis de serem mantidos. A leitura é dificultada e a eliminação de *bugs* é mais árdua.
- Modularização: separação do algoritmo em módulos para resolução de problemas.
- Não há como desenvolver sistemas complexos e em equipe se não utilizarmos este princípio.



Introdução: Funções

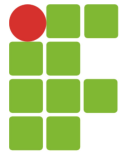
Funções

- Funções são mecanismos que agrupam um conjunto de instruções.
- Quando as funções são chamadas, ou invocadas, o conjunto de instruções subjacente é executado.



Introdução: Funções

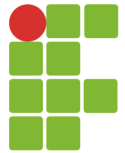
- Você já está acostumado com a utilização de funções.
- Por exemplo: `scanf` e `printf` são funções!
- Funções também podem retornar resultados. Utilizamos até o retorno de algumas funções, por exemplo a função `sqrt` é outra muito utilizada por vocês, dado um número `double`, ela devolve a raiz quadrada deste número.
  - Ex: `x = sqrt(y);`.



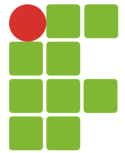
Introdução Funções

Por que usar funções?

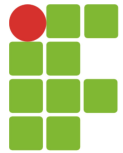
- Modularização: há uma divisão lógica no programa. As partes podem ser entendidas separadamente.
- Depuração: uma vez que temos várias partes comunicantes no programa, é mais fácil a detecção e correção de bugs.
- Legibilidade: os programas passam a ser melhor entendidos, visto que não consistem apenas de uma única estrutura monolítica.
- Reuso: é possível reusar os códigos presentes nas funções sem que seja necessário replicá-lo.
- Compartilhamento: através das funções podemo usar ou disponibilizar bibliotecas que facilitam o desenvolvimento de programas mais complexos.



Sumário



Sumário

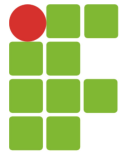


Funções: Sintaxe

Definição de funções

```
1  tipo nome_funcao(tipo1 parametro1, tipo2 parametro2, ... , tipoN parametroN){
2      comando_1;
3      comando_2;
4      ...
5      comando_M;
6      return valor_de_retorno;
7  }
```

- Toda função deve ter um tipo: `float`, `double`, `void`, `int`...

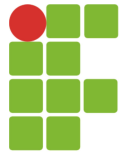


Funções: Sintaxe

Definição de funções

```
1  tipo nome_funcao(tipo1 parametro1, tipo2 parametro2, ... , tipoN parametroN){
2      comando_1;
3      comando_2;
4      ...
5      comando_M;
6      return valor_de_retorno;
7  }
```

- Os parâmetros são variáveis, pertencentes ao escopo da função, que são inicializados com os valores indicados na invocação da função.

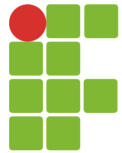


Funções: Sintaxe

Definição de funções

```
1  tipo nome_funcao(tipo1 parametro1, tipo2 parametro2, ... , tipoN parametroN){
2      comando_1;
3      comando_2;
4      ...
5      comando_M;
6      return valor_de_retorno;
7  }
```

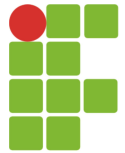
- O comando `return` devolve para o invocador da função o resultado da execução desta.



Funções: Exemplo

```
1  int soma(int a, int b) {
2      int c = a + b;
3      return c;
4  }
```

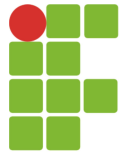
- O código acima define uma função `soma`, que recebe dois inteiros e retorna o resultado da soma destes inteiros.
- Note que a variável que está sendo retornada possui o mesmo tipo da função.
- Ao encontrar o comando `return`, a função finaliza a sua execução e retorna o valor para o invocador da função.



Funções: Exemplo

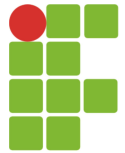
```c
#include <stdio.h>

int soma(int a, int b) {
    int c = a + b;
    return c;
}

int main(void) {
    int x = 5, y = 7;
    int r = soma(2, 3);
    printf("%d\n", r);
    r = soma(x, y);
    printf("%d\n", r);
    printf("%d\n", soma(10, -20));
    return 0;
}
```

O que será impresso?



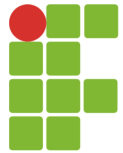
Funções: Exemplo

- Uma função não necessariamente precisa receber argumentos.
- Podemos ter uma função que apenas retorna o valor lido.
- O exemplo a seguir traz uma função que imprime na tela uma mensagem para o usuário solicitando a entrada de um inteiro, em seguida lê o inteiro e por fim, retorna o inteiro lido para o invocador da função.



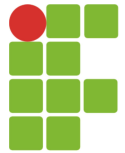
Funções: Exemplo

```
1  int le_inteiro(void){
2      int inteiro;
3      printf("Digite um inteiro: ");
4      scanf("%d",&inteiro);
5      return inteiro;
6  }
```

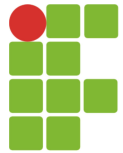


Funções Exemplo

```
1  #include <stdio.h>
2
3  int le_inteiro(void){
4      int inteiro;
5      printf("Digite um inteiro: ");
6      scanf("%d",&inteiro);
7      return inteiro;
8  }
9
10 int main(void){
11     int numero = le_inteiro();
12     printf("O numero lido foi: %d\n",numero);
13     return 0;
14 }
```

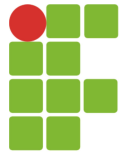


Sumário



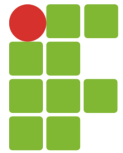
Funções: Retorno

- A partir do momento que alcança-se um comando `return expressão;` em uma função, ela retorna o valor da expressão para o invocador da função.
- Nada abaixo do comando `return` é executado.



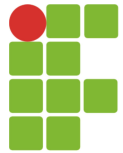
Funções: Retorno

```
1  int par(int x){
2      return x % 2 == 0;
3      printf("Isso não será executado");
4  }
```



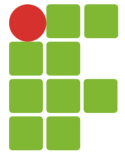
Funções: Retorno

```c
#include <stdio.h>
int par(int x){
    return x % 2 == 0;
    printf("Isso não será executado");
}

int main(void){
    int numero;
    printf("Digite um número inteiro: ");
    scanf("%d",&numero);
    if(par(numero))
        printf("O número digitado é par.\n");
    else
        printf("O número digitado é ímpar.\n");
    return 0;
}
```

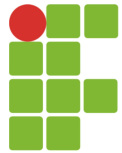


Funções: Retorno

- Como visto nos exemplos anteriores, o retorno de uma função é uma expressão.
- Podemos usar expressões diretamente em estruturas condicionais, de repetição, ou até mesmo em funções como o `printf`, sem que seja necessário guardar o valor do retorno.

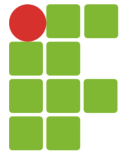


Sumário



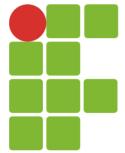
Funções: Erros Comuns

- Devemos sempre obedecer o número de parâmetros definidos na função.
- A invocação de uma função com um número de parâmetros diferentes do previsto acarretará em um erro de compilação.



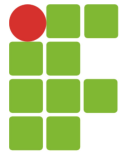
Funções: Erros Comuns

```
1  #include <stdio.h>
2
3  int soma(int a, int b) {
4      int c = a + b;
5      return c;
6  }
7
8  int main(void) {
9      printf("O resultado da soma de 2, 3 e 5 é: %d\n",soma(2,3,5));
10     return 0;
11 }
```



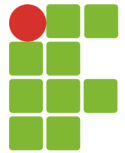
Funções: Erros Comuns

- A ordem dos argumentos e os tipos dos parâmetros também devem ser observados.
- A função a seguir recebe dois valores, dois `int`, e uma string de tamanho 22, imprime a mensagem dada pela string e retorna a soma dos dois valores.



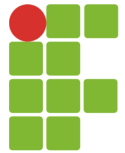
Funções: Erros Comuns

```
1  int soma_com_mensagem(int a, int b, char mensagem[22]) {
2      printf("%s", mensagem);
3      return a + b;
4  }
```



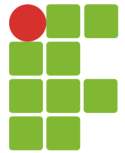
Funções: Erros Comuns

- O código a seguir tenta invocar a função, mas da maneira incorreta.
- O segundo valor passado para função é uma string, enquanto a função esperava o segundo inteiro.
- Erro de compilação!



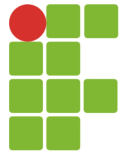
Funções: Erros Comuns

```c
#include <stdio.h>

int soma_com_mensagem(int a, int b, char mensagem[22]) {
    printf("%s", mensagem);
    return a + b;
}

int main(void) {
    printf("Resultado = %d\n", soma_com_mensagem(1, "Somando dois numeros\n", 1));
    return 0;
}
```



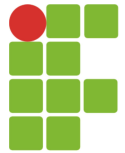
Funções: Erros Comuns

- Invocando a função com os parâmetros na ordem correta o programa funciona.

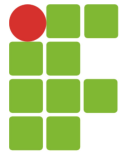


Funções: Erros Comuns

```c
#include <stdio.h>

int soma_com_mensagem(int a, int b, const char mensagem[22]) {
    printf("%s", mensagem);
    return a + b;
}

int main(void) {
    printf("Resultado = %d\n", soma_com_mensagem(1, 1, "Somando dois numeros\n"));
    return 0;
}
```

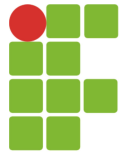


Sumário



Funções: Passagem por Valor

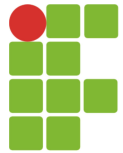
- Ao passarmos variáveis simples (não vetores) como argumentos de uma função, elas serão copiadas para as variáveis parâmetros da função.
- Isto é conhecido como **passagem por valor**.
- O valor das variáveis passadas para a função não sofrerão alterações com a execução dela.



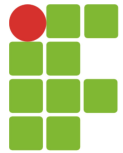
Funções: Passagem por Valor

```c
#include <stdio.h>

int incrementa(int x) {
    x++;
    return x;
}

int main(void) {
    int a = 2;
    int b = incrementa(a);
    printf("a = %d\nb = %d\n", a, b);
    return 0;
}
```

O que será impresso?

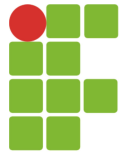


Sumário



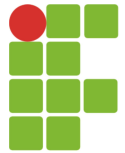
Funções: Void

- Uma função não necessariamente precisa retornar um valor.
- Quando não faz sentido que uma função retorne um valor, podemos declarar o seu tipo de retorno como `void`.



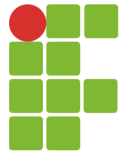
Funções: Void

- A função a seguir imprime um inteiro que é passado como parâmetro.
- Como a função não precisa retornar nada, declaramos o tipo de retorno como `void`.



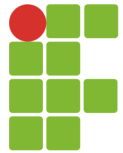
Funções: Void

```
1  void imprime_int(int x){
2      printf("Inteiro: %d\n",x);
3  }
```



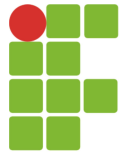
Funções: Void

```c
#include <stdio.h>

void imprime_int(int x){
    printf("Inteiro: %d\n",x);
}

int main(void){
    int numero = 10;
    imprime_int(numero);
    return 0;
}
```



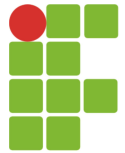
Funções: Void

- Não é necessário ter um comando `return` quando o tipo de retorno da função é `void`.
- Mas, podemos utilizá-lo mesmo assim, desde que ele não seja acompanhado de uma expressão.
- Quando a função encontra o comando `return`, ela retorna imediatamente para o ponto de invocação da mesma.
- Podemos usar o `return` em funções com tipo de retorno `void` para encerrá-las prematuramente, caso alguma condição seja atingida, por exemplo.

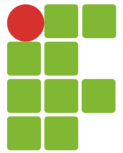


Funções: Void

```c
#include <stdio.h>

void imprime_int(int x) {
    printf("Inteiro: %d\n", x);
    return;
    printf("Isso não será impresso.\n");
}

int main(void) {
    int numero = 10;
    imprime_int(numero);
    return 0;
}
```

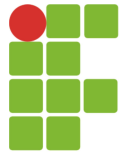


Sumário

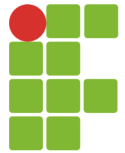


Funções: Main

- O ponto de partida de qualquer programa em C, é a função `main`.
- Ela é invocada automaticamente pelo sistema operacional quando inicia-se a execução do programa.
- A função `main` deve ter o tipo de retorno `int`, pois ela deve sinalizar ao sistema operacional se o programa funcionou corretamente ou não.
- O padrão é que o programa retorno 0, caso o programa tenha funcionado corretamente: `return 0;`.

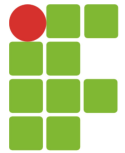


Sumário



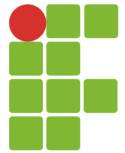
Funções: Vetores

- Diferentemente de variáveis simples, os vetores, ao serem passados como parâmetros de função, ao serem modificados pela função, persistem as modificações.
- Isto é: não é criado uma cópia do vetor!



Funções: Vetores

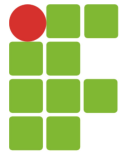
```c
#include <stdio.h>

void zera_vetor(int vetor[5]){
    int i;
    for(i=0;i<5;i++){
        vetor[i] = 0;
    }
}

void imprime_vetor(int vetor[5]){
    int i;
    for(i=0;i<5;i++){
        printf("%d ",vetor[i]);
    }
    printf("\n");
}
```



Funções: Vetores

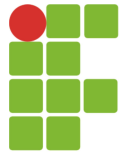
```
18  int main(void){
19      int v[5] = {1,2,3,4,5};
20      imprime_vetor(v);
21      zera_vetor(v);
22      imprime_vetor(v);
23      return 0;
24  }
```

O que será impresso?



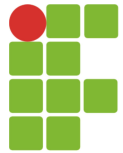
Funções: Vetores

- Também não é possível retornar vetores através de funções.



Funções: Vetores

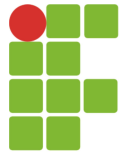
```c
#include <stdio.h>

int[] le_vetor() {
    int i;
    int v[100];
    for (i = 0; i < 100; i++) {
        scanf("%d", &v[i]);
    }
    return v;
}

void imprime_vetor(int v[100]){
    int i;
    for(i=0;i<100;i++){
        printf("%d",v[i])
    }
}

int main(void) {
    int vetor[100] = le_vetor();
    return 0;
}
```



Funções: Vetores

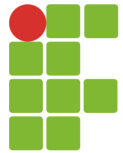
Retorno de Vetores

- Na verdade conseguimos retornar vetores, mas de outra forma: ponteiros + alocação dinâmica de memória.
- Isso ficará para outra aula.



