Técnicas e Análise de Algoritmos

Manipulação de Programas e Introdução ao C++ - Parte 01

Professor: **Jeremias Moreira Gomes**
E-mail: jeremias.gomes@idp.edu.br

Introdução

Juízes Eletrônicos

- **Juízes eletrônicos** são programas que fornecem mecanismos de correção automática para problemas de programação competitiva
- A correção é feita através de testes unitários, e contempla desde a compilação e execução da solução proposta até a validação dos resultados de cada teste unitário
- Uma solução só é considerada correta se passar, de forma bem sucedida, pelo processo de compilação e por todos os testes unitários

Juízes Eletrônicos

- Existem inúmeros juízes eletrônicos no mundo
  - No Brasil, o Beecrowd é o maior e mais conhecido
    - https://www.beecrowd.com.br
    - Contém milhares de problemas para praticar
  - A nível internacional, pode-se citar
    - HackerRank
    - leetCode
    - Codeforces

Juízes Eletrônicos

- Esses sites são bastante utilizados para quem gosta de programação competitiva e como treinamento para **entrevistas técnicas em programação**
- Eles costumam possuir competições próprias, bem como catálogos das principais competições
  - ACM ICPC (*International Collegiate Programming Contest*)
  - Maratona SBC (Sociedade Brasileira de Computação)
  - OBI (Olimpíada Brasileira de Informática)
    - Foco em estudos de programação no ensino médio

Juízes Eletrônicos - Feedback

- No caso do VJudge e outros juízes, cada solução submetida por parte do usuário, retorna um feedback sobre a solução
- Caso a solução esteja correta, a resposta o juiz será Accepted (AC)
- Caso a solução esteja incorreta, será retornada uma dentre várias respostas de erro possíveis, a depender da característica do erro
- **Importante ressaltar que o juiz não informa exatamente qual foi o erro, mas uma categorização possível do erro**
- **Cabe ao estudante/usuário interpretar este retorno e tentar localizar e corrigir o erro antes de sua próxima submissão**

Juízes Eletrônicos - Feedback (1/3)

Código	Erro	Descrição
WA	*Wrong Answer*	Uma ou mais saídas geradas estão incorretas. O juiz não informa as entradas que geraram o erro nem a resposta correta para tais entradas
PE	*Presentation Error*	As saídas do programa estão corretas, mas a apresentação (formatação, espaçamento, etc) está diferente do que foi especificado
CE	*Compilation Error*	O programa não compila corretamente. Em geral, os juízes listam os parâmetros de compilação utilizados na correção

Juízes Eletrônicos - Feedback (2/3)

Código	Erro	Descrição
RE	*Runtime Error*	O programa trava durante a execução, geralmente por conta de falhas de segmentação, divisão por zero, etc
TLE	*Time Limit Exceeded*	Os programas devem gerar as saídas válidas dentro de um limite de tempo especificado. Caso o programa exceda este tempo, esta será a resposta do juiz
MLE	*Memory Limit Exceeded*	O programa requer mais memória em sua execução do que o juiz permite

Juízes Eletrônicos - Feedback (3/3)

Código	Erro	Descrição
RF	*Restricted Functions*	O programa faz uma chamada a uma função considerada ilegal (por exemplo, `fork()` e `fopen()`)
SE	*Submission Error*	O formulário de envio da submissão tem campos vazios ou incorretos
OLE	*Output Limit Exceeded*	O programa tentou imprimir mais informações do que o permitido. Geralmente causado por laços infinitos

Juízes Eletrônicos - Linguagens

- O Codeforces possui um grande conjunto de linguagens aceitas para a resolução dos problemas
- Nesta disciplina, serão permitidas as seguintes:
  - C, C++ (foco da disciplina), Java, Python
- C++ é a linguagem mais utilizada em competições
  - Linguagem compilada
  - Rápida
  - Vasto número de estruturas pela STL (*standard template library*)

Entrada e Saída

Entrada e Saída

- Problemas de programação requerem que as soluções **leiam entradas e escrevam saídas** em arquivos específicos
- Na maioria dos casos, estes arquivos são a entrada (stdin) e a saída (stdout) padrão do sistema (teclado e tela)
- Cada linguagem tem mecanismos para ler a entrada e escrever nestes arquivos

Entrada e Saída

- As entradas se encaixam em quatro categorias:
  a. **Uma única instância do problema**
  b. **T instâncias do problema (o valor de T é dado na primeira linha)**
  c. **N instâncias do problema, a entrada termina com um valor sentinela**
  d. **N instâncias do problema, a entrada termina com fim de arquivo (EOF)**

Entrada e Saída - Categorias de Entradas

- Dados dois inteiros X e Y, determinar a sua soma.

