Aritmética e Álgebra

André Gustavo dos Santos

Departamento de Informática
Universidade Federal de Viçosa

INF 492 - 2012/1

Computação x Matemática

Aliás, computação + matemática

Aritmética em Computadores

Inteiros

- Toda linguagem de programação tem um tipo inteiro, que suporta as quatro operações aritméticas básicas: adição, subtração, multiplicação e divisão
- Estas operações são tipicamente implementadas em hardware, então o "tamanho" do inteiro depende do processador

Positivos x Negativos

- Inteiros positivos são guardados como números binários (positivos)
- Inteiros negativos com a representação complemento-de-2, que embora meio obscura facilita a aritmética em hardware

Tipo `int`

O tamanho típico do `int` é 32 bits, que suporta então valores de:

- -2^{31} a $2^{31} - 1$
- ou seja, -2.147.483.648 a 2.147.483.647

o que significa que você pode contar até 1 bilhão sem problemas

Tipo `unsigned int`

O bit de sinal também faz parte do valor, "dobrando" o limite disponível

- 0 a $2^{32} - 1$
- ou seja, 0 a 4.294.967.295

Aritmética em Computadores

Tipo `long int` e `long long int`

Depende da máquina, mas o `long long int` geralmente é de 64 bits:

- -2^{63} a $2^{63} - 1$
- ou seja, -9.223.372.036.854.775.808 a 9.223.372.036.854.775.808

ou seja, pra lá de 1 quintilhão...

Tipo `short int` e `char`

Inteiros de 32-bits gastam 4 bytes, e os de 64-bits, 8 bytes.
Para armazenar grande quantidade de números, cada um deles não tão grande, é conveniente usar 1 byte por número

- `char`: -128 a 127
- `unsigned char`: 0 a 255

por exemplo, imagens podem ser representadas por matrizes de bytes (256 cores ou tons de cinza) por questão de espaço

E se precisar mais que isso?

Tipos de ponto flutuante (ex. `float`), podem armazenar valores imensos, principalmente os de precisão dupla (`double`, `long double`), mas...

- São armazenados na forma $a \times 2^c$
- Tanto a quanto c são inteiros, logo com precisão limitada

Solução...

Implementar um tipo inteiro utilizando um *array de dígitos*

Observações:

- Para usar realmente uma precisão arbitrária, usar array dinâmico
- A linguagem Java tem a classe `java.math BigInteger`

Inteiros de alta precisão

Código proposto no livro Programming Challenges, disponível on-line em
`http://www.cs.sunysb.edu/~skiena/392/programs/bignum.c`

```
#define MAXDIGITS 100   /* maximum length bignum */

#define PLUS     1      /* positive sign bit */
#define MINUS   -1      /* negative sign bit */

typedef struct {
   char digits[MAXDIGITS];    /* represent the number */
   int signbit;               /* 1 if positive, -1 if negative */
   int lastdigit;             /* index of high-order digit */
} bignum;
```

```
print_bignum(bignum *n)
{
  int i;

  if (n->signbit == MINUS) printf("- ");
  for (i=n->lastdigit; i>=0; i--)
    printf("%c",'0'+ n->digits[i]);

  printf("\n");
}
```

Inteiros de alta precisão

```
int_to_bignum(int s, bignum *n)
{
  int i; /* counter */
  int t; /* int to work with */

  if (s >= 0) n->signbit = PLUS;
  else n->signbit = MINUS;

  for (i=0; i<MAXDIGITS; i++) n->digits[i] = (char) 0;

  n->lastdigit = -1;
  t = abs(s);

  while (t > 0) {
    n->lastdigit ++;
    n->digits[ n->lastdigit ] = (t % 10);
    t = t / 10;
  }

  if (s == 0) n->lastdigit = 0;
}
```

Para facilitar, todos os `bignum` são inicializados com 0's à esquerda

```
initialize_bignum(bignum *n)
{
  int_to_bignum(0,n);
}
```

Aritmética de alta precisão - adição

Adição

- Adicionar os dígitos da direita pra esquerda, passando o *overflow* para o dígito seguinte como “vai-um”
- Isso é facilitado por termos colocado 0’s à esquerda em todo `bignum`
- Adicionar número negativo pode ser convertido numa subtração (trocando o sinal)