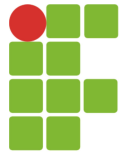
Funções: Vetores

- Como os vetores são alterados, podemos modificar o exemplo anterior para o seguinte.



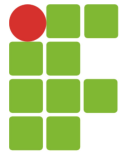
Funções: Vetores

```
1  #include <stdio.h>
2  void le_vetor(int v[100]) {
3      int i;
4      for (i = 0; i < 100; i++) {
5          scanf("%d", &v[i]);
6      }
7  }
8
9  void imprime_vetor(int v[100]) {
10     int i;
11     for (i = 0; i < 100; i++) {
12         printf("%d", v[i]);
13     }
14     printf("\n");
15 }
16
```



Funções: Vetores

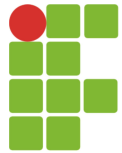
```
17  int main(void) {
18      int vetor[100];
19      le_vetor(vetor);
20      imprime_vetor(vetor);
21      return 0;
22  }
```



Funções: Vetores

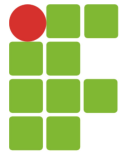
- Durante a definição de uma função que receba o vetor, podemos omitir o tamanho do vetor entre os colchetes.
- Para que a função saiba qual o tamanho do vetor, podemos passar uma variável inteira extra.
- Ex:

```
1  void exemplo_funcao(int vetor[], int n){
2      ...
3  }
```



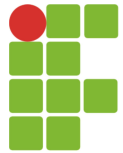
Funções: Vetores

- Mas qual a vantagem desta forma de declaração? Afinal, precisamos de dois parâmetros agora...
- A vantagem é que, com esta forma, é possível criar funções que funcionem com qualquer tamanho de vetor, pois o parâmetro do tamanho do vetor está desacoplado do mesmo.
- Vamos modificar as funções `le_vetor` e `imprime_vetor` para este fim.



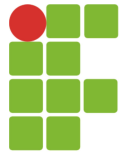
Funções: Vetores

```
1  void le_vetor(int v[], int n) {
2      int i;
3      for (i = 0; i < n; i++) {
4          scanf("%d", &v[i]);
5      }
6  }
7
8  void imprime_vetor(int v[], int n) {
9      int i;
10     for (i = 0; i < n; i++) {
11         printf("%d ", v[i]);
12     }
13     printf("\n");
14 }
```



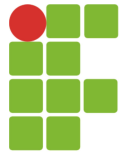
Funções: Vetores

- Com isso, podemos escrever programas que leiam e imprimam vetores de tamanhos diferentes reutilizando as funções.



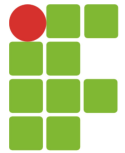
Funções: Vetores

```c
#include <stdio.h>

void le_vetor(int v[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        scanf("%d", &v[i]);
    }
}

void imprime_vetor(int v[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");
}
```

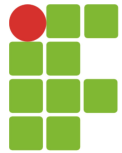


Funções: Vetores

```
18  int main(void) {
19      int vetor[5], vetor2[10];
20      le_vetor(vetor,5);
21      le_vetor(vetor2,10);
22      imprime_vetor(vetor,5);
23      imprime_vetor(vetor2,10);
24      return 0;
25  }
```

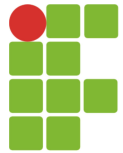


Sumário



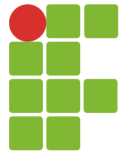
Funções: Protótipos

- Até o momento estamos declarando as funções antes da função `main`.
- O que acontece se as declararmos após a função `main`?
- Dependendo do compilador, podemos ter um **erro de compilação**.



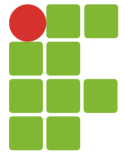
Funções: Protótipos

```c
#include <stdio.h>

int main(void) {
    int x = 5, y = 7;
    int r = soma(2, 3);
    printf("%d\n", r);
    r = soma(x, y);
    printf("%d\n", r);
    printf("%d\n", soma(10, -20));
    return 0;
}

int soma(int a, int b) {
    int c = a + b;
    return c;
}
```



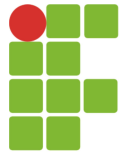
Funçoes: Protótipos

- Para evitar isso, podemos definir os **protótipos das funções**.
- Uma vez que o protótipo esteja definido, no início do arquivo fonte, podemos colocar as implementações da função onde bem entender.
- Os protótipos correspondem à **assinatura** da função, isto é, a primeira linha da mesma, seguida de ';'.
- `tipo_retorno nome(tipo1 parametro1,...,tipoN parametroN);`



Funções: Protótipos

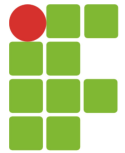
```c
#include <stdio.h>

int soma(int a, int b);

int main(void) {
    int x = 5, y = 7;
    int r = soma(2, 3);
    printf("%d\n", r);
    r = soma(x, y);
    printf("%d\n", r);
    printf("%d\n", soma(10, -20));
    return 0;
}

int soma(int a, int b) {
    int c = a + b;
    return c;
}
```



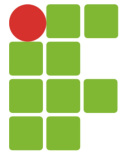
Funções: Protótipos

- Em geral, programas em C podem ser organizados da seguinte forma:

```
1  // Inclusão de cabeçalhos
2  #include <...>
3
4  // Definição de protótipos
5  int soma(int a,int b);
6  ...
7
8  // Função Main
9  int main(void){
10     ...
11     return 0;
12 }
13
14 // Implementação das funções
15 int soma(int a,int b){
16     ...
17 }
18 ...
```

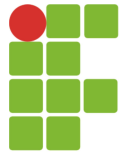


Sumário



Função: Invocação

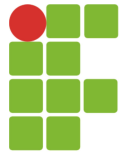
- Nos exemplos anteriores, a função `main` invocava as funções definidas.
- Contudo, qualquer função pode invocar outra função.



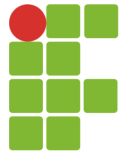
Função: Invocação

```c
#include <stdio.h>

int foo(int x);
int bar(int y);

int main(void) {
    int c = 5;
    c = foo(c);
    printf("c = %d\n", c);
    return 0;
}

int foo(int x) {
    x++;
    x = bar(x);
    return x;
}

int bar(int y) {
    y *= 2;
    return y;
}
```

O que será impresso?

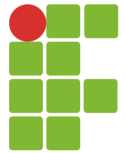


Sumário



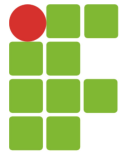
Funções: Escopo

- Variáveis **globais** são aquelas visíveis por qualquer função. Elas são declaradas fora de qualquer função.
- Variáveis **locais** são aquelas declaradas dentro de funções (incluindo os parâmetros). Elas só existem dentro da função. Após o término da função, elas deixam de existir.



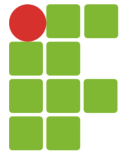
Funções: Escopo

```
1  // Inclusão de cabeçalhos
2  #include <...>
3
4  // Declaração de variáveis globais
5  int global;
6  ...
7
8  // Definição de protótipos
9  int soma(int a,int b);
10 ...
11
12 // Função Main
13 int main(void){
14     ...
15     return 0;
16 }
17
18 // Implementação das funções
19 int soma(int a,int b){
20     ...
21 }
22 ...
```



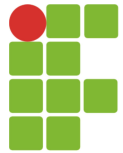
Funções: Escopo

- O escopo de uma variável determina quais as partes do código que conseguem acessá-la.
- Em suma:
  - As variáveis globais são vistas por qualquer função.
  - As variáveis locais são vistas só pelas funções onde foram declaradas.



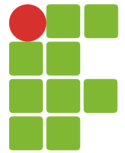
Funções: Escopo

```
1  // Inclusão de cabeçalhos
2  #include <...>
3
4  int global;
5
6  void foo();
7  int bar(int b);
8  ...
9
10 int main(void){
11     int local_main;
12     // são visíveis global e local_main
13     return 0;
14 }
15
16 void foo(){
17     //  global é visível
18 }
19
20 int bar(int b){
21     int c;
22     // são visíveis global, b e c
23 }
```



Funções: Escopo

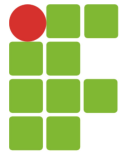
- É possível declarar variáveis locais com o mesmo nome da variável global.
- Neste caso, a variável local irá se sobressair sobre a variável global.



Funções: Escopo

```c
#include <stdio.h>

int nota = 10;

void foo();

int main(void){
    printf("%d\n",nota);
    nota = 20;
    printf("%d\n",nota);
    foo();
    printf("%d\n",nota);
    return 0;
}

void foo(void){
    int nota;
    nota = 5;
    printf("%d\n",nota);
}
```

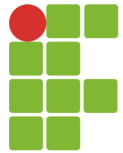
O que será impresso?



Funções: Escopo

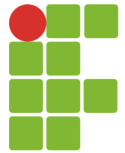
Boas Práticas de Programação

- O uso de variáveis **globais** deve ser evitado quando possível.
- Várias partes do código podem manipular as variáveis globais, o que deixa o código difícil de ler, depurar e manter.
- Ao invocar qualquer função, é difícil inferir sobre o estado das variáveis globais.



Funções: Escopo

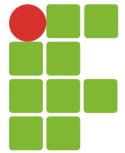
- Como os parâmetros das funções estão em um escopo diferente das variáveis de onde a função foi invocada, não há problema de reutilizar nome de variáveis.
- Não haverá conflito, uma vez que estamos falando de escopos distintos.



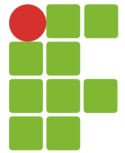
Funções: Escopo

```c
#include <stdio.h>

int incrementa(int x) {
    x++;
    return x;
}

int main(void) {
    int x = 3;
    int y = incrementa(x);
    printf("x = %d y = %d\n", x, y);
    return 0;
}
```

O que será impresso?

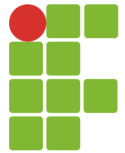


Sumário

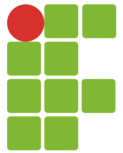


Exemplos

- Mostraremos agora como aplicar o conceito de modularização em alguns problemas.



Sumário



Exemplo: Raiz Quadrada

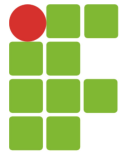
Problema

- Dado um número real positivo, computar sua raiz quadrada de acordo com o método de Newton.
- A raiz quadrada de um número z é a raiz da equação:

$$f(x) = x^2 - z$$

- O método de Newton funciona através de iterações. Na primeira iteração, $x_1 = \frac{z}{2}$. Na $n+1$ iteração, temos que:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

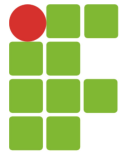


Exemplo: Raiz Quadrada

- Vamos dividir o problema por partes e aplicar o conceito de modularização.
- Primeiramente, vamos calcular a função $f(x)$:

```
double f(double x,double z){
    return x*x - z;
}
```

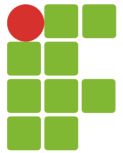
- Computacionalmente falando, passamos o parâmetro z para função também!.



Exemplo: Raiz Quadrada

- Agora, calculamos a derivada da função.
- Note que, com z é constante, ele some na derivada, por isso a implementação da função derivada só possui um parâmetro.

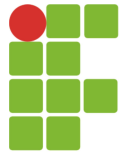
```
double f_linha(double x){
    return 2*x;
}
```



Exemplo: Raiz Quadrada

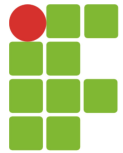
- Agora definiremos a função que aplica o método de Newton por n iterações, onde n é um parâmetro da entrada.
- Quanto maior o n, mais precisa será a resposta.
- A função retornará a raiz quadrada do número z.

```
double newton(double z, int n) {
    int i;
    double x_atual = z / 2;
    double x_proximo = x_atual;
    for (i = 2; i <= n; i++) {
        x_proximo = x_atual - f(x_atual, z) / f_linha(x_atual);
        x_atual = x_proximo;
    }
    return x_atual;
}
```



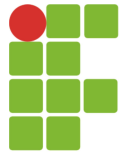
Exemplo: Raiz Quadrada

```c
#include <stdio.h>

double newton(double z,int n);
double f(double x,double z);
double f_linha(double x);

int main(void){
    double z;
    int n;
    printf("Digite o valor de z: ");
    scanf("%lf",&z);
    printf("Digite o número de iterações para o método de Newton: ");
    scanf("%d",&n);
    double raiz = newton(z,n);
    printf("A raiz de %.2f é %.2f\n",z,raiz);
    return 0;
}
```

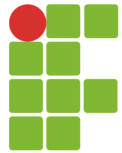


Exemplo: Raiz Quadrada

```
19  double newton(double z, int n) {
20      int i;
21      double x_atual = z / 2;
22      double x_proximo = x_atual;
23      for (i = 2; i <= n; i++) {
24          x_proximo = x_atual - f(x_atual, z) / f_linha(x_atual);
25          x_atual = x_proximo;
26      }
27      return x_atual;
28  }
29
30
31  double f(double x,double z){
32      return x*x - z;
33  }
34
35  double f_linha(double x){
36      return 2*x;
37  }
```

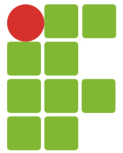


Sumário



Exemplo: Crivo de Eratóstenes

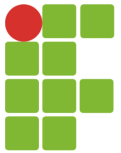
- O crivo de Eratóstenes é um método que permite computar todos os números primos de um intervalo $[1, n]$.
- Ele funciona da seguinte forma:
  - Computa-se a raiz quadrada, arredondada para baixo, de n, chame o resultado de y. Este será o último número a ser checado.
  - Crie uma lista de todos os números naturais de 2 até n.
  - 2 é primo, então eliminamos todos os múltiplos de 2 desta lista.
  - O próximo número não eliminado, 3, tem que ser primo, repetimos o procedimento.
  - E assim fazemos para os demais números, até que o próximo número não eliminado seja maior que y. Os números que não foram eliminados são primos.