1 - Entrada única

```
3 8
```

2 - T instâncias

```
3
5 7
8 9
9 2
```

3 - Sentinela

```
5 7
8 9
9 2
-1 -1
```

4 - EOF (fim de arquivo)

```
5 7
8 9
9 2
```

Entrada e Saída - Entrada Única

```c
int main()
{
    int X, Y;
    scanf("%d %d", &X, &Y);
    printf("%d\n", X + Y);

    return 0;
}
```

```cpp
int main()
{
    int X, Y;
    cin >> X >> Y;
    cout << X + Y << endl;

    return 0;
}
```

```python
line = input()
X, Y = [int(n) for n in line.split()]

print(X + Y)
```

```java
public class C1 {
    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        int X = scanner.nextInt();
        int Y = scanner.nextInt();
        System.out.println(X + Y);
    }
}
```

Entrada e Saída - T instâncias

```c
int main()
{
    int T;
    scanf("%d", &T);

    for (int i = 0; i < T; i++)
    {
        int X, Y;
        scanf("%d %d", &X, &Y);

        printf("%d\n", X + Y);
    }
    return 0;
}
```

```python
T = int(input())

for _ in range(T):
    line = input()
    X, Y = [int(n) for n in line.split()]

    print(X + Y)
```

Entrada e Saída - Valor(es) Sentinela

```c
int main()
{
    int X, Y;

    while (scanf("%d %d", &X, &Y), X != -1 && Y != -1)
    {
        printf("%d\n", X + Y);
    }

    return 0;
}
```

```python
while True:
    line = input()
    X, Y = [int(n) for n in line.split()]

    if X == -1 and Y == -1:
        break

    print(X + Y)
```

Entrada e Saída - EOF (fim de arquivo)

```c
int main()
{
    int X, Y;

    while (scanf("%d %d", &X, &Y) != EOF)
    {
        printf("%d\n", X + Y);
    }

    return 0;
}
```

```python
while True:
    try:
        line = input()
        X, Y = [int(n) for n in line.split()]

        print(X + Y)
    except EOFError:
        break
```

```java
public class C {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            int X = scanner.nextInt();
            int Y = scanner.nextInt();
            System.out.println(X + Y);
        }
    }
}
```

Entrada e Saída - Teste de Soluções

- De posse de um teste de entrada (entrada.txt) e uma saída esperada (saida.txt) (1/2)
  - Para gerar a resposta da sua solução (arquivo resposta.txt), pode-se fazer o seguinte:

```
# C/C++
$ ./a.out < entrada.txt > resposta.txt

# Java
$ java Main < entrada.txt > resposta.txt

# Python
$ python sol.py < entrada.txt > resposta.txt
```

Entrada e Saída - Teste de Soluções

- De posse de um teste de entrada (entrada.txt) e uma saída esperada (saida.txt) (2/2)
  - Para verificar se a solução proposta está correta, basta usar o comando diff do Linux::

```
$ diff -s gabarito.txt saida.txt
```

C++

Por que C++?

- **Velocidade**
  - Programas são mais rápidos que os equivalentes em Java ou Python
- **Controle da memória**
  - Não há coletor de lixo, o controle da memória é determinístico

Por que C++?

- **Estruturas de dados e algoritmos**
  - STL provê uma série de facilidades, evitando a implementação de estruturas de dados e algoritmos mais conhecidos
- Possibilita o uso de programação genérica (templates)
- **Sintaxe similar a do C**
  - Diminui a curva de aprendizagem para quem já conhece a linguagem C, sem muita abstração

Sobre C++ nesta disciplina

- A linguagem não será estudada em sua totalidade
  - C++ é uma linguagem muito complexa
    - Orientação a objetos
    - Interfaces Gráficas
    - Sistemas Embarcados
    - Jogos
  - O foco ficará em **estruturas de dados, velocidade** e programação competitiva