Aritmética de alta precisão - adição

```
add_bignum(bignum *a, bignum *b, bignum *c)
{
  int carry;      /* carry digit */
  int i;          /* counter */

  initialize_bignum(c);

  /* Se os sinais sao diferentes, faz subtracao */

  if (a->signbit != b->signbit) {
     if (a->signbit == MINUS) {
       a->signbit = PLUS;
       subtract_bignum(b,a,c);
       a->signbit = MINUS;
     }
     else {
       b->signbit = PLUS;
       subtract_bignum(a,b,c);
       b->signbit = MINUS;
     }
     return;
  }
  ...
```

Aritmética de alta precisão - adição

```
  ...

  /* Se os sinais sao iguais, faz a adicao digito a digito*/

  c->signbit = a->signbit;
  c->lastdigit = max(a->lastdigit,b->lastdigit)+1;
  carry = 0;

  for (i=0; i<=(c->lastdigit); i++) {
    c->digits[i] = (char) (carry+a->digits[i]+b->digits[i]) % 10;
    carry = (carry + a->digits[i] + b->digits[i]) / 10;
  }

  zero_justify(c);
}
```

Zeros à esquerda

- Todo `bignum` é preenchido com zeros à esquerda, mas é importante ajustar o valor `lastdigit` após cada operação, para evitar considerar esses zeros como parte do número
- Isso será feito após todas as operações, inclusive para corrigir -0

Aritmética de alta precisão - zeros à esquerda

```
zero_justify(bignum *n)
{
  while ((n->lastdigit > 0) && (n->digits[ n->lastdigit ] == 0))
    n->lastdigit --;

  if ((n->lastdigit == 0) && (n->digits[0] == 0))
    n->signbit = PLUS;      /* hack to avoid -0 */
}
```

Aritmética de alta precisão - subtração

Subtração

- É um pouco mais complicada que a adição pois requer “pegar emprestado”
- Para ter certeza que tal “pegar emprestado” termina, o maior número deve ficar “em cima”

Aritmética de alta precisão - subtração

```
subtract_bignum(bignum *a, bignum *b, bignum *c)
{
  int borrow;    /* has anything been borrowed? */
  int v;         /* placeholder digit */
  int i;         /* counter */

  initialize_bignum(c);

  /* Se os sinais sao diferentes, faz adicao */
  if ((a->signbit == MINUS) || (b->signbit == MINUS)) {
    b->signbit = -1 * b->signbit;
    add_bignum(a,b,c);
    b->signbit = -1 * b->signbit;
    return;
  }

  /* Se o segundo e' maior, troca os operandos, resultado negativo  */
  if (compare_bignum(a,b) == PLUS) {
    subtract_bignum(b,a,c);
    c->signbit = MINUS;
    return;
  }
  ...
```

Aritmética de alta precisão - subtração

```
  ...

  /* realiza a subtracao, digito a digito */

  c->lastdigit = max(a->lastdigit,b->lastdigit);
  borrow = 0;

  for (i=0; i<=(c->lastdigit); i++) {
    v = (a->digits[i] - borrow - b->digits[i]);
    if (a->digits[i] > 0)
      borrow = 0;
    if (v < 0) {
      v = v + 10;
      borrow = 1;
    }

    c->digits[i] = (char) v % 10;
  }

  zero_justify(c);
}
```

Aritmética de alta precisão - comparação

Comparação

- Usada para decidir qual `bignum` é maior
- Compara os dígitos no sentido da esquerda (de maior ordem) para direita, começando pelo sinal

Aritmética de alta precisão - comparação

```
compare_bignum(bignum *a, bignum *b)
{
  int i; /* counter */

  if ((a->signbit == MINUS) && (b->signbit == PLUS)) return(PLUS);
  if ((a->signbit == PLUS) && (b->signbit == MINUS)) return(MINUS);

  if (b->lastdigit > a->lastdigit) return (PLUS * a->signbit);
  if (a->lastdigit > b->lastdigit) return (MINUS * a->signbit);

  for (i = a->lastdigit; i>=0; i--) {
    if (a->digits[i] > b->digits[i]) return(MINUS * a->signbit);
    if (b->digits[i] > a->digits[i]) return(PLUS * a->signbit);
  }

  return(0);
}
```

Multiplicação

- Poderia ser feita como adição sucessiva, mas seria um processo muito lento
- Por exemplo, 999.999 ao quadrado envolveria 999.999 adições!
- Pode ser feita facilmente pelo método aprendido na escola que envolve somar várias linhas, alinhadas apropriadamente
- Cada operação envolve fazer um "shift" à esquerda e somar d vezes o primeiro número no total, sendo d o dígito apropriado do segundo número
- O fato de somar d vezes em vez de multiplicar pode parecer ineficiente, mas lembre-se que $d \leq 9$