Exemplo: Crivo de Eratóstenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

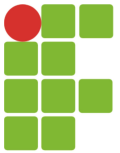


Exemplo: Crivo de Eratóstenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

2

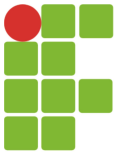


Exemplo: Crivo de Eratóstenes

	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~
41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~
51	~~52~~	53	~~54~~	55	~~56~~	57	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	69	~~70~~
71	~~72~~	73	~~74~~	75	~~76~~	77	~~78~~	79	~~80~~
81	~~82~~	83	~~84~~	85	~~86~~	87	~~88~~	89	~~90~~
91	~~92~~	93	~~94~~	95	~~96~~	97	~~98~~	99	~~100~~
101	~~102~~	103	~~104~~	105	~~106~~	107	~~108~~	109	~~110~~
111	~~112~~	113	~~114~~	115	~~116~~	117	~~118~~	119	~~120~~

Prime numbers

2

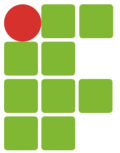


Exemplo: Crivo de Eratóstenes

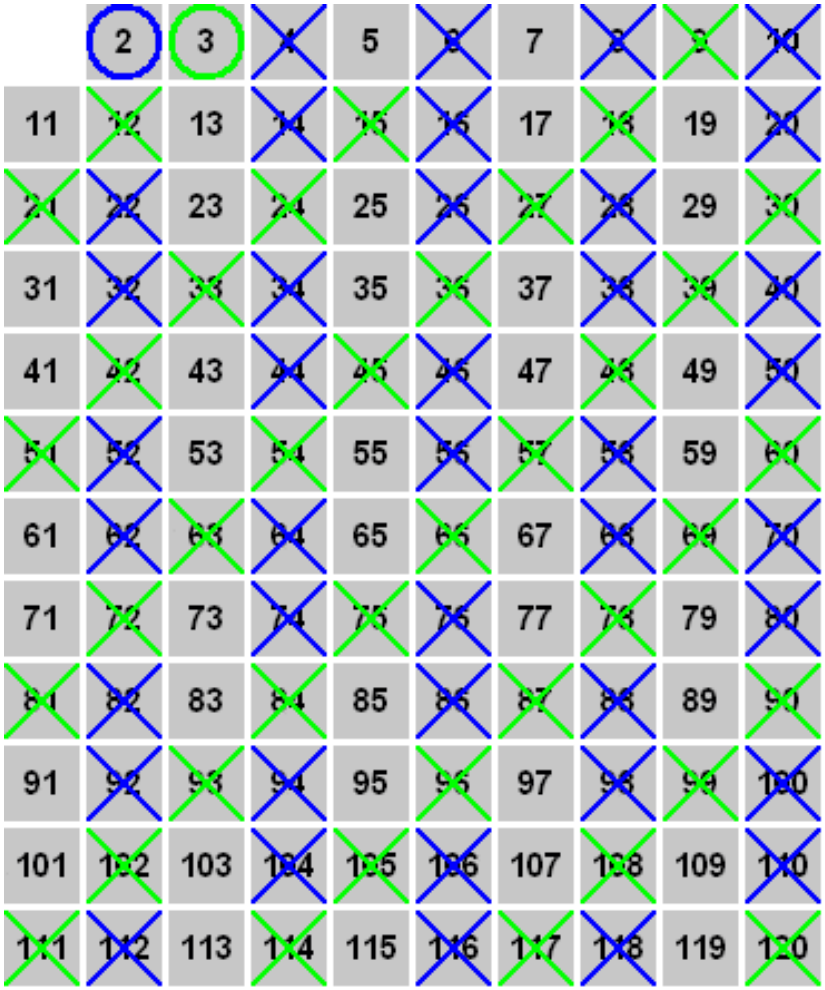
	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
31	~~32~~	33	~~34~~	35	~~36~~	37	~~38~~	39	~~40~~
41	~~42~~	43	~~44~~	45	~~46~~	47	~~48~~	49	~~50~~
51	~~52~~	53	~~54~~	55	~~56~~	57	~~58~~	59	~~60~~
61	~~62~~	63	~~64~~	65	~~66~~	67	~~68~~	69	~~70~~
71	~~72~~	73	~~74~~	75	~~76~~	77	~~78~~	79	~~80~~
81	~~82~~	83	~~84~~	85	~~86~~	87	~~88~~	89	~~90~~
91	~~92~~	93	~~94~~	95	~~96~~	97	~~98~~	99	~~100~~
101	~~102~~	103	~~104~~	105	~~106~~	107	~~108~~	109	~~110~~
111	~~112~~	113	~~114~~	115	~~116~~	117	~~118~~	119	~~120~~