Compilação

- Os códigos-fonte em C++ utilizam a extensão **`.cpp`**
- C++ é uma linguagem compilada e o compilador mais utilizado é o g++, que faz parte do GNU Compiler Collection (GCC)
- Para compilar, basta utilizar:
  - `g++ codigo.cpp -o nome_programa`
- Caso exista interesse em habilitar suporte para os padrões mais novos da linguagem, como o C++ 14 ou o C++ 17, pode-se utilizar a flag **`-std=c++xx`**, onde xx representa a versão da linguagem
  - `g++ codigo.cpp -std=c++17 -Wall -o programa`

Tipos Primitivos de Dados

- **Variáveis Integrais (inteiro, short e char)**
  - Em C++, inteiros podem ou não ter sinais (`unsigned`)
  - Possuem representações de tamanhos diferentes
  - Escolher o tipo adequado para um problema é essencial

Tipo	Tamanho	Sem Sinal (unsigned)	Com Sinal
char	8 bits	0 a 255	-128 a 127
short	16 bits	0 a 65.535	-32.768 a 32.767
int	32 bits	0 a 4.294.967.295	-2.147.483.648 a 2.147.483.647
long long int	64 bits	0 a 18.446.744.073.709.551.615	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

Tipos Primitivos de Dados

- **Bases Numéricas em Inteiros**
  - C++ (e outras linguagens) suporta a atribuição de inteiros representados em diferentes bases
  - Utiliza-se o prefixo 0x para hexadecimal e 0b para binário

```
int a = 0x41;       // 65
int b = 0b101010;   // 42
```

Tipos Primitivos de Dados

- **Aritimética Estendida**
  - Para armazenar valores que excedem o limite de variáveis do tipo `long long`, é necessário utilizar de **aritmética estendida**
  - C++ não tem suporte nativo à aritmética estendida (até C++17)
  - Uma alternativa é utilizar o Python ou Java

```java
import java.math.BigInteger;
class A
{
    public static void main(String args[]) {
        BigInteger dois = BigInteger.valueOf(2);
        BigInteger p = dois.pow(100);
        System.out.println(p);
    }
}
```

Tipos Primitivos de Dados

- **Ponto Flutuante (1/5)**
  - Em C e C++ há dois tipos float e double para ponto flutuante
  - O tipo float representa valores em ponto flutuante com precisão simples (7 dígitos de precisão)
  - O tipo double representa valores em ponto flutuante com precisão dupla (15 casas dígitos de precisão)
  - O GCC tem suporte para o tipo long double, com 80 bits e precisão superior ao tipo double
    - A precisão extra traz custos de memória e performance

Tipos Primitivos de Dados

- **Ponto Flutuante (2/5)**
  - Em ponto flutuante, nem todo número pode ser representado

```
int main()
{
    float x = 123456789.0;
    double y = 123456789.0;
    cout.precision(7);
    cout << fixed << x << endl;     // 123456792.0000000
    cout << fixed << y << endl;     // 123456789.0000000

    return 0;
}
```

Tipos Primitivos de Dados

- **Ponto Flutuante (3/5)**
  - Diferente dos números reais, a propriedade associativa e comutativa não se aplicam ao ponto flutuante

```c
#define N 100000
int main()
{
    double total_1 = 0, total_2 = 0, vetor[N];
    for (int i = 0; i < N; i++) vetor[i] = (rand() % 1000) / 1000.0;

    for (int i = 0; i < N; i++) {
        total_1 += vetor[i];
        total_2 += vetor[N - i - 1];
    }
    return 0;
}
```

Tipos Primitivos de Dados

- **Ponto Flutuante (4/5)**
  - Comparações entre valores flutuantes podem gerar resultados incorretos por conta de erros de precisão

```
int main()
{
    if (0.3f * 2 == 0.6) {
        cout << "iguais" << endl;
    } else {
        cout << "diferentes" << endl;
    }
    return 0;
}
```

Tipos Primitivos de Dados

- **Ponto Flutuante (4/5)**
  - Comparações entre valores flutuantes podem gerar resultados incorretos por conta de erros de precisão

```
int main()
{
    if (0.3f * 2 == 0.6) {
        cout << "iguais" << endl;
    } else {
        cout << "diferentes" << endl;
    }
    return 0;
}
```

```
int main()
{
    // precisão de 6 casas decimais
    if (fabs(0.3f * 2 - 0.6) < 0.0000001) {
        cout << "iguais" << endl;
    } else {
        cout << "diferentes" << endl;
    }
    return 0;
}
```