Aritmética de alta precisão - multiplicação

```
multiply_bignum(bignum *a, bignum *b, bignum *c)
{
  bignum row;      /* represent shifted row */
  bignum tmp;      /* placeholder bignum */
  int i,j;         /* counters */

  initialize_bignum(c);

  row = *a;

  for (i=0; i<=b->lastdigit; i++) {
    for (j=1; j<=b->digits[i]; j++) {
      add_bignum(c,&row,&tmp);
      *c = tmp;
    }
    digit_shift(&row,1);
  }

  c->signbit = a->signbit * b->signbit;

  zero_justify(c);
}
```

Aritmética de alta precisão - shifting

Shifting

- Equivalente a multiplicar por 10 (já que estamos usando base 10)

Aritmética de alta precisão - shifting

```
digit_shift(bignum *n, int d)     /* multiply n by 10^d */
{
  int i;       /* counter */

  if ((n->lastdigit == 0) && (n->digits[0] == 0)) return;

  for (i=n->lastdigit; i>=0; i--)
    n->digits[i+d] = n->digits[i];

  for (i=0; i<d; i++) n->digits[i] = 0;

  n->lastdigit = n->lastdigit + d;
}
```

Aritmética de alta precisão - divisão

Divisão

- Operação temida tanto pelas crianças na escola quanto por quem trabalha com arquitetura de computadores, mas também pode ser feita com loop, mais simples do que se imagina
- Dividir por sucessivas subtrações é também ineficiente, assim como a multiplicação por sucessivas adições
- Mas também pode ser feita com a idéia de shifting:
    - fazer shift do resto para esquerda
    - adicionar o próximo dígito
    - subtrair sucessivamente o divisor (mais fácil que “adivinhar” o dígito do quociente, como aprendemos na escola)

Observação: o método calcula o quociente $a \div b$, e descarta o resto

Aritmética de alta precisão - divisão

```
divide_bignum(bignum *a, bignum *b, bignum *c)
{
  bignum row;                /* represent shifted row */
  bignum tmp;                /* placeholder bignum */
  int asign, bsign;          /* temporary signs */
  int i,j;                   /* counters */

  initialize_bignum(c);

  c->signbit = a->signbit * b->signbit;

  asign = a->signbit;
  bsign = b->signbit;

  a->signbit = PLUS;
  b->signbit = PLUS;

  initialize_bignum(&row);
  initialize_bignum(&tmp);

  ...
```

Aritmética de alta precisão - divisão

```
  ...

  c->lastdigit = a->lastdigit;

  for (i=a->lastdigit; i>=0; i--) {
    digit_shift(&row,1);
    row.digits[0] = a->digits[i];
    c->digits[i] = 0;
    while (compare_bignum(&row,b) != PLUS) {
      c->digits[i] ++;
      subtract_bignum(&row,b,&tmp);
      row = tmp;
    }
  }

  zero_justify(c);

  a->signbit = asign;
  b->signbit = bsign;
}
```

Aritmética de alta precisão - exponenciação

Exponenciação

- Pode ser feita com sucessivas multiplicações (e então sujeita aos mesmos problemas de desempenho de usar sucessivas adições para multiplicar)
- O truque é observar que
  $a^n = a^{n/2} \times a^{n/2} \times a^{n\%2}$
- Desta forma são usadas *logn* multiplicações

Conversão de bases

Bases

- **Binária**: números representados por 0 e 1; representação interna em computadores, já que 0 e 1 são naturalmente mapeados em *on*/*off* ou estado *alto*/*baixo*
- **Octal**: facilita leitura de binários, agrupando de 3 em 3 dígitos; $11111001_2 = 371_8$
- **Decimal**: base que usamos naturalmente (já que aprendemos a contar com os 10 dedos)
- **Hexadecimal**: mais conveniente que octal, para ler binários, agrupando de 4 em 4 dígitos; nesse caso, usando A a F para 10 a 15; $11111001_2 = F9_{16}$
- **Alfanumérica**: usando as 26 letras podemos usar base 36

Conversão de bases

Há basicamente dois algoritmos distintos para converter um número x na base a para um número y na base b

Algoritmos

- **Esquerda para direita**: encontrar o dígito mais significativo de y primeiro

  É o inteiro d_l tal que $(d_l + 1)b^k > x \geq d_l b^k$, sendo $1 \leq d_l \leq b - 1$

  Isso pode ser feito por tentativa e erro, ou semelhante à divisão de `bignum`

- **Esquerda para direita**: encontrar o dígito menos significativo de y primeiro

  Esse é o resto da divisão de x por b

  Foi o processo usado para converter `int` para `bignum` (no caso usando base 10)

Conversão de bases

Piadinha:

Porque programadores acham que Halloween é Natal?
Porque 31 Oct = 25 Dec !