Prime numbers

2 3

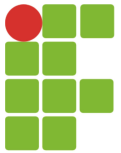


Exemplo: Crivo de Eratóstenes

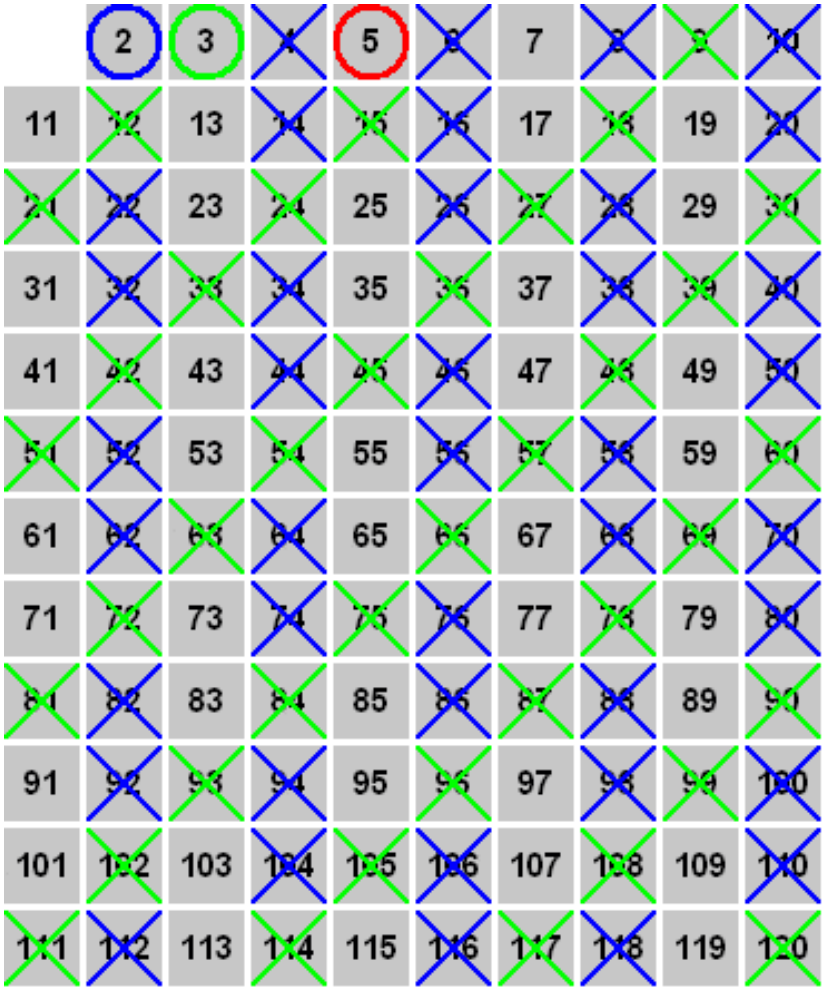


Prime numbers

2 3



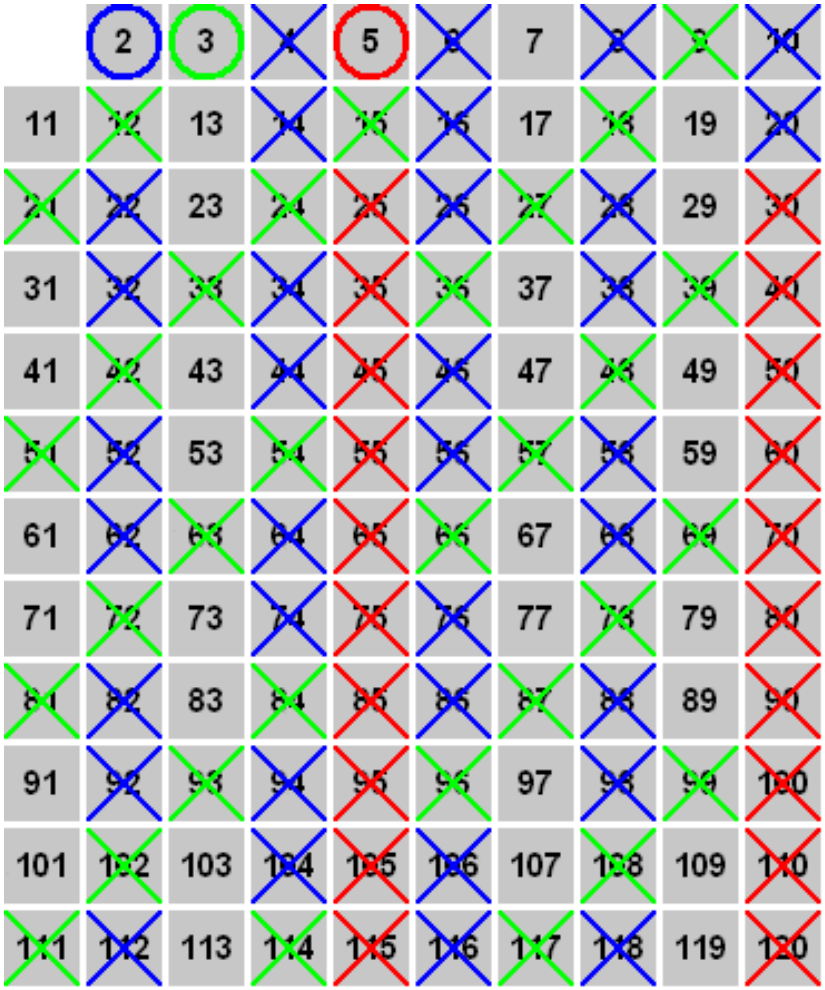
Exemplo: Crivo de Eratóstenes



Prime numbers

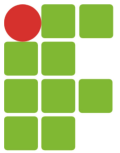
2 3 5

Exemplo: Crivo de Eratóstenes

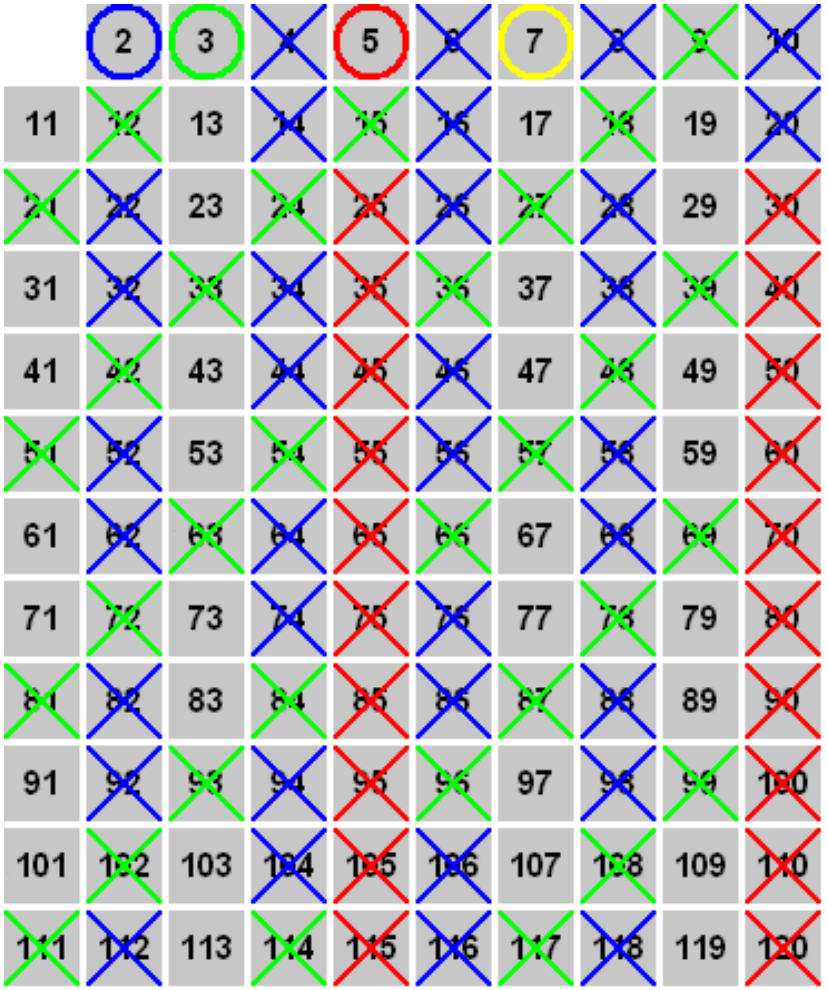


Prime numbers

2 3 5

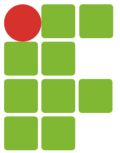


Exemplo: Crivo de Eratóstenes

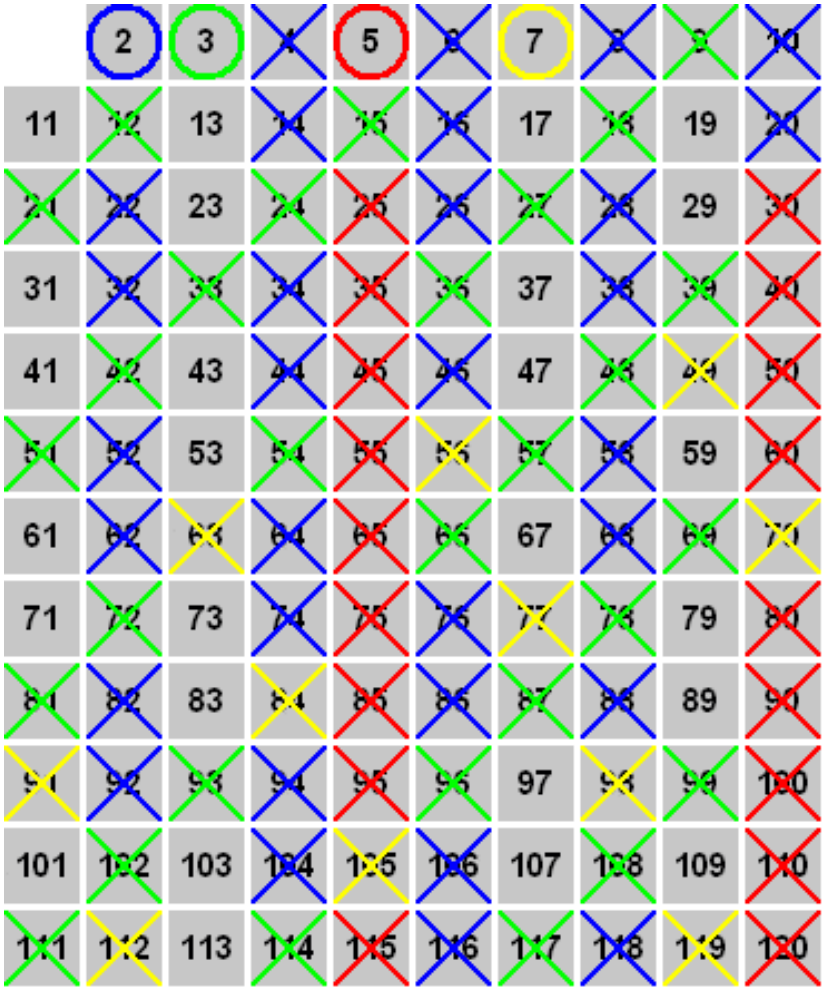


Prime numbers

2 3 5 7



Exemplo: Crivo de Eratóstenes



Prime numbers

2 3 5 7

Exemplo: Crivo de Eratóstenes

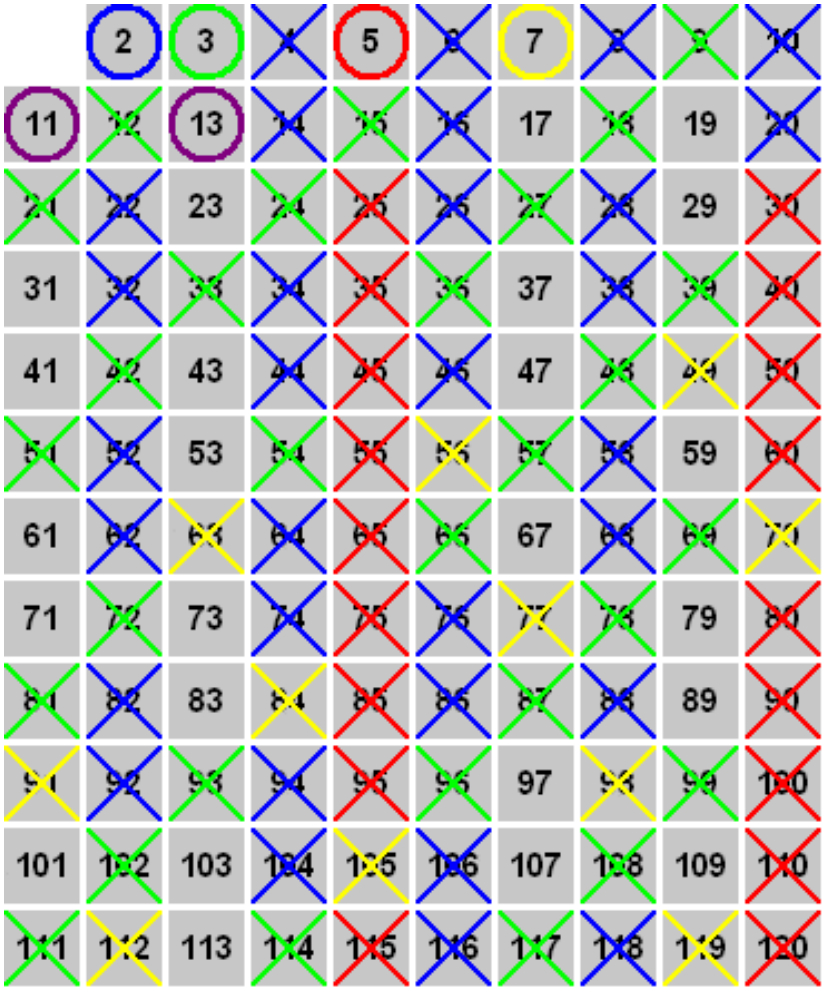
	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