Tipos Primitivos de Dados

- **Ponto Flutuante (5/5)**
  - Na prática, o ideal é evitar utilizar ponto flutuante
    - Exemplo
      - Em unidades monetárias, trabalhar com múltiplos de centavos, de modos que todos os valores sejam inteiros

Tipos Primitivos de Dados

- **Booleano**
  - Nativamente o C++ suporta o tipo bool , que pode assumir dois valores:
    - **`true` -** equivalente ao valor numérico 1
    - **`false` -** equivalente ao valor numérico 0

Referências

Referências

- C++ possui suporte às referências, que são um apelido para outra variável
- Referências devem ser inicializadas no momento da sua declaração, não podendo ter o seu valor atualizado
- O símbolo **&** é utilizado após o nome do tipo para indicar que a declaração se trata de uma referência para aquele tipo

Referências

```
int main()
{
    int a = 42;
    int &b = a;

    cout << "a = " << a << endl;
    b = 1337;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}
```

Referências

- Referências são menos poderosas que ponteiros, já que não podem ser alteradas
  - C++ também possui ponteiros
- Referências simplificam a escrita do código e evitam sintaxes desnecessárias
  - Exemplos
    - Funções
    - Iteradores sem cópia
    - etc

Funções

Funções

- A linguagem C possibilita apenas a passagem por valor, isto é, uma cópia do valor da variável passada por parâmetro é realizada.
- Em C++ existem dois métodos de passagem de parâmetros:
  - **Passagem por valor**
  - **Passagem por referência**

Funções - Passagem por Referência

- Na passagem por referência, uma referência à variável ou objeto é utilizada, fazendo com que qualquer alteração reflita no original
- Além disto, como uma referência é utilizada, e não é realizada uma cópia, pode-se obter algum ganho de desempenho quando objetos grandes são passados para funções
  - Exemplo: objetos do tipo vector

Funções - Passagem por Referência

- Para utilizar a passagem por referência, basta utilizar o **&** após o tipo do parâmetro
  - Caso o & não seja utilizado, a passagem é por valor (cópia)
- É claro que a escolha do método de passagem de parâmetros vai depender da função e aplicação

Funções - Passagem por Referência

```
void altera(int &valor)
{
    valor = 1337;
}

int main()
{
    int a = 42;
    altera(a);
    cout << a << endl;
    return 0;
}
```

Funções - Passagem por Referência

- Quando utilizar?
  - Para modificar a variável original, passada para a função
  - Para passar variáveis grandes: realizar uma cópia é muito mais custoso do que utilizar uma referência

Bibliotecas Padrão (em C++)

Bibliotecas Padrão

- Assim como em C, C++ possui algumas bibliotecas padrão
  - **<iostream>** - Fluxos de entrada e saída
  - **<iomanip>** - Formatação de saída (ex: precisão de float)
  - **<string>** - Manipulação de cadeias de caracteres
  - **<algorithm>** - Implementação de algoritmos mais conhecidos
  - **<cstdint>** - Adequação da `<stdint.h>` para tipos inteiros
  - **<cstdio>** - Adequação da `<stdio.h>` para C++
  - **<cmath>** - adequação da `<math.h>` para C++
  - etc

Bibliotecas Padrão

- Mais voltado para competições de programação e testes, existe um cabeçalho que concentra todos esses principais citados:
  - `<bits/stdc++.h>`
- Observação:
  - Apenas tomar cuidado, porque a inclusão desse cabeçalho inclui muitas funções que acabam não sendo utilizadas, então ambientes mais formais ou profissionais optam por incluir apenas o necessário para o que se está sendo resolvido

namespace

namespace

- Um espaço de nomes (*namespace*) fornecem um método para evitar conflitos de nomes em projetos grandes
- Os identificadores declarados dentro de um bloco de namespace são colocados em um escopo que evita que sejam confundidos com símbolos com nomes idênticos em outros escopos
- Utilizado para organizar códigos, evitando colisão de nomes
- Identificadores pertencentes a um espaço de nomes devem ser utilizados com a sintaxe **`nome_do_espaco::identificador`**

namespace

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, IDP!" << std::endl;
    return 0;
}
```

