CENTRO DE MATEMÁTICA, COMPUTAÇÃO E COGNIÇÃO – UFABC
MCTA028-15 – Programação Estruturada – 3^o Quadrimestre de 2024
http://professor.ufabc.edu.br/~m.sambinelli/courses/2024Q3-PE/index.html
Prof. Maycon Sambinelli

O problema

Existem muitas versões para a origem do nome Google, a famosa empresa norte-americana que se destaca, entre outras coisas, pelos seus serviços de busca na Web. Uma das explicações mais aceitas vem de fontes ligadas à Universidade de Stanford, local onde tal empresa foi criada. Essa versão da história conta que o nome pretendido era na verdade "Googol" e que um dos seus fundadores cometeu um erro de grafia e acabou escrevendo "Google". Googol é o nome dado ao número 10^{100}. A ideia dos fundadores era escolher um nome que refletisse a imensa quantidade de informação que precisaria ser processada para indexar as páginas da Internet.

Por outro lado, podiam muito bem ter sido os zimbabuanos a inventar o nome Google. Existe uma lenda na Internet que diz que a inflação anual estimada em 2008 teria sido por volta de $6,5 \times 10^{108}\%$ ou 650 milhões de googols (googois?). Essa lenda surgiu do equívoco de alguém ter usado a inflação estimada de janeiro a novembro (79.600.000.000%) como sendo a inflação apenas do mês de novembro e partir daí extrapolando a estimativa anual[1]. De um jeito ou de outro, seria muito legal ter uma nota de Cem Trilhões de Dólares (ainda que sejam dólares zimbabuanos).

Figura 1: Nota de Z$100.000.000.000.000.

Para se ter uma ideia do quão absurdamente grande é um googol, veja a sua comparação com outros números enormes[2]:

- Desde que ocorreu o Big Bang, "só" se passaram 17×10^{39} de ioctossegundos (1 ioctossegundo = 10^{-24} segundo).
- Juntas, todas as pessoas do mundo viveram 5×10^{11} anos, ou 17×10^{18} segundos, ou "apenas" 17×10^{42} ioctossegundos.
- A massa do universo observável é estimada entre 10^{50} e 10^{60} Kg.
- Um Googol é aproximadamente igual a 70! (70 fatorial).

[1] https://en.wikipedia.org/wiki/Hyperinflation_in_Zimbabwe
[2] Exemplos "emprestados" da Wikipedia: https://pt.wikipedia.org/wiki/Googol.

Tendo em vista este tipo de problema, é comum em sistemas computacionais precisarmos de estruturas de dados que comportem números gigantescos. A maneira mais comum é guardar estes números em variáveis do tipo *float* ou *double* que, no final das contas (sem trocadilhos), armazenam uma aproximação do número em notação científica. Isso funciona em grande parte dos casos. Contudo, em alguns ramos da computação, como a criptografia, é preciso calcular e manipular números extremamente grandes (da ordem de 10^{300} ou mais) precisamente, sem aproximações. Comumente, variáveis do tipo inteiro em C vão apenas até 64 bits (ou $2^{64} \sim= 1,8 \times 10^{19}$) e alguns compiladores mais recentes têm suporte a inteiros de até 128 bits ($2^{128} \sim= 3,4 \times 10^{38}$) mesmo em máquinas 64 bits.

Neste projeto queremos ir além. Nós vamos desenvolver uma estrutura de dados cuja única limitação para o tamanho do número a ser armazenado seja a quantidade de memória do computador. Ao final do quadrimestre nosso projeto será capaz de lidar com números inteiros, positivos e negativos, sobre os quais efetuaremos as 4 operações aritméticas básicas: $+$, $-$, $\times$, $\div$ (divisão inteira).

1 Entradas e saídas

A entrada para o seu programa é fornecida via a entrada padrão (teclado do usuário) e o seu programa deve fornecer a saída na saída padrão (escrever na tela).

A entrada consiste de n casos de teste, onde $n \geq 1$. Cada caso de teste consiste de 3 linhas:

- A primeira linha de cada caso de teste consiste de um inteiro a e a segunda, de um inteiro b. Os números a e b podem ser prefixados por um sinal $+$ ou $-$. Também não há um limite superior nem inferior para os números a e b: a única limitação para esses números será a quantidade de memória da máquina que está rodando o nosso problema.
- A terceira linha do caso de teste consiste de um único caractere que pode ser `+`, `-`, `*` ou `/`.

Cada caso de teste (3 linhas) está fornecendo uma expressão numérica em formato pós-fixo. Exemplos: "5 3 +" , "7 3 -" . No nosso caso, cada um dos elementos da expressão estará em uma linha separada. Para cada caso de teste (conjunto de 3 linhas), você deve fornecer uma única linha contendo o resultado da operação aritmética, efetuando a operação da terceira linha sobre os dois números que a antecederam. Caso o resultado seja um número negativo, o sinal `-` deve ser prefixado aos dígitos do número (sem a inserção de espaço entre o sinal e o número).

