# Tópicos Avançados em Algoritmos

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

31 de março de 2019

## *Big Nums*
Representação de inteiros realmente grandes

## Limite computacional

- A representação numérica de inteiros no computador está relacionada com o barramento da CPU e às suas instruções.
- O inteiro ocupa uma "palavra" do barramento.
- Para CPU de 64 bits, o inteiro é de 64 bits, para CPU de 32 bits, o inteiro é de 32 bits.
- O uso de long int está previsto para 2 "palavras", neste caso, para CPU de 32 bits, o long int tem 64 bits, e para CPU de 64 bits, o long int tem 128 bits.
- O long long int possui 128 bits.

## Limite Computacional

- Dentro isto, temos um limite computacional para valores inteiros. Operações que geram resultados além deste limite irão gerar o flag de "overflow" e descartará os bits mais significativos do resultado.
- Tabela dos limites, e constantes definidas no C/C++ (limits.h)

| **Tipo (64bits)** | **Constante(s)** | **valor** |
|---|---|---|
| signed int (min) | `INT_MIN` | -2.147.483.648 |
| signed int (max) | `INT_MAX` | 2.147.483.647 |
| unsigned int (max) | `UINT_MAX` | 4.294.967.295 |
| signed long int (min) | `LONG_MIN` | -9.223.372.036.854.775.808 |
| signed long int (max) | `LONG_MAX` | 9.223.372.036.854.775.807 |
| unsigend long int (max) | `ULONG_MAX` | 18.446.744.073.709.551.615 |
| signed long long int (min) | `LLONG_MIN` | -9.223.372.036.854.775.808 |
| signed long long int (max) | `LLONG_MAX` | 9.223.372.036.854.775.807 |
| unsigend long long int (max) | `ULLONG_MAX` | 18.446.744.073.709.551.615 |

## Bignum

- *Bignums* representam números inteiros além da capacidade de representação numérica no computador.
- Tipicamente números $> 10^{19}$.
- Os (des)afortunados do Java possuem uma biblioteca que se dedica a fazer operações com bignums, é a `BigInteger`.
- Infelizmente o STL não traz uma classe que represente bignum no C++, logo teremos de fazer a nossa.
- É uma boa experiência para lidar, também, com operadores.

# Bignum

- Representação:
  - Podemos fazer algo mais complexo, como uma representação dependente da base, mas a base precisa ser múltipla de 10 (para gerar strings da representação diretamente para cada dígito da base)..
  - Um número representado no bignum é representado por: bool sign, e vector¡int¿ digits:
  - o primeiro elemento representa o sinal, `false` é um número negativo e `true` um número positivo (zero usa true).
  - o segundo elemento é um vetor inteiros (cada inteiro é um dígito da base), se for base 10, os dígitos são 0 a 9. na base 100, os dígitos são 00 a 99.
  - O dígito menos significativo ocupa a posição 0 do vetor, por exemplo, na base 100 o número 9475928 é representado como: `{ 28, 59, 47, 9 }`
  - Os zeros no final do vetor podem ser eliminados pois são zeros à esquerda do número.

# Bignum

- Para os números, precisaremos realizar as operações:
  - Adição: dígito a dígito, somamos, se passar da base, vai um a ser somado no próximo dígito significativo, e descontamos a base para guardar este dígito.
  - Subtração: Se os dois têm o mesmo sinal, fazemos a subtração do maior para o menor, e damos o sinal do maior. Se tiverem sinal diferente, mudamos o sinal do que será subtraído e fazemos a soma, o resultado tem o sinal de um deles.
  - Multiplicação: Dados dois números $A$ e $B$, representados pelos dígitos $A_i$ e $B_j$, o produto $C$, terá o seu dígito $C_k = (vai\_um) + \sum_{i+j=k} (A_i \times B_j)$, carregando o "vai_um" para o próximo dígito $C$.
  - Divisão e Resto: O procedimento é o algoritmo que conhecemos desde criança, enquanto houver divisor maior que o dividendo (com o mesmo número de casas), vemos quantos inteiros existem, este é o primeiro valor do quociente. Pegamos o próximo dígito e continuamos.

## class bignum

```
#include <utility>
#include <string>
#include <vector>

#ifndef __BIG_NUM__
#define __BIG_NUM__
#define numbase 10 // base múltiplo de 10, menor que 10⁹
class bignum{
public:
  bool sign;
  std::vector<int> num;

  bignum();
  bignum(const bignum &n);
  bignum(const std::string &n);
  bignum(bool s, const std::vector<int> &n);
  bignum(int n); { bignum((long) n); }
  bignum(long n);
  bignum(unsigned long n);
  std::string tostring();
private:
  void ulongtobn(unsigned long n);
};
```

## constantes e operadores

```
// CONSTANTS DEFINED FOR BIGNUM

const bignum um(1);
const bignum zero(0);

// RELATIONAL OPERATORS, com include de utility, basta ‘‘==’’ e ‘‘<’’

bool operator==(const bignum &a, const bignum &b);
bool operator<(const bignum &a, const bignum &b);

// ARITMETIC OPERATORS

bignum operator+(const bignum &a, const bignum &b);
bignum operator-(const bignum &a, const bignum &b);
bignum operator*(const bignum &a, const bignum &b);
bignum operator/(const bignum &a, const bignum &b);
bignum operator%(const bignum &a, const bignum &b);

#endif // __BIG__NUM__
```

## Implementação: construtores

```
#include <algorithm>
#include <utility>
#include <vector>
#include <climits>
#include "bignum.hpp"

using namespace std;

bignum::bignum():  sign(true), digits(vector<int>({0})) { }

bignum::bignum(const bignum &n) {
  sign = n.sign;
  digits = vector<int>(n.digits);
} bignum::bignum(bool s, const vector<int> &n):  sign(s),
digits(vector<int>(n)) { }
```

## Implementação: construtores

```
bignum::bignum(long n) {
  sign=true;
  unsigned long u;
  if(n < 0) {
    sign = false;
    if(n == LONG_MIN) u = (unsigned long) LONG_MAX + 1;
    else u = -n;
  }
  ulongtobn(u);
}
bignum::bignum(unsigned long n) {
  sign = true;
  ulongtobn(n);
}

void bignum::ulongtobn(unsigned long n) {
  if(n==0) digits.push_back(0);
  else while(n) {
    digits.push_back((char)(n%numbase));
    n/=numbase;
  }
  return;
}
```

## Implementação: construtores

```
bignum::bignum(const string &n) {
  sign = n[0]!= '-';
  auto p = n.rbegin();
  while(p!=n.rend() && *p != '-') {
    int b=1, aux=0;
    while(b != numbase && p!=n.rend() && *p != '-') {
      aux += (*p-'0')*b;
      b*=10;
      p++;
    }
    digits.push_back(aux);
  }
}
string bignum::toString() {
  string s;
  if(!sign) s.push_back('-');
  auto pa = digits.rbegin();
  while(pa != digits.rend()) {
    s+=to_string(*pa);
    pa++;
  }
  return s;
}
```