Entrada/Saída em diferentes bases

```
#include <iostream>
using namespace std;

int main()
{
    int x = 31;

    cout << (hex) << x << endl;    // base hexadecimal: 1f
    cout << (oct) << x << endl;    // base octal: 37
    cout << (dec) << x << endl;    // base decimal: 31

    cin >> (hex) >> x;             // suponha entrada 'ff'
    cout << (dec) << x << endl;    // base decimal: 255

    return 0;
}
```

Entrada/Saída em diferentes bases

```
#include <cstdio>

int main()
{
    int x = 31;

    printf("%x %o %d\n",x,x,x); // 1f 37 31

    scanf("%x",&x);                   // suponha entrada 'ff'
    printf("%d\n",x);                 // 255

    return 0;
}
```

- `%x`, `%o`, `%d`: hexadecimal, octal e decimal
- `%X`: hexadecimal, maiúsculo. Por exemplo, 31_{10} sairia 1F

Números reais

Continuidade

- Muitas propriedades matemáticas são baseada na *continuidade* dos números reais, o fato de que sempre existe um número c entre a e b
- Isto não é verdade nos números reais representados em computadores

Por exemplo, $(a+b)/2$, com $a < b$, pode resultar em a se estiver no limite da precisão

Exatidão

- Muitos algoritmos assumem computação *exata*
- Isto não é verdade nos números reais representados em computadores

Por exemplo, $(a+b)+c$ pode ser diferente de $a+(b+c)$ por problemas de arredondamento

Números reais

Representação

- São representados em notação científica, i.e., $a \times 2^c$
- Tanto a *mantissa a* quanto o *expoente c* tem um número limite de bits (de acordo com um padrão IEEE)
- Operar com números de expoentes bem diferentes pode resultar em erros de *overflow* ou *underflow*, já que a mantissa não tem bits suficientes para acomodar todos os dígitos
- Isso causa muitos erros de arredondamento
- O que consequentemente torna inútil comparar dois reais, pois pode haver um lixo nos bits menos significativos

NUNCA teste de um número real é igual a zero! (ou outro valor qualquer) Em vez disso, teste se a diferença é menor que um ϵ bem pequeno

```
if (fabs(x-y)<EPS)    //sendo EPS = 1E-9
```

Números reais

Arredondamento x Truncamento

- Alguns problemas pedem o resultado com certo número de dígitos de precisão à direita do ponto decimal
- É preciso distinguir entre *truncar* e *arredondar*
- Truncar "corta" os dígitos (função `floor` por exemplo corta todos)
- Arredondar é usado para uma melhor precisão do dígito menos significativo (`cout` e `printf` fazem isso automaticamente)

Para arredondar um número x:

```
floor(X + 0.5)
```

Para arredondar um número x para k casas decimais:
$floor(10^k * X + 0.5)/10^k$

Números reais

Representação por frações

- Números racionais podem ser representados de forma **exata** usando frações
- O número x/y é representado por x e y, separadamente

```
struct frac {
   int num, den;
};

...
frac c;
c.num = 1;
c.den = 3;
```

representação exata de 1/3, sem o problema de precisão de ponto flutuante do 0.333333...

Números reais

pause

Operações aritméticas

Seja $a = \frac{n_A}{d_A}$ e $b = \frac{n_B}{d_B}$

- $a + b = \frac{n_A d_B + n_B d_A}{d_A d_B}$
- $a - b = \frac{n_A d_B - n_B d_A}{d_A d_B}$
- $a * b = \frac{n_A n_B}{d_A d_B}$
- $a/b = \frac{n_A d_B}{d_A n_B}$

Problemas de *overflow*

- Tais operações com frações geram rapidamente problemas de *overflow*, pois o denomidador está sempre crescendo
- Para reduzir esse problema, sempre *simplifique* a fração! (dividir pelo MDC)

Polinômios

Definição

- $P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \ldots$
- onde x é a variável e a_i o coeficiente do i-ésimo termo, x^i.
- o grau do polinômio é o maior i tal que $a_i \neq 0$.

Representação

A forma mais natural é um array de $n + 1$ coeficientes, de a_0 a a_n, sendo n o grau do polinômio. Semelhante à representação do `bignum`.