```cpp
#include <iostream>
namespace computacao {
    void nome_curso() {
        std::cout << "Computação!" << std::endl;
    }
}
namespace arquitetura {
    void nome_curso() {
        std::cout << "Arquitetura!" << std::endl;
    }
}
int main()
{
    computacao::nome_curso();
    arquitetura::nome_curso();
    return 0;
}
```

namespace

- C++ possui uma diretiva de uso de espaços de nomes chamada **`using`** que serve para introduzir um nome que foi definido em outro lugar no escopo onde esta declaração aparece aparece

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, IDP!" << std::endl;
    {
        using std::cout;
        using std::endl;
        cout << "Hello, IDP!" << endl;
    }
    return 0;
}
```

```cpp
#include <iostream>

using namespace std;    // Trás o conteúdo do
                        // namespace std para
                        // o escopo global
int main()
{
    cout << "Hello, IDP!" << endl;
    return 0;
}
```

Entrada e Saída (em C++ agora)

Entrada e Saída

- O C++ possui suporte às funções **`scanf`** e **`printf`** legadas do C, pertencentes ao cabeçalho **`cstdio`**
- Além disso, existe a possibilidade de utilizar os mecanismos de entrada e saída próprios do C++
  - Estes mecanismos estão descritos no cabeçalho **`iostream`**
- Por meio dos dos objetos **`cin`** e **`cout`** podemos ler dados da entrada padrão e imprimir dados na saída padrão

`cin`

- `cin` é chamado stream padrão para leitura dos dados
  - Possibilita ler dados da entrada padrão e armazená-los em variáveis
- Por meio do operador **>>** é possível realizar diversas leituras com uma única linha de código

cin

```cpp
#include <iostream>

int main()
{
    int i;
    float f;
    double d;
    char c;
    std::cin >> i >> f >> d >> c;
    return 0;
}
```

`cout`

- `cout` é o chamado stream padrão para escrita dos dados
  - Possibilita imprimir o conteúdo das variáveis na saída padrão
- Por meio do operador `<<` é possível realizar diversas escritas com uma única linha de código

cout

```cpp
#include <iostream>

int main()
{
    int i = 8;
    float f = 3.14;
    double d = 6.28;
    char c = 'K';
    std::cout << i << " " << f << " " << d << " " << c << std::endl;
    return 0;
}
```

`cin/cout` mais rápidos

- C++ suporta tanto as funções do C como os streams `cin` e `cout`
  - É possível utilizar ambos os mecanismos simultaneamente
  - Para manter a compatibilidade de uso dos dois mecanismos ao mesmo tempo, há uma inclusão de overhead na entrada e saída para manter essa sincronia
- Ao desabilitar a sincronia entre os dois mecanismos, `cin` e `cout` tornam-se bem mais eficientes
  - Ao desabilitar a sincronia, não será mais possível misturar métodos de I/O das duas linguagens

`cin/cout` mais rápidos

- Para desabilitar a sincronia, basta adicionar a linha **`std::ios::sync_with_stdio(false);`** no início do programa

```cpp
#include <iostream>

using namespace std;
int main()
{
    ios_base::sync_with_stdio(false);

    // Não há garantia da ordem de saída
    cout << "a\n";
    printf("b\n");
    cout << "c\n";

    return 0;
}
```

Operações bit a bit

Operações bit a bit

- Operações que manipulam bits individuais de números
- Vantagens:
  - Eficiência
  - Compactação de dados
- Uso:
  - Controle de hardware
  - Programação de baixo nível
  - Criptografia
  - etc

Operações bit a bit

- Operações bit a bit operam em inteiros

OPERAÇÃO	SIGNIFICADO
~x	Representação binária complementar de x
x & y	Operação de E bit a bit entre x e y
x | y	Operação de OU bit a bit entre x e y
x ^ y	Operação de XOR bit a bit entre x e y
x << 1	Operação de deslocamento à esquerda (shift left) de i posições
x >> 1	Operação de deslocamento à direita (shift right) de i posições

~x - Operação de Complemento

- O operador ~ realiza o complemento de um da representação binária de um inteiro

```
uint8_t x = 170; // x = 0b1010’1010
uint8_t y = ~x;  // y = 0b0101’0101 (85)
```