2 3 5 7

11

Exemplo: Crivo de Eratóstenes

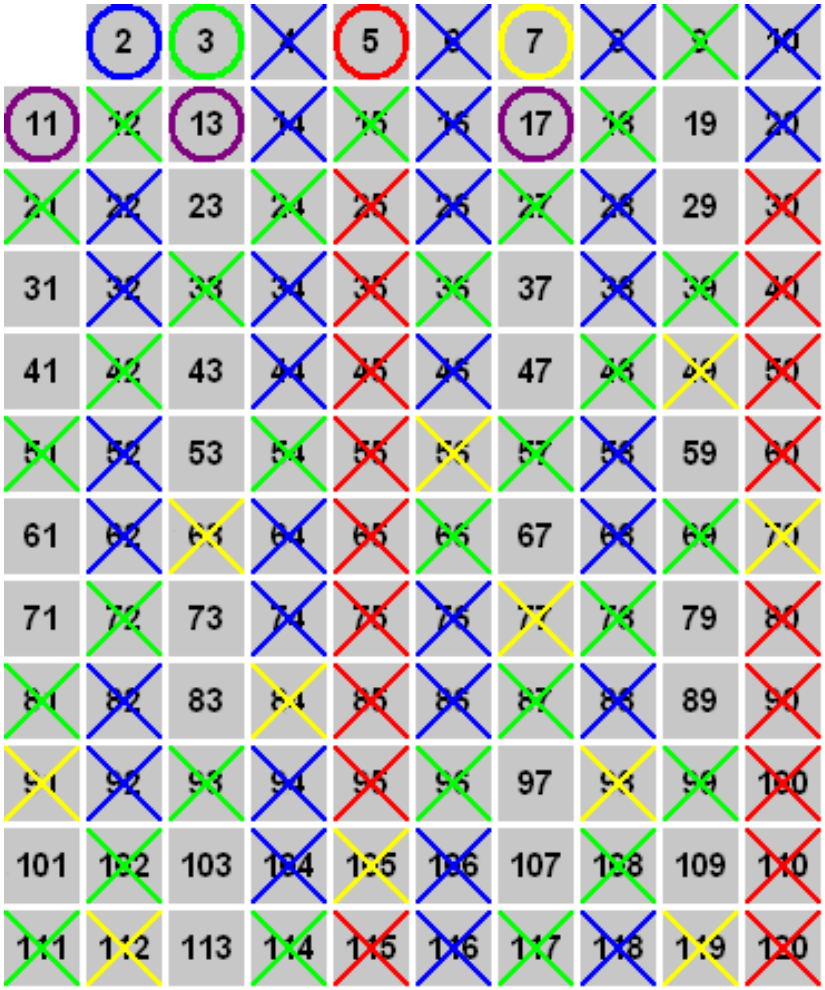


Prime numbers

2 3 5 7

11 13

Exemplo: Crivo de Eratóstenes

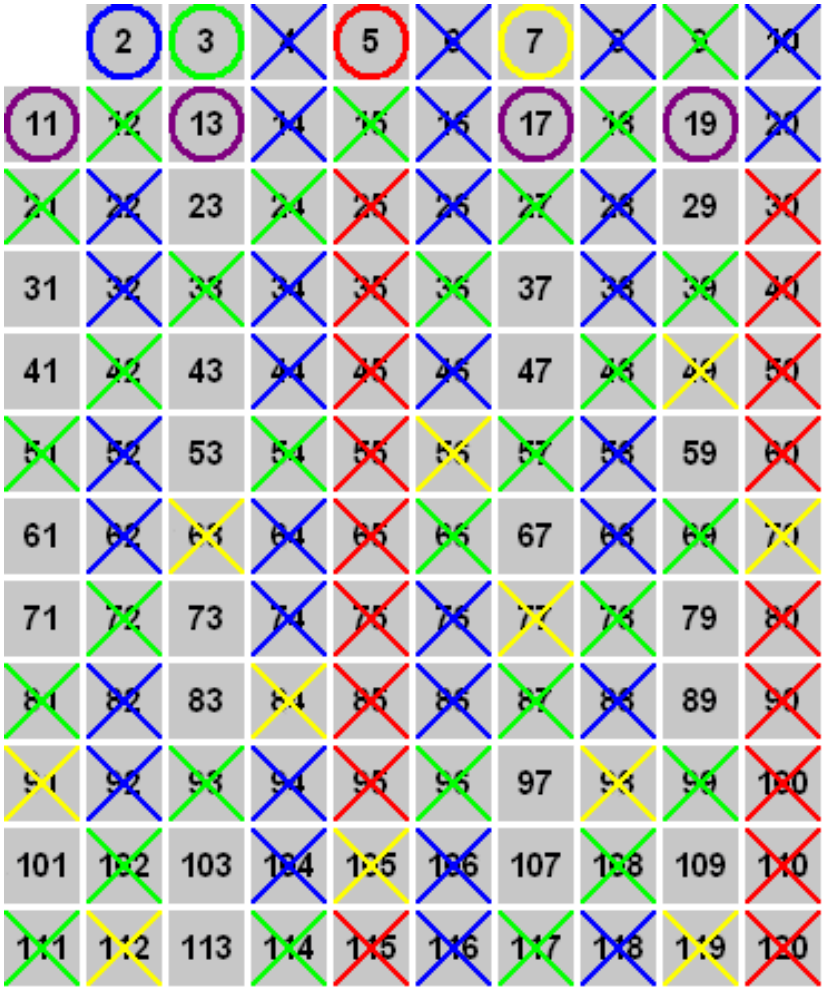


Prime numbers

2 3 5 7

11 13 17

Exemplo: Crivo de Eratóstenes

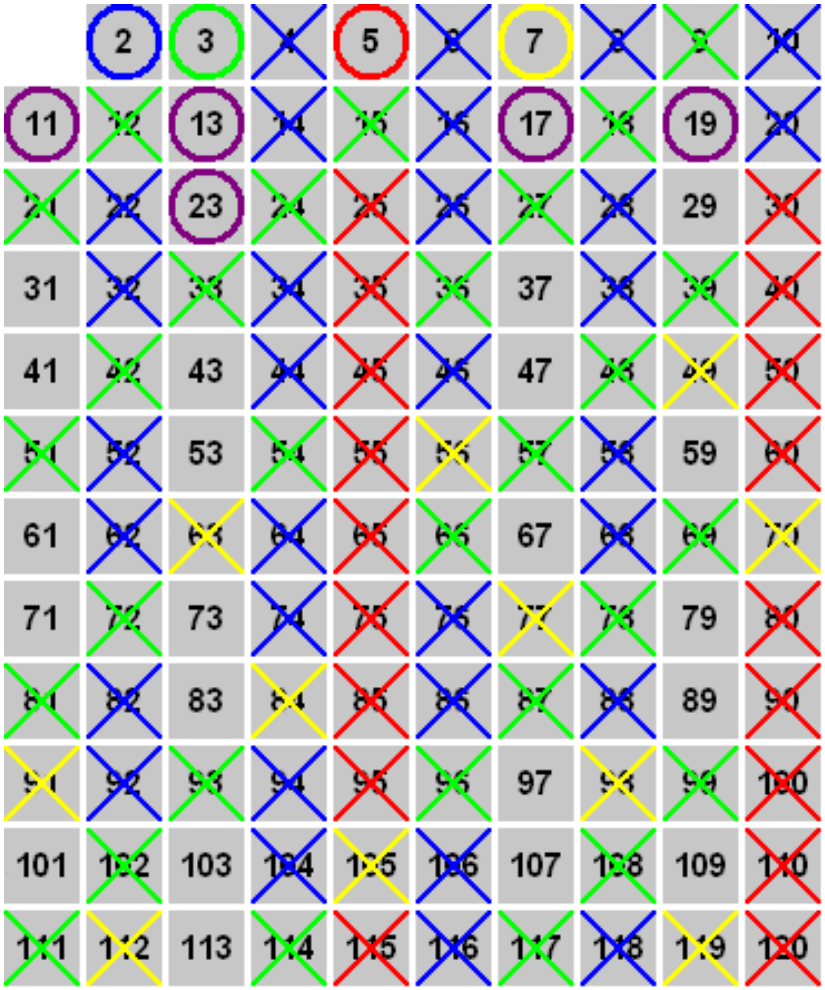


Prime numbers

2 3 5 7

11 13 17 19

Exemplo: Crivo de Eratóstenes



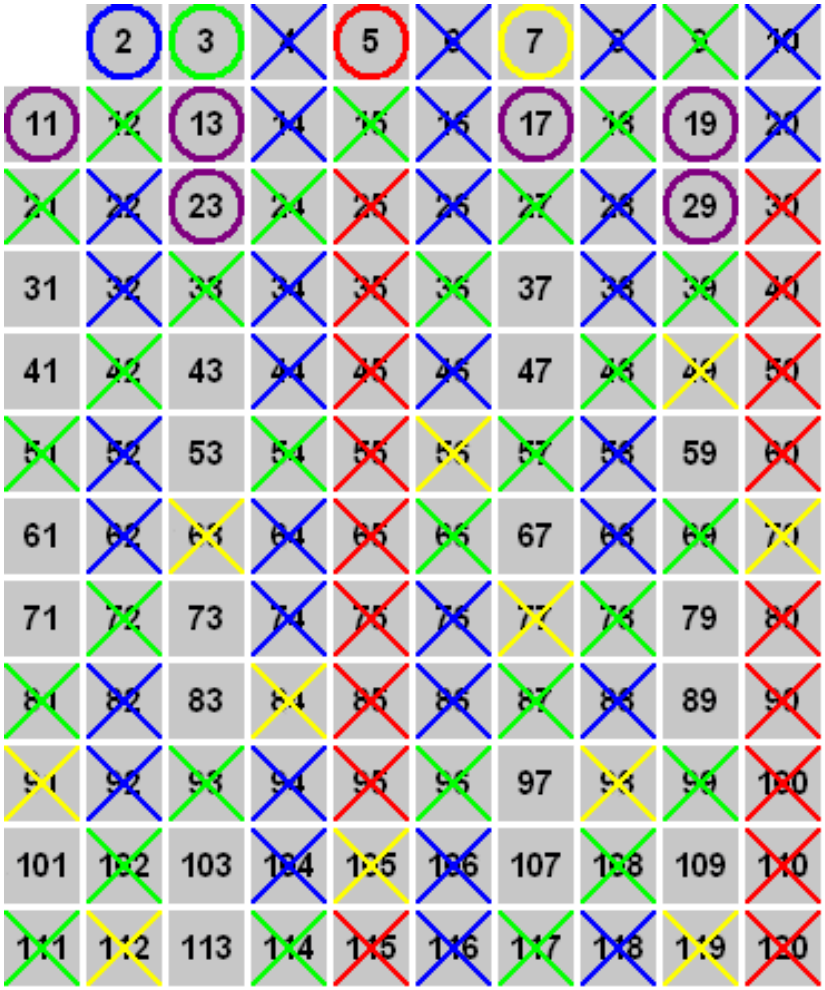
Prime numbers

2 3 5 7

11 13 17 19

23

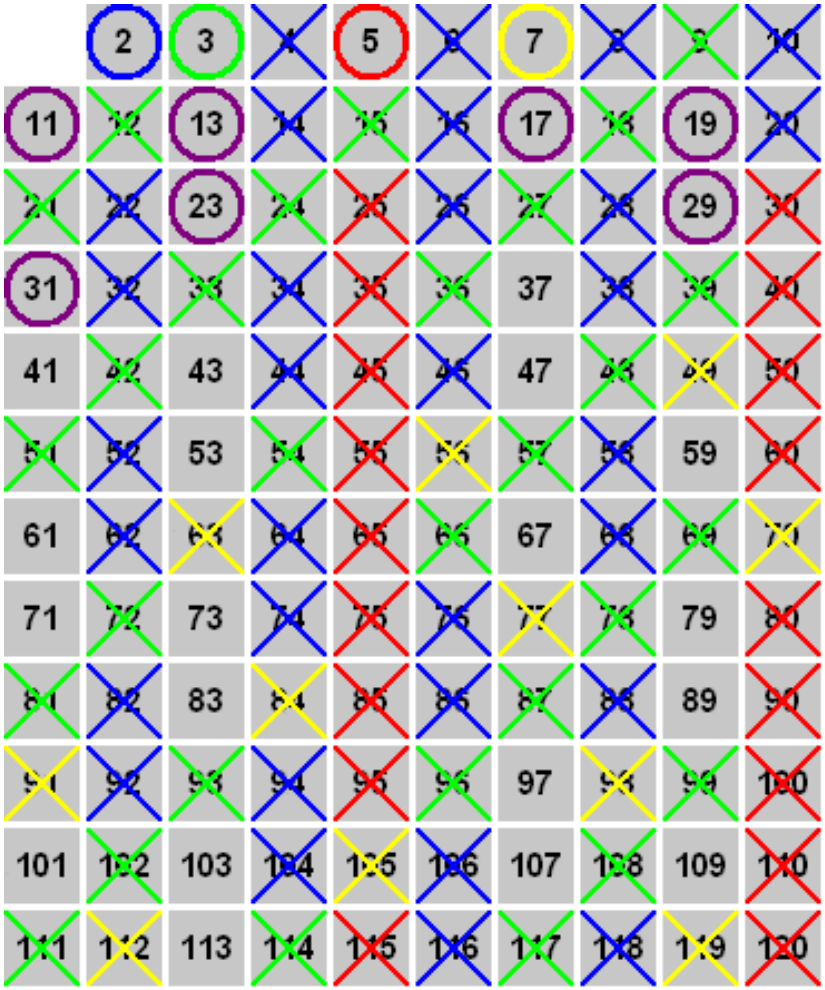
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29		

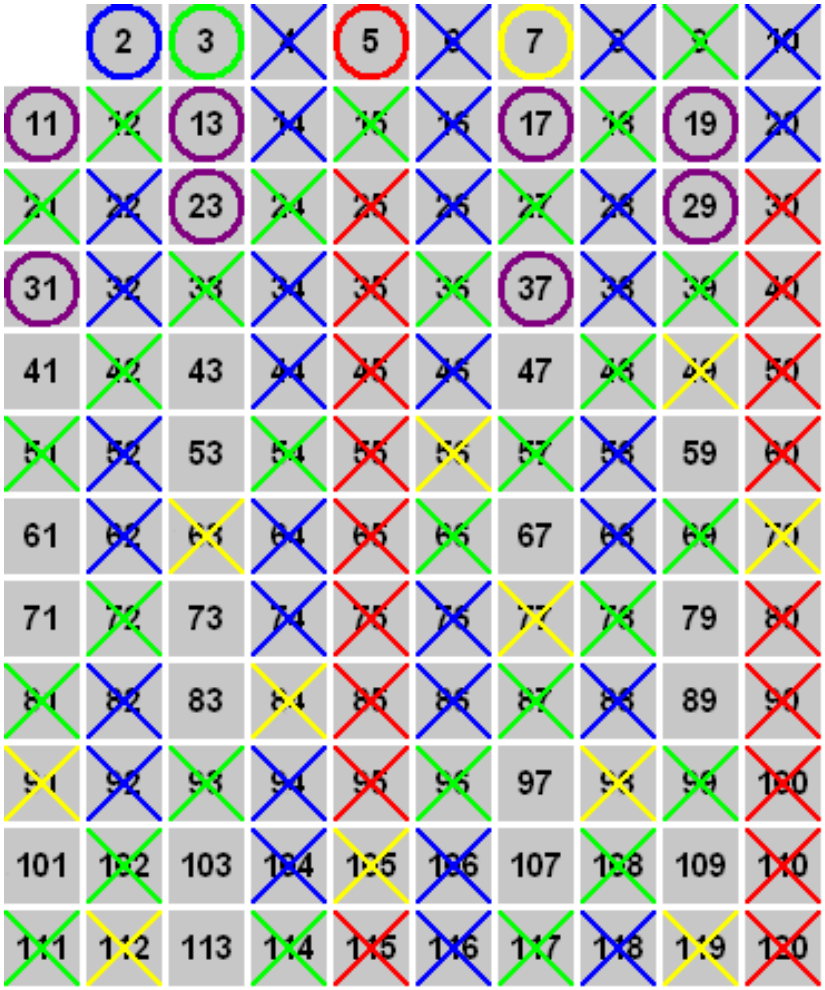
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	

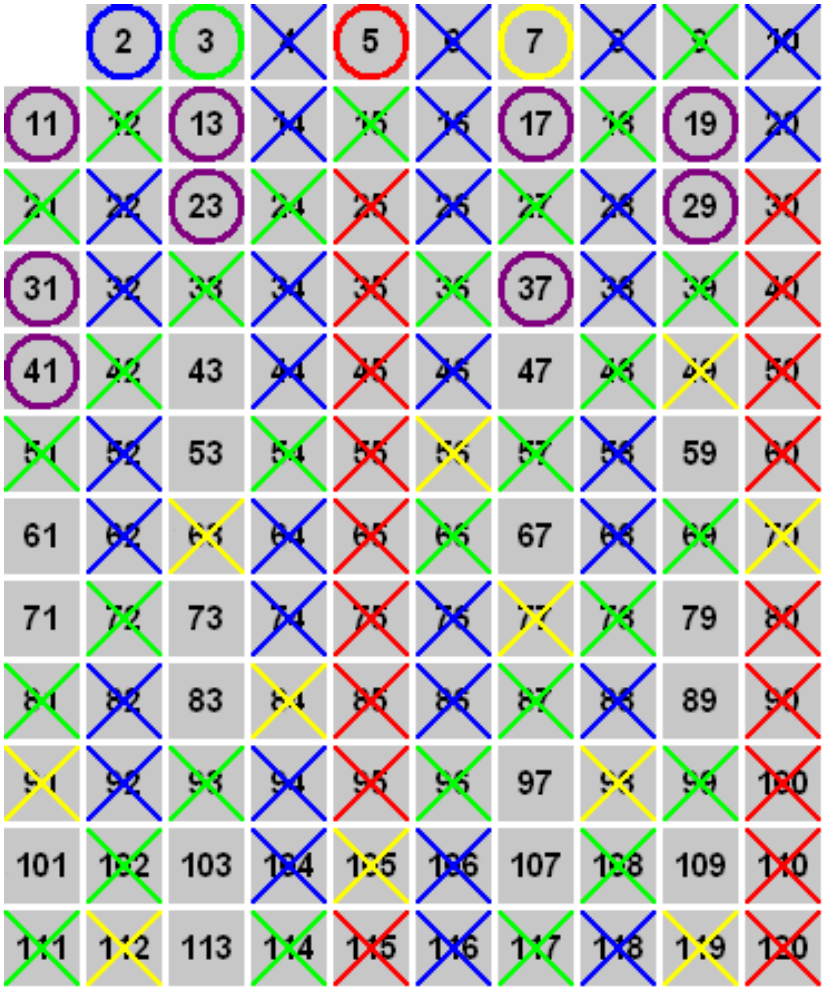
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37

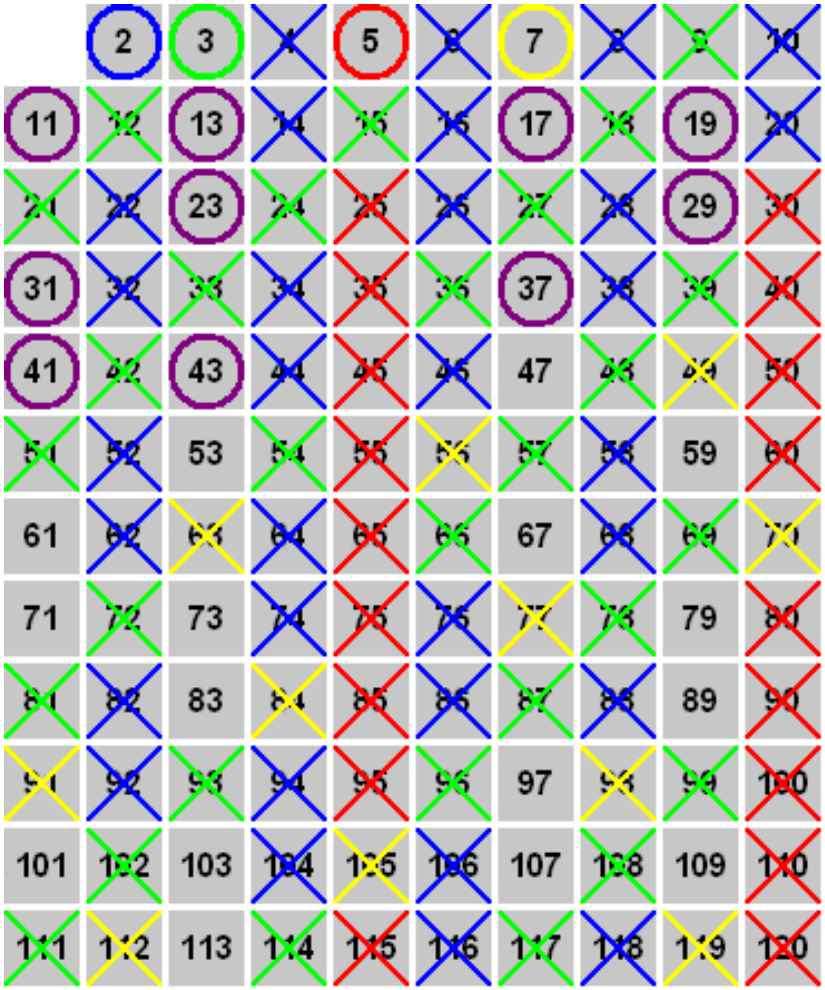
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41			

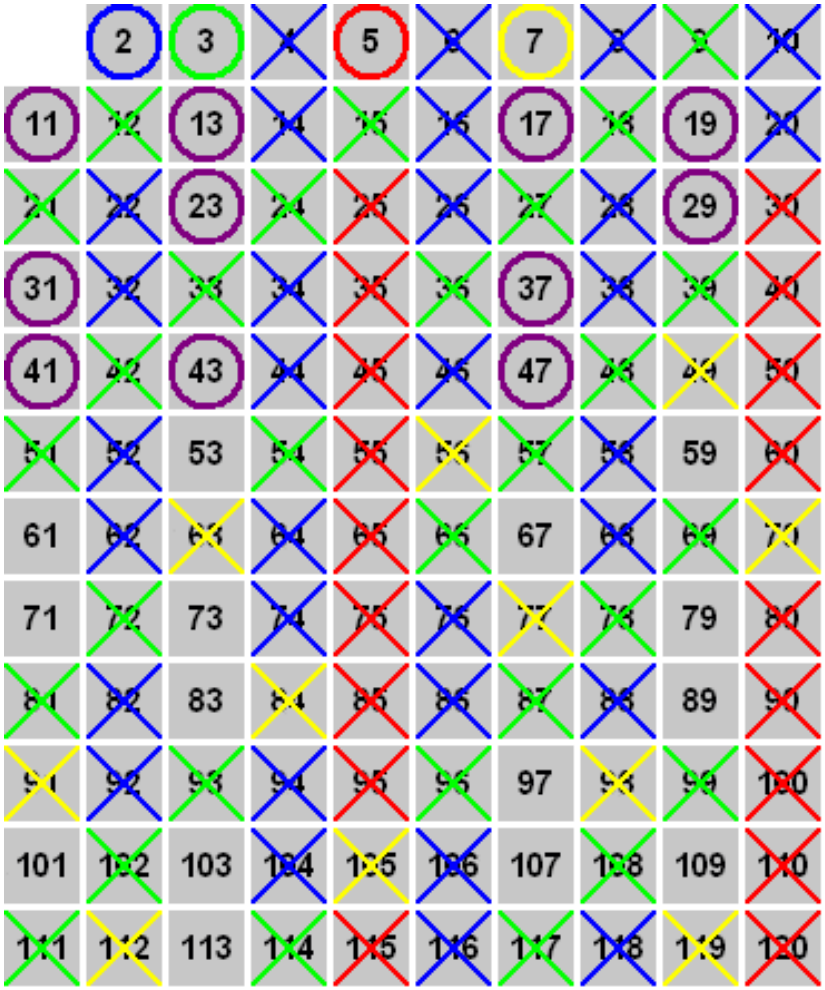
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43		

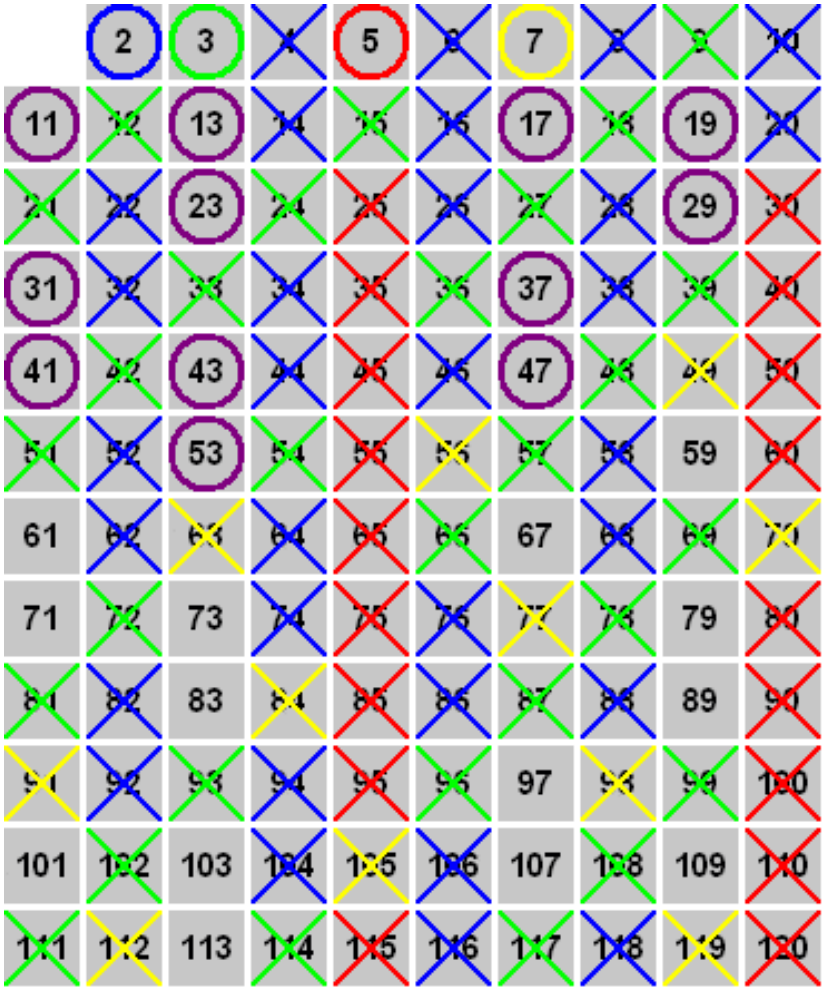
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	