Avaliação

- Calcular $P(x)$ para algum x pode ser feito facilmente por força bruta, isto é, calcular cada termo $a_i x^i$ independentemente e somar os resultados
- Isso custa $O(n^2)$ multiplicações

Avaliação eficiente

- $P(x)$ pode ser avaliado com $O(n)$ multiplicações, usando o fato de que $x^i = x^{i-1}x$
- Calcular do menor para o maior grau, usando a potência de x anterior
- Desta forma são necessárias apenas 2 multiplicações para $a_i x^i$, que são $x^{i-1} \times x$ e $a_i \times x^i$

Regra de Horner

- Outra forma ainda mais esperta é usar a regra de Horner: $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0 = ((a_n x + a_{n-1})x + \ldots)x + a_0$

Polinômios

Adição/Subtração

- Ainda mais fácil que para inteiros de grande precisão, pois não é necessário usar "vai-um" nem "pegar emprestado"
- Simplesmente adicionar/subtrair os coeficientes do i-ésimo termo, para i de zero ao maior grau

Multiplicação

- O produto de dois polinômios $P(x)$ e $Q(x)$ é a soma do produto de cada par de termos, sendo cada termo de um dos polinômios
- $P(x) \times Q(x) = \sum_{i=0}^{grau(P)} \sum_{j=0}^{grau(Q)} (p_i q_j) x^{i+j}$

- Uma operação deste tipo, todos com todos, é chamada *convolução*
- Outros exemplos de convolução já vistos: multiplicação de inteiros (todos os dígitos com todos os dígitos) e string matching (toda posição do padrão com toda posição do texto)
- Existe um método que faz isso em $O(nlogn)$ em vez de $O(n^2)$!
  chama-se transformada rápida de Fourier

Divisão

- Dividir polinômios é uma tarefa complicada, pois a divisão não é uma operação fechada para polinômios (nem toda divisão resulta em polinômio)
- Por exemplo, $1/x$ pode até ser pensado como o polinômio x^{-1}, mas $2x/(x^2+1)$ não resulta de forma alguma em um polinômio

Polinômios

Polinômios esparsos

- Polinômios esparsos possuem muitos coeficientes iguais a zero
- Polinômios muito esparsos podem ser representados com pares coeficiente/grau para economia de memória

Polinômios multivariáveis

- São definidos com mais de uma variável
- Um polinômio de duas variáveis $P(x, y)$ pode ser representado em uma matriz A de coeficientes, tal que $A[i][j]$ guarda o coeficiente de $x^i y^j$

Logaritmos

Definição

- é o inverso da função exponencial
- $\log_b y = x$ é equivalente a $b^x = y$

Logaritmo natural

- Denotado por $\ln_x$, é o logaritmo base $e = 2.71828...$
- Sua função inversa é $\exp(x) = e^x$
- Logo, $\exp(\ln_x)) = x$

Logaritmo comum

- Denotado por $\log_{10}$ ou simplesmente log, é o logaritmo base 10
- Muito usado antes das calculadoras de bolso

Logaritmo binário

- Denotado por $\log_2$, ou lg, ou simplesmente log quanto o assunto é computação, é o logaritmo base 2

Logaritmos

Aplicações

- Lembre-se que $\log_a xy = \log_a x + \log_a y$
- ou seja, o logaritmo de um produto é a soma dos logaritmos
- Consequentemente, $\log_a n^b = b \log_a n$
- Logo, a^b pode ser calculado com as funções $\exp(x)$ e $\ln(x)$:

  $a^b = \exp(\ln(a^b)) = \exp(b \ln a)$
- Basta uma multiplicação e uma chamada a cada função
- Isto pode ser usado para calcular $\sqrt{x} = x^{\frac{1}{2}}$ ou qualquer outra potência fracionária

- Esteja ciente que são funções numéricas complicadas, calculadas usando expansões de séries de Taylor
- Então, estão sujeitas a erros de precisão
- Portanto, não espere que $\exp(0.5 \ln 16)$ dê exatamente 4

Logaritmos

Conversão de base

$$\log_a b = \frac{\log_c b}{\log_c a}$$

- Então, para mudar um logaritmo da base c para a base a basta dividir por $\log_c a$
- Portanto, é fácil escrever uma função para logaritmo comum a partir de logaritmo natural e vice-versa

```
#include <cmath>

double floor(double x);              // piso
double ceil (double c);              // teto
double fabs (double x);              // valor absoluto

double sqrt (double x);              // raiz quadrada
double exp  (double x);              // e^x
double log  (double x);              // logaritmo base e
double log10(double x);              // logaritmo base 10
double pow  (double x, double y);    // x^y
```