~x - Operação de Complemento

- O operador ~ realiza o complemento de um da representação binária de um inteiro

```
#include <bits/stdc++.h>
int main()
{
    uint8_t x = 170; // x = 0b1010’1010
    uint8_t y = ~x;  // y = 0b0101’0101 (85)
    printf("%hhu - %hhu\n", x, y);
    return 0;
}
```

~x - Operação de Complemento

- No caso de um inteiro (com sinal), o operador **~** equivale ao **complemento de 1** da representação binária

```
int8_t x = 85;  // x = 0b0101’0101
int8_t y = ~x;  // y = 0b1010’1010 (-86)
```

~x - Operação de Complemento

- No caso de um inteiro (com sinal), o operador **~** equivale ao **complemento de 1** da representação binária

```
#include <bits/stdc++.h>
int main()
{
    int8_t x = 85;  // x = 0b0101’0101
    int8_t y = ~x;  // y = 0b1010’1010 (-86)
    printf("%hhd - %hhd\n", x, y);
    return 0;
}
```

Complemento de 2

- Em C++, os números negativos são representados via **complemento de dois**
- O complemento de dois é obtido a partir do complemento de um somado de uma unidade
- Propriedade: **`-x == ~x+1`**

E, OU e XOR

```
#include <bits/stdc++.h>

int main()
{

    int8_t x = 0b10101010; // 170
    int8_t y = 0b00001111; // 15

    printf("%hhd\n", x & y);    // 0000'1010 (10)
    printf("%hhd\n", x | y);    // 1010'1111 (-81)
    printf("%hhd\n", x ^ y);    // 1010'0101 (-91)
    return 0;
}
```

Deslocamento à Esquerda (<<)

- Seja o inteiro x composto pelos bits $x_0 x_1 \ldots x_{n-1}$ e i um inteiro
- O operador << desloca todos os bits de x, i posições para a esquerda
- A cada deslocamento, temos que

$$x_0 \leftarrow x_1, x_1 \leftarrow x_2, \ldots, x_{n-1} \leftarrow 0$$

```
uint8_t x = 0b0100'1111;    // x = 79
uint8_t y = x << 1;         // y = 0b1001'1110 (158)
uint8_t z = x << 2;         // z = 0b0011'1100 (60)
uint8_t w = x << 3;         // w = 0b0111'1000 (120)
```

Deslocamento à Direita (>>)

- Seja o inteiro x composto pelos bits $x_0 x_1 \ldots x_{n-1}$ e i um inteiro
- O operador >> desloca todos os bits de x, i posições para a direita
- A cada deslocamento, temos que

$$x_0 \leftarrow 0, x_1 \leftarrow x_0, \ldots, x_{n-1} \leftarrow x_{n-2}$$

```
uint8_t x = 0b0100'1111;    // x = 79
uint8_t y = x >> 1;         // y = 0b0010'0111 (39)
uint8_t z = x >> 2;         // z = 0b0001'0011 (19)
uint8_t w = x >> 3;         // w = 0b0000'1001 (9)
```

Ativação do i-ésimo bit

- Para ligar o i-ésimo bit menos significativo de um inteiro x, basta fazer: **`x = x | (1 << i)`**
  - É o equivalente a **`x |= (1 << i)`**
  - **`1 << i`** é exatamente o número em que todos os bits são zeros, exceto aquele que ocupa a posição i, então, ao aplicar a operação OU, a posição i passar a ser 1

```
uint8_t x = 0b1000'0001;    // x = 129
x |= (1 << 4);              // 1000'0001 | 0001'0000 = 1001'0001
                            // 129 | 16 = 145
```

Desativação do i-ésimo bit

- Para desligar o i-ésimo bit menos significativo de um inteiro x, basta fazer: **`x = x & ~(1 << i)`**
  - É o equivalente a **`x &= ~(1 << i)`**
  - **`~(1 << i)`** é exatamente o número em que todos os bits são uns, exceto aquele que ocupa a posição i, então, ao aplicar a operação E, a posição i passar a ser 0

```
uint8_t x = 0b1111'1111;    // x = 129
x &= ~(1 << 4);             // 1111'1111 & 1110'1111 = 1110'1111
                            // 255 & 239 = 239
```

Bithacks

https://graphics.stanford.edu/~seander/bithacks.html

Conclusão