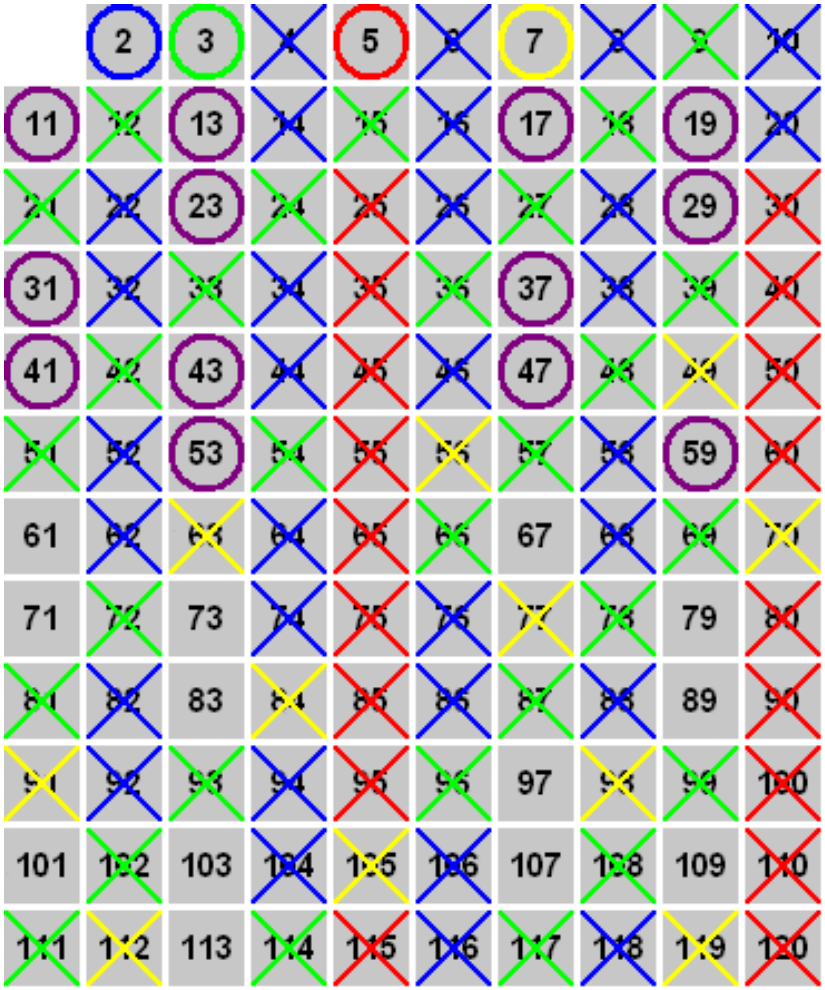
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53

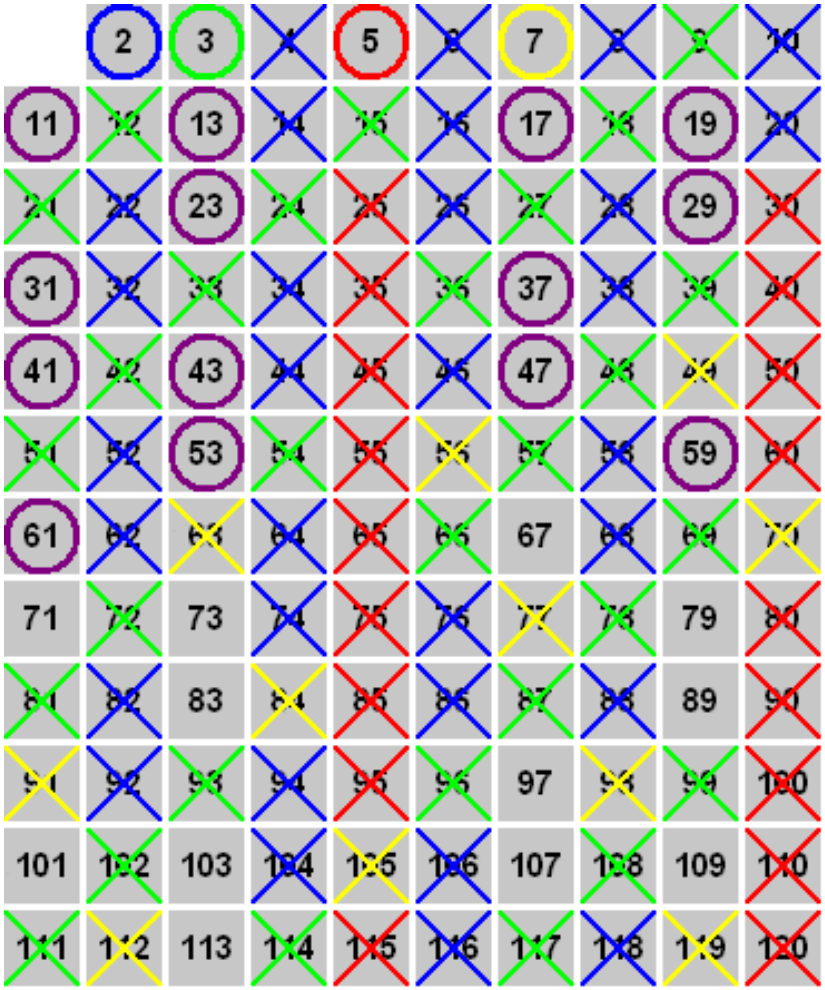
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59			

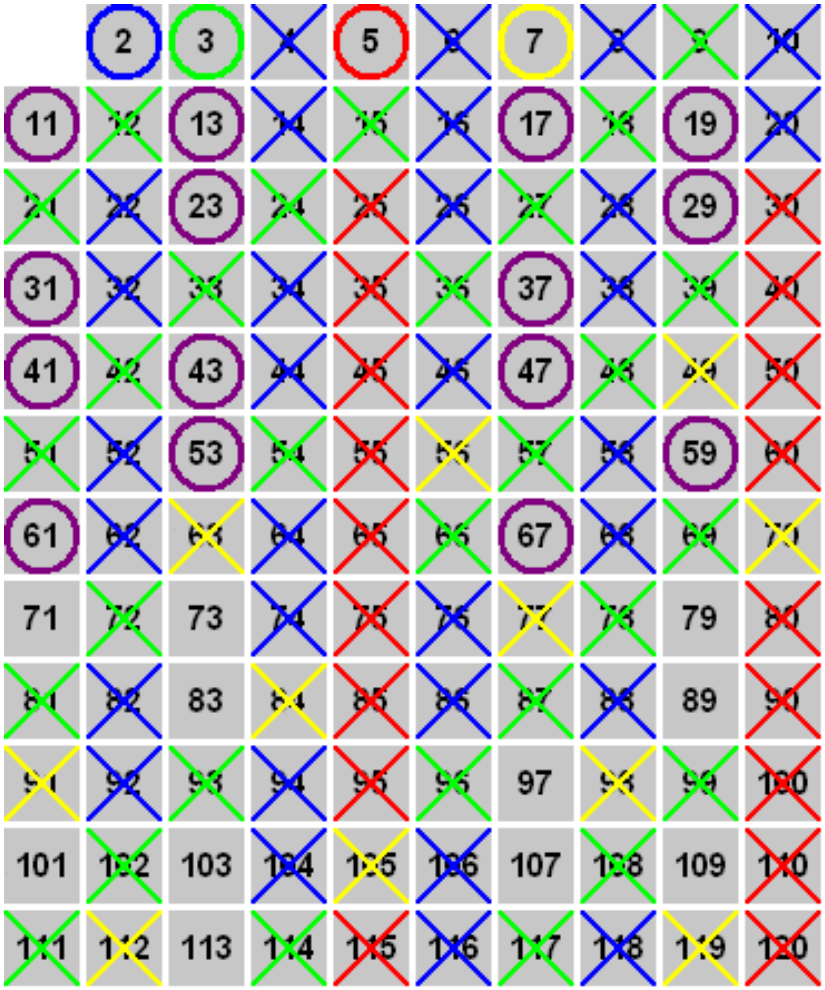
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61		

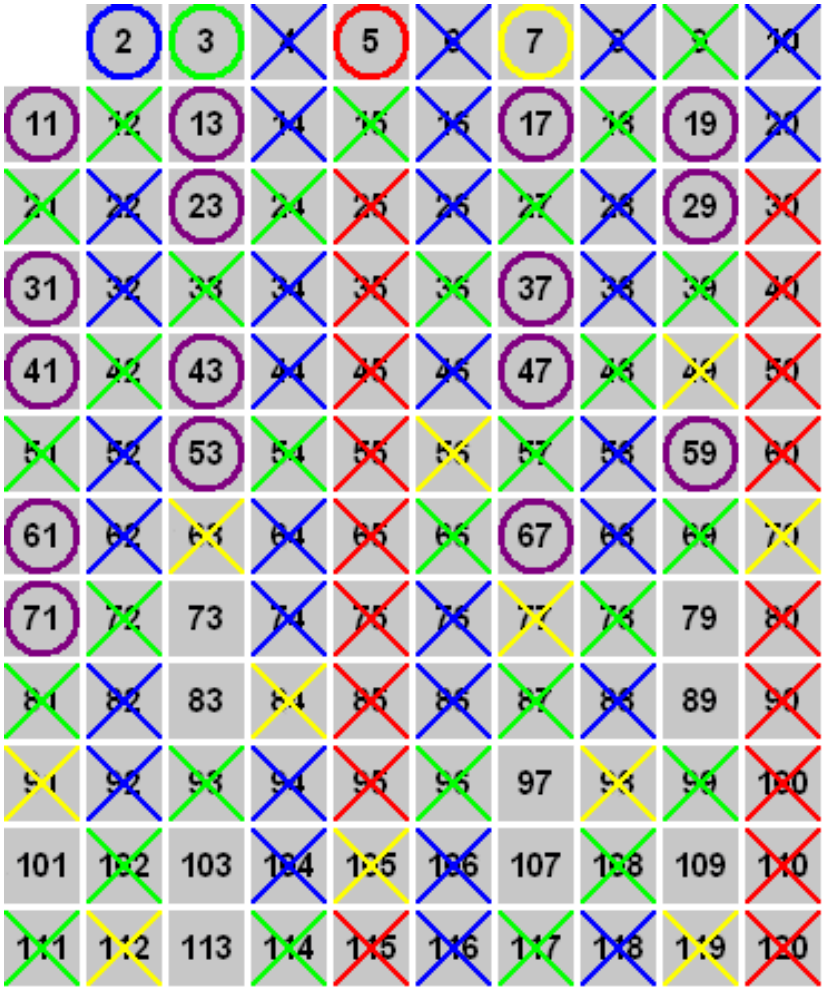
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	

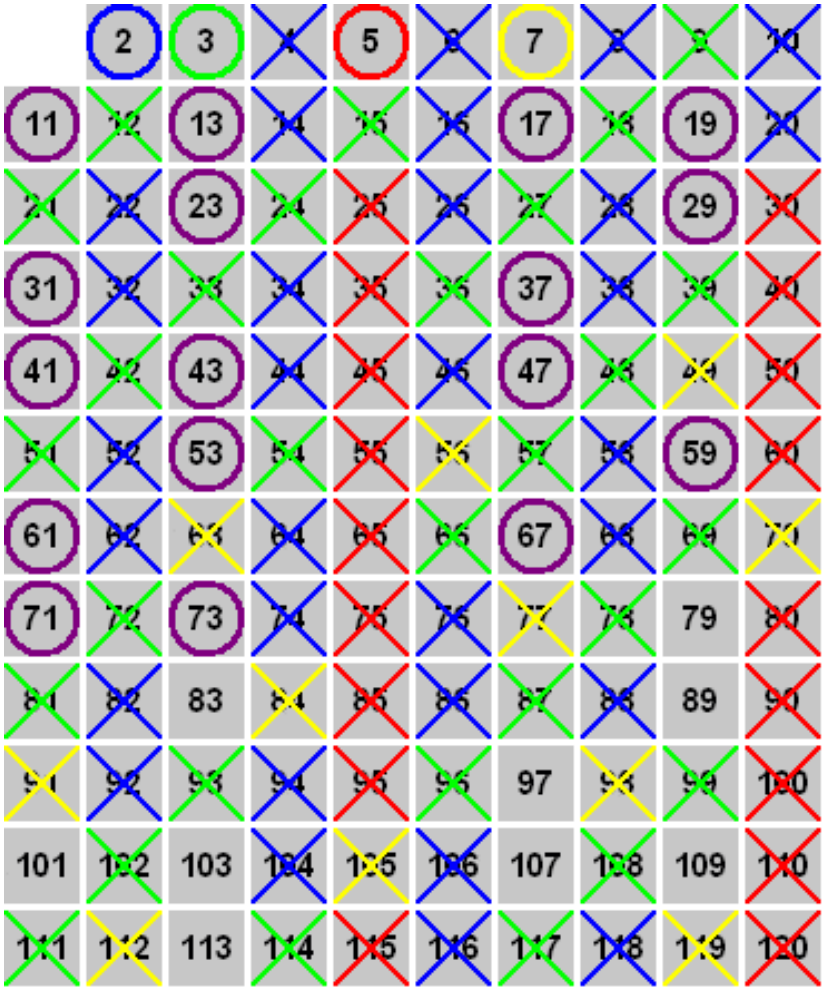
Exemplo: Crivo de Eratóstenes



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71

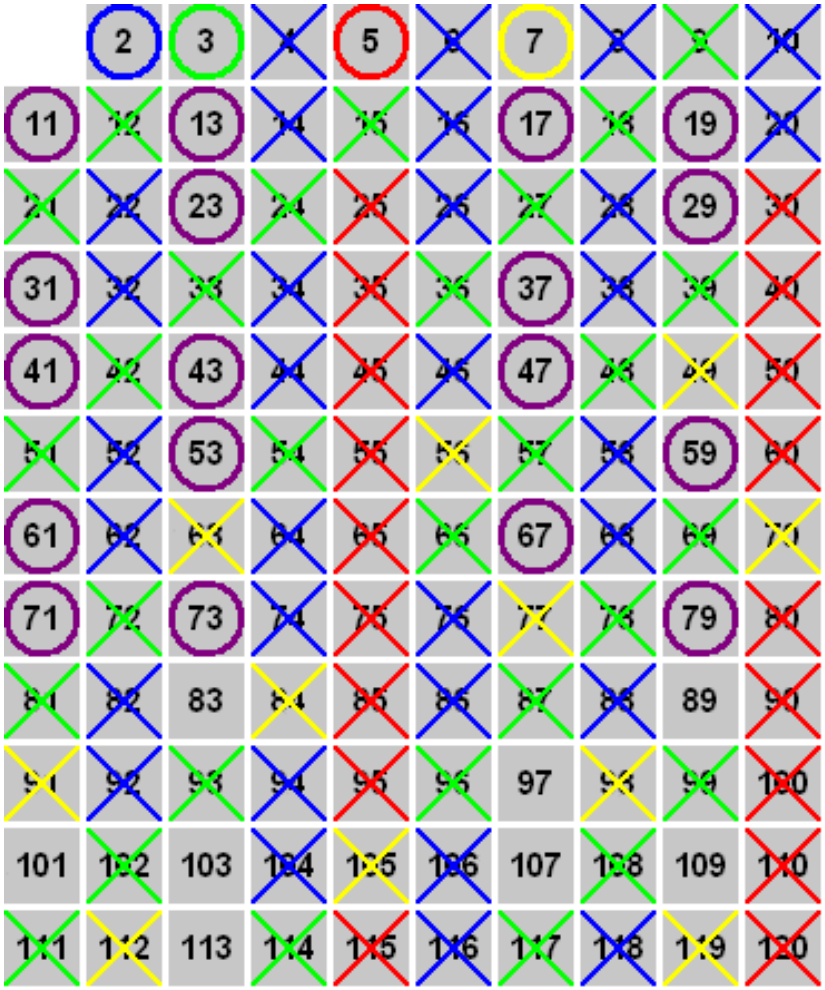
Exemplo: Crivo de Eratóstenes



Prime numbers

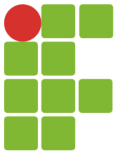
2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73			

Exemplo: Crivo de Eratóstenes

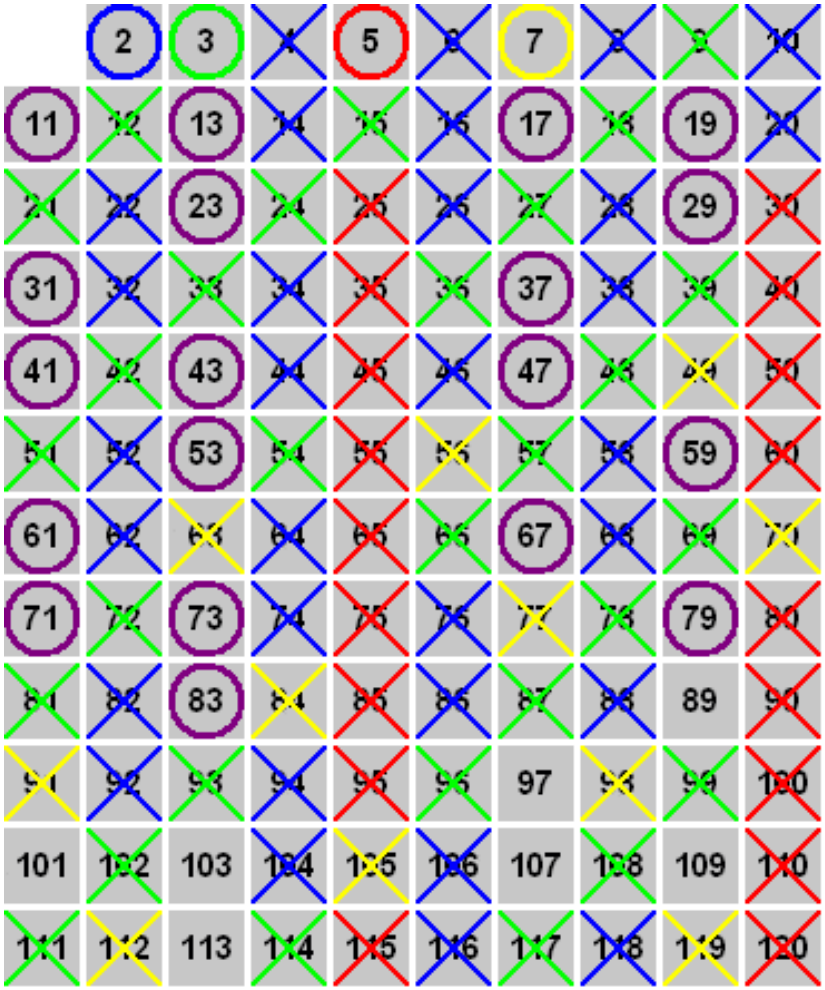


Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79		



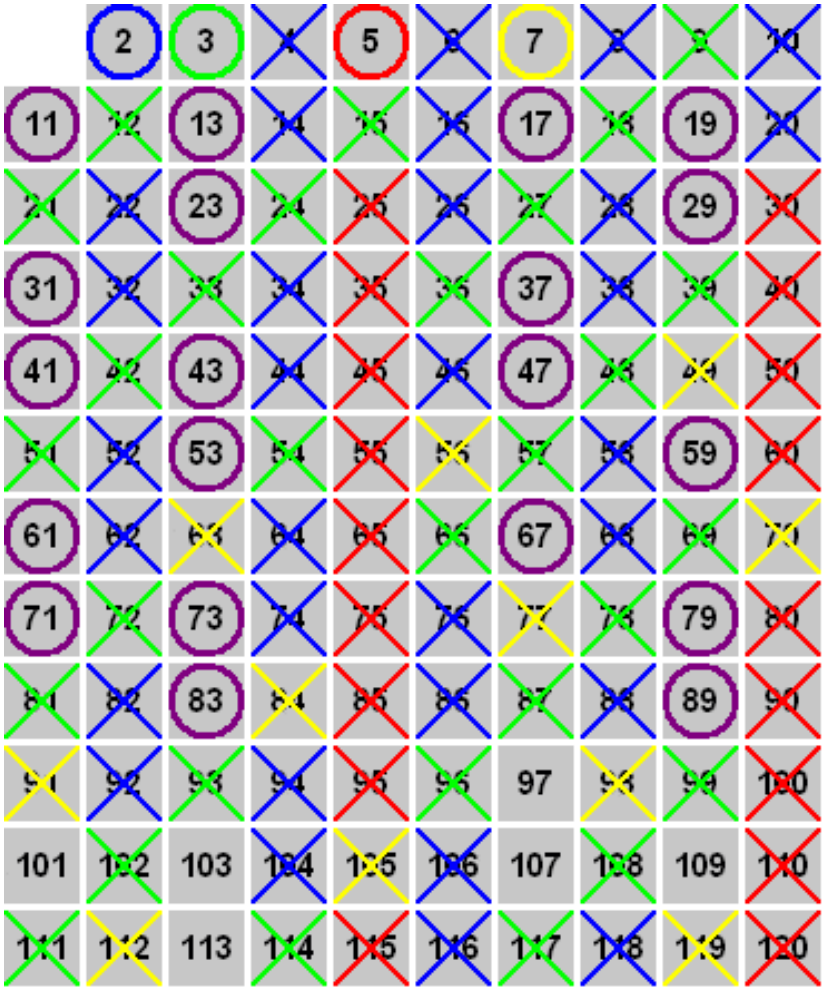
Exemplo: Crivo de Eratóstenes



Prime numbers

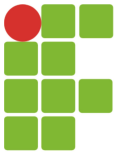
2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	

Exemplo: Crivo de Eratóstenes

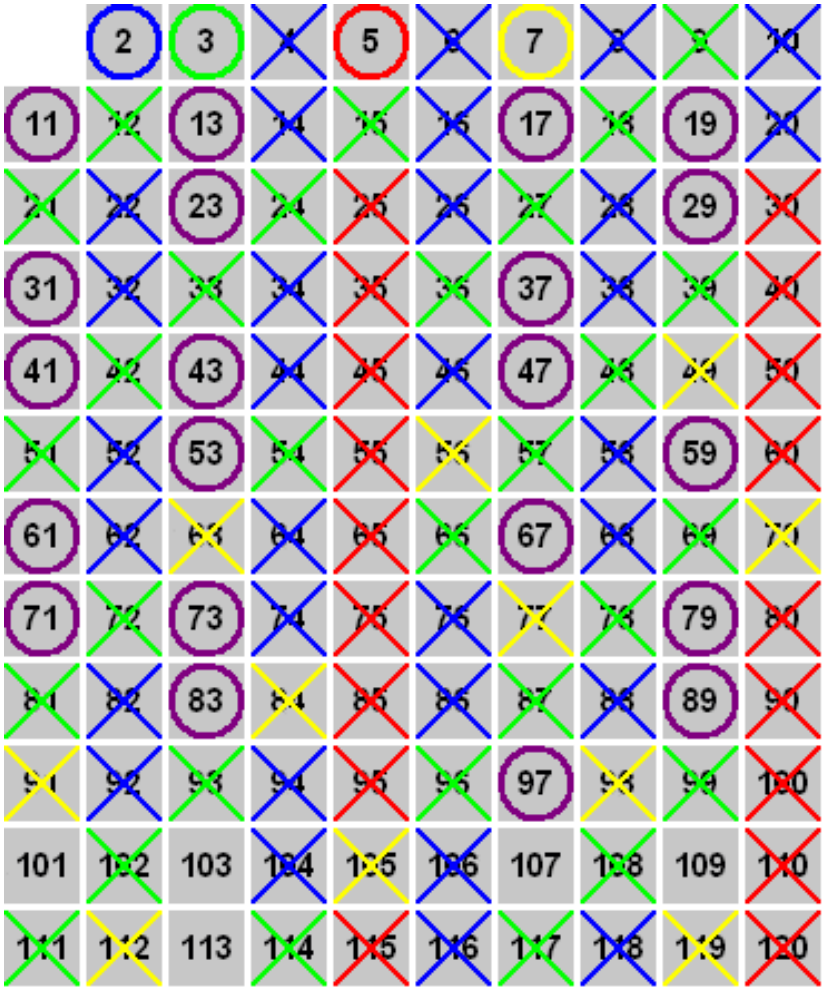


Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89

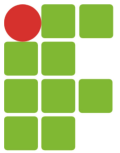


Exemplo: Crivo de Eratóstenes

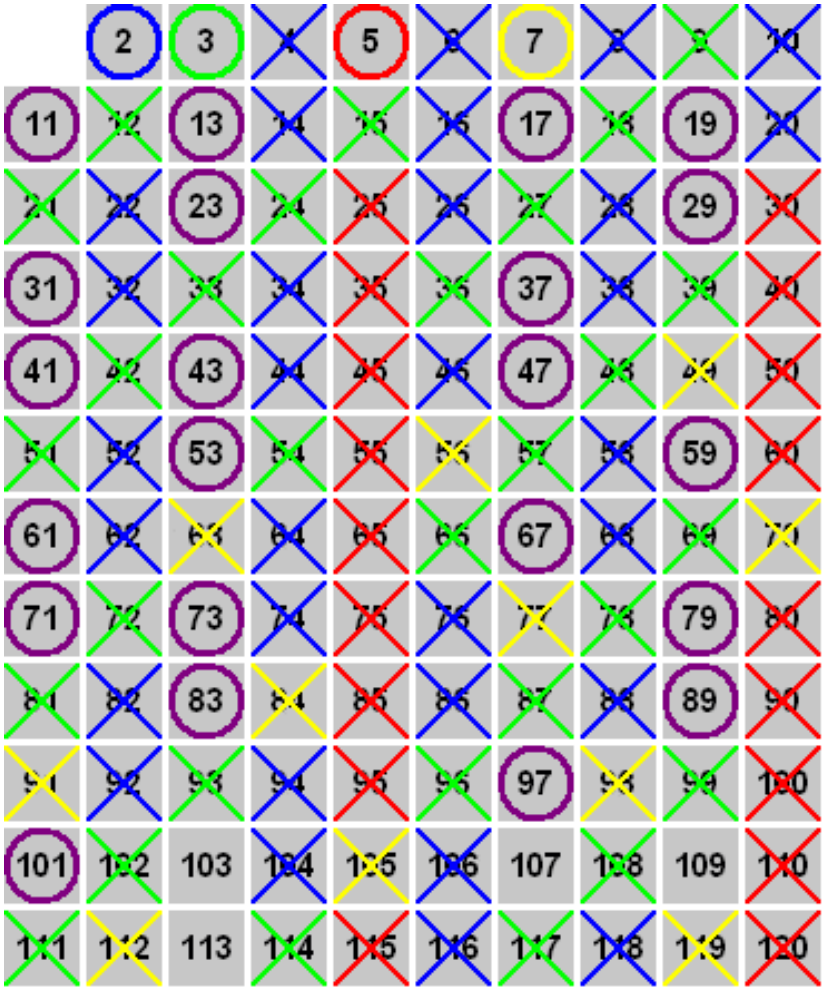


Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97			

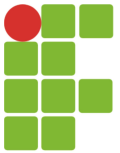


Exemplo: Crivo de Eratóstenes

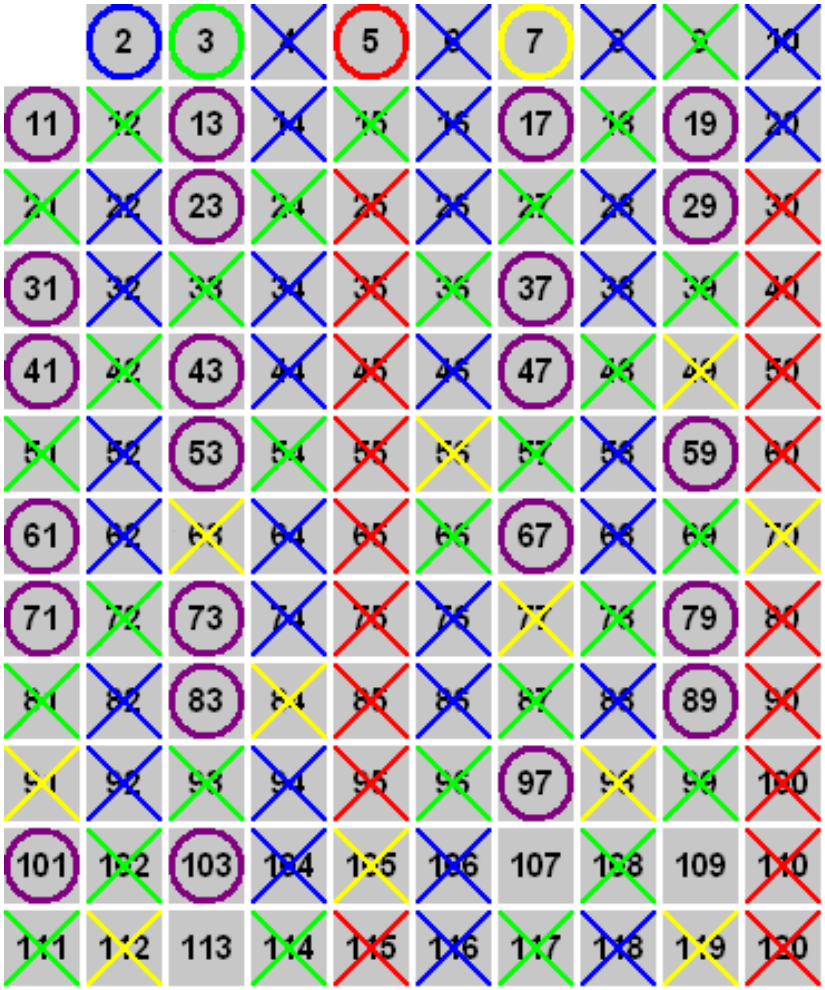


Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97	101		

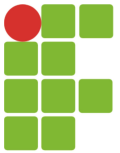


Exemplo: Crivo de Eratóstenes

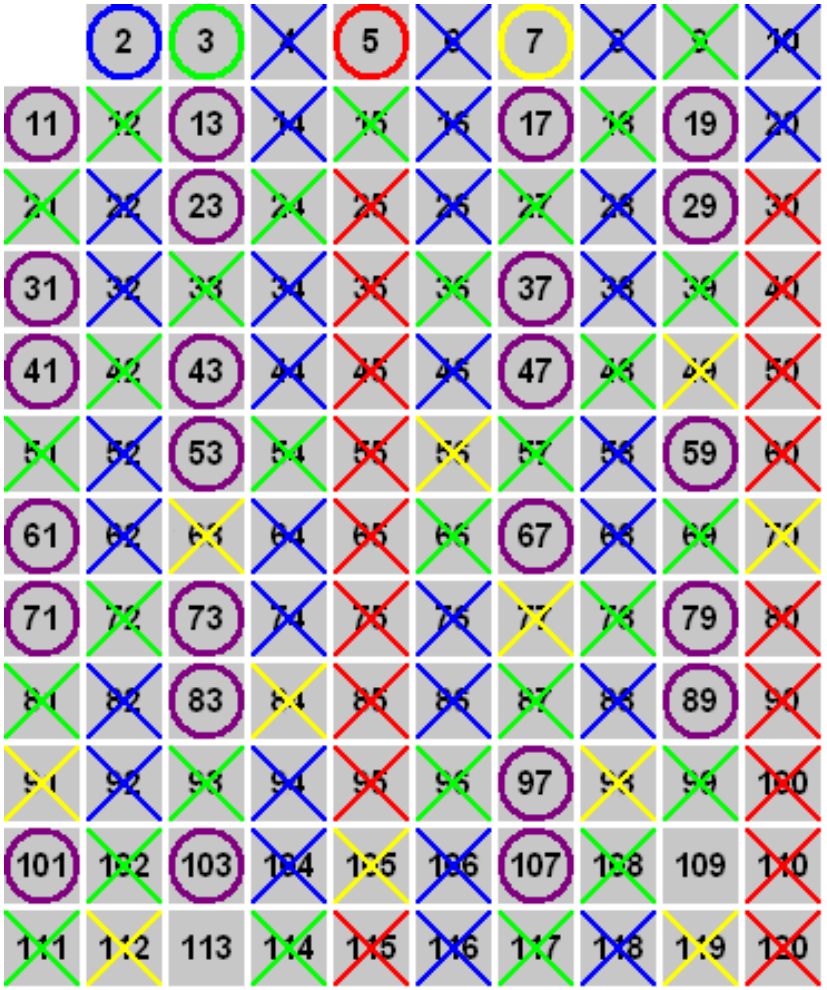


Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97	101	103	

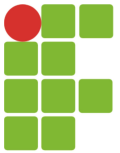


Exemplo: Crivo de Eratóstenes

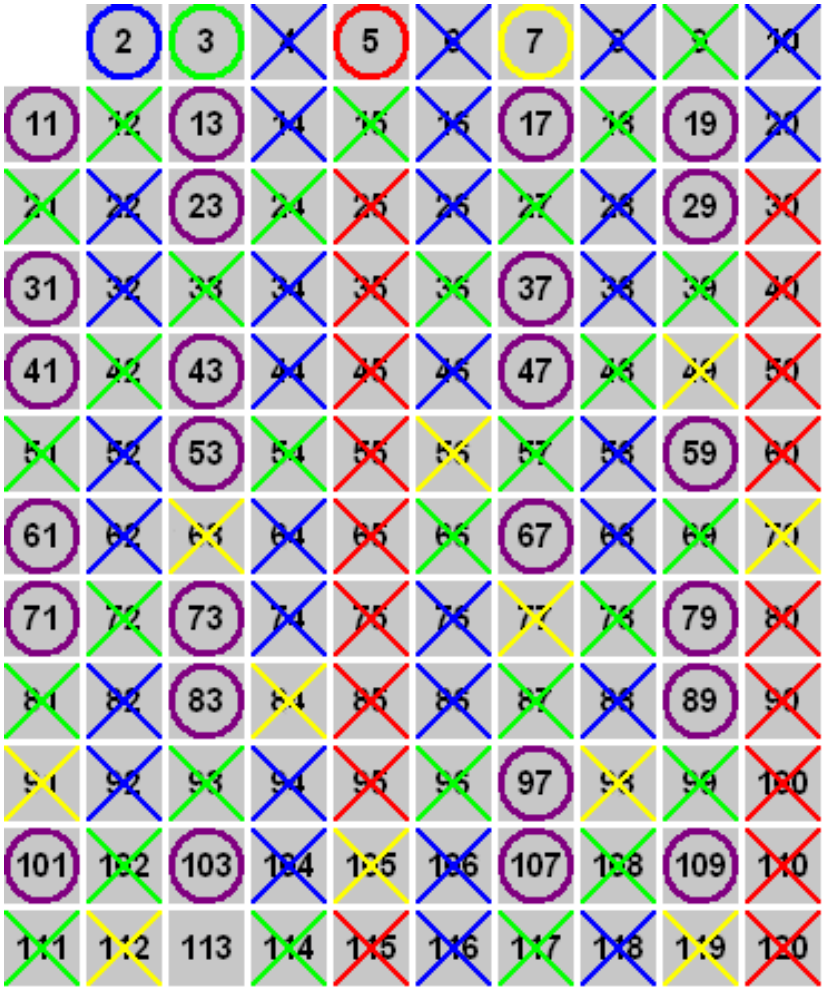


Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97	101	103	107

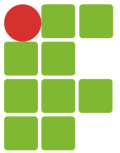


Exemplo: Crivo de Eratóstenes

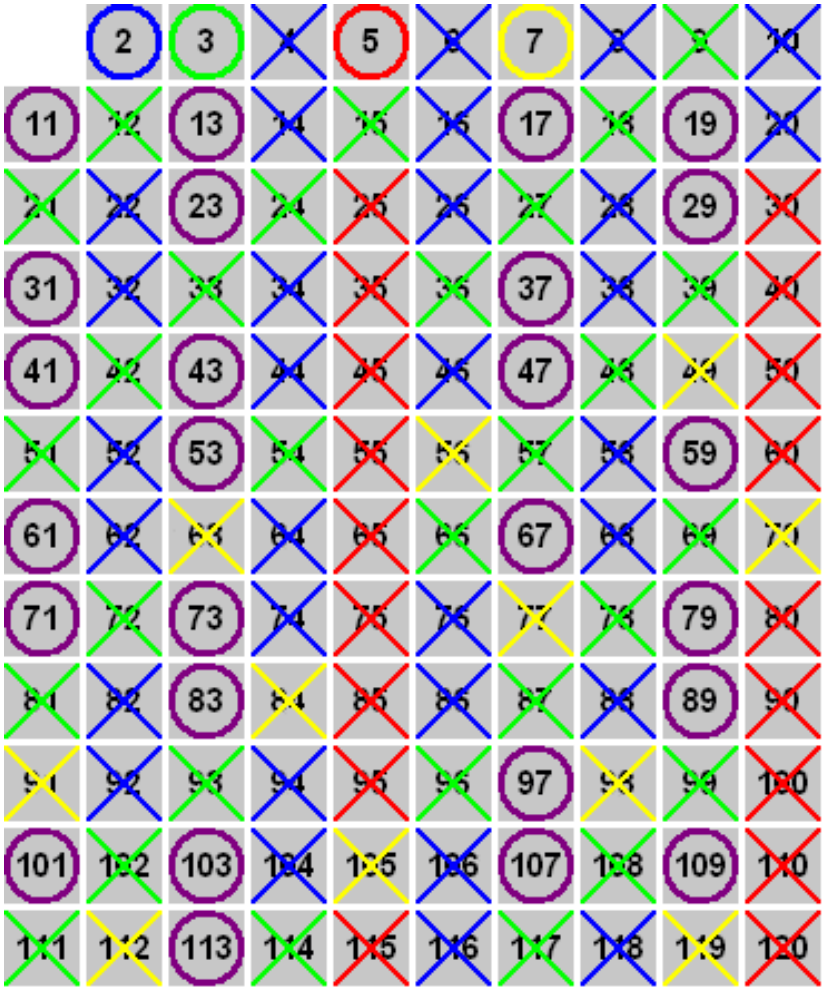


Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97	101	103	107
109			

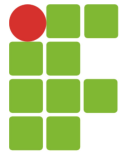


Exemplo: Crivo de Eratóstenes



Prime numbers

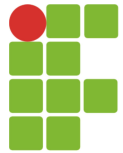
2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97	101	103	107
109	113		



Exemplo: Crivo de Eratóstenes

Problema

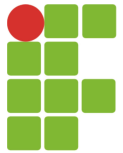
- Dado um inteiro n, imprimir todos os números primos do intervalo $[1, n]$.



Exemplo: Crivo de Eratóstenes

- Antes de tudo, vamos criar uma função que inicialize a nossa lista.
- Nossa lista será um vetor de inteiros `crivo`, e cada entrada i do nosso vetor será 1, indicando que o número ainda não foi eliminado.
- Quando o número for eliminado, basta fazer `crivo[i]=0`.
- Claramente, as entradas 0 e 1 tem que ser falsas, pois não são primos.

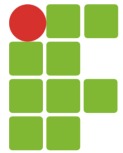
```
void inicializa(int crivo[], int n) {
    int i;
    crivo[0] = 0;
    crivo[1] = 0;
    for (i = 2; i <= n; i++) {
        crivo[i] = 1;
    }
}
```



Exemplo: Crivo de Eratóstenes

- Agora, vamos criar uma função que elimine todos os múltiplos de um determinado número (maiores que este número) até o valor de n.

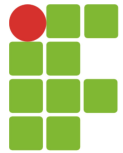
```
void elimina(int lista[],int n,int numero){
    for(int i=numero*2;i<=n;i+=numero){
        lista[i] = 0;
    }
}
```



Exemplo: Crivo de Eratóstenes

- Com posse dessas duas funções, podemos implementar o crivo.

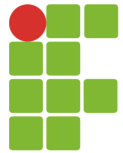
```
void executa_crivo(int n){
    int crivo[n+1];
    inicializa_crivo(crivo,n);
    int i;
    for(i=2;i<=sqrt(n);i++){
        if(crivo[i]){
            elimina(crivo,n,i);
        }
    }
}
```



Exemplo: Crivo de Eratóstenes

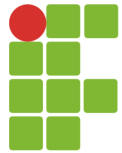
- Por fim, podemos simplesmente varrer a nossa tabela e imprimir todos os números primos.

```
void imprime(int crivo[], int n) {
    int i;
    printf("Números primos de 1 a %d\n", n);
    for (i = 2; i <= n; i++) {
        if (crivo[i]) {
            printf("%d ", i);
        }
    }
    printf("\n");
}
```



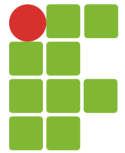
Exemplo: Crivo de Eratóstenes

```
1  #include <math.h>
2  #include <stdio.h>
3
4  void inicializa(int crivo[], int n);
5  void elimina(int lista[], int n, int numero);
6  void imprime(int crivo[], int n);
7  void executa_crivo(int n);
8
9  int main(void) {
10     int n;
11     printf("O programa imprimirá todos os primos até um número n. Digite o "
12            "valor de n: ");
13     scanf("%d", &n);
14     executa_crivo(n);
15     return 0;
16 }
```

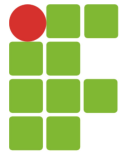


Exemplo: Crivo de Eratóstenes

```
18  void inicializa(int crivo[], int n) {
19      int i;
20      crivo[0] = 0;
21      crivo[1] = 0;
22      for (i = 2; i <= n; i++)
23          crivo[i] = 1;
24  }
25
26  void elimina(int lista[], int n, int numero) {
27      for (int i = numero * 2; i <= n; i += numero)
28          lista[i] = 0;
29  }
30
31  void imprime(int crivo[], int n) {
32      int i;
33      printf("Números primos de 1 a %d\n", n);
34      for (i = 2; i <= n; i++) {
35          if (crivo[i]) {
36              printf("%d ", i);
37          }
38      }
39      printf("\n");
40  }
```



Sumário



Considerações

- Procure sempre modularizar o seu código.
- Organize a suas ideias de modo a quebrar um problema grande em vários problemas menores.
- Você terá um código limpo, organizado e mais fácil de manter.