Observação você deve seguir o formato de saída rigorosamente e não deve imprimir nenhum texto na tela que não tenha sido especificado acima, já que usarei testes automatizados para verificar a correção do trabalho.

A seguir, apresentamos alguns exemplos de entradas e saídas.

Exemplo 1

Entrada

```
3
5
+
2
8
-
12157865549787498543194363453256756324532235218789765476
3
-
```

```
3245899634153843978545123478312549796317362134
98312663286465303725297340976915494785435245565384217 8
+
9831266361105526714068173883142784261669022519712043 12
-1
-
```

Saída

```
8
-6
12157865549787498543194363453256756324532235218789765473
9831266361105526714068173883142784261669022519712043 12
9831266361105526714068173883142784261669022519712043 13
```

Exemplo 2

Entrada

```
-123582
2
*
979879888347697865634597873459723495879874529348798273458797987324524579872523 4
398798572345987528457982734598798723495879872398458823459879872359887234 58
+
3443
987987412341212341
-
```

Saída

```
-247164
979883876333421325509882453287069483867109488147522258047032586123248178744869 2
-987987412341208898
```

Outros casos de teste

Outros casos de teste podem ser obtidos no link abaixo:

http://professor.ufabc.edu.br/~m.sambinelli/courses/2024Q3-PE/trabalho/instances.tar.gz

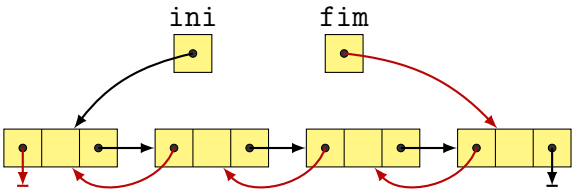
Os arquivos `.in` contêm os casos de testes de entrada e os arquivos `.out`, as saídas. Para usar de forma efetiva e simples esse gabarito fornecido, ao executar o seu programa, faça redirecionamento da entra e saída padrão [3] e use um programa de comparação de arquivos como: `diff`, `delta` [4] ou `meld` [5].

[3] https://www.ppgia.pucpr.br/pt/arquivos/techdocs/linux/foca-iniciante/ch-redir.html
[4] https://github.com/dandavison/delta
[5] https://meldmerge.org/

2 Requisitos

2.1 Representação do BigNumber

Internamente, você **deve** armazenar os dígitos do seu *BigNumber* em uma *lista duplamente encadeada*. Uma lista duplamente encadeada é uma versão um pouco mais complexa da lista encadeada vista em sala de aula. Na duplamente encadeada, cada nó da lista, além da informação armazenada, possui dois ponteiros: um para o próximo nó e um para o nó anterior, como exibido abaixo. Cada nó da lista irá armazenar um digito do BigNumber.



2.2 Git

O desenvolvimento do projeto deverá ser realizado com o auxílio do programa de controle de versão *git*. É provável que o grupo necessite utilizar um servidor de hospedagem online comercial para armazenar o projeto.

A plataforma gratuita mais amplamente utilizada é o GitHub, mas há outras alternativas igualmente boas, como GitLab e Bitbucket. Adicionalmente, é possível hospedar o projeto em um servidor privado, desde que os integrantes do grupo possuam acesso a tal máquina.

Caso optem por utilizar um dos serviços comerciais mencionados, configurem os repositórios como privados, a fim de evitar que colegas do curso tenham acesso indevido ao trabalho, o que poderia acarretar em plágio.

O uso do `git` não será abordado de maneira explícita ao longo do curso; entretanto, dúvidas podem ser esclarecidas durante as aulas práticas. Para aqueles que não estão familiarizados com o `git`, sugere-se a leitura dos seguintes materiais.

- Tutorial de Git e Github para Iniciantes
- git - Guia Prático
- Git Tutorial (eng)

Para aqueles que preferem videoaulas, o YouTube está repleto de tutorias sobre o Git.

Caso os recursos mencionados não sejam suficientes para sanar todas as dúvidas, sintam-se à vontade para esclarecê-las comigo durante as aulas.

O (histórico) do repositório git será utilizado, entre outras coisas, para avaliar a contribuição de cada membro do grupo. Por isso, certifiquem-se de distribuir as tarefas de desenvolvimento do projeto de forma igualitária.

Projetos desenvolvidos sem o uso do `git`, ou com um uso inadequado (por exemplo, desenvolver todo o projeto com apenas 4 *commits*), serão severamente penalizados.

3 Entrega

Este projeto poderá ser feito em grupos de duas a três pessoas. A entrega deverá ser feita pelo Moodle por, **apenas**, um dos integrantes do grupo. O prazo final para entrega é às 23h59m do dia **23-01-2025**.

1. Um único arquivo chamado `xxxxx.zip` deve ser entregue, onde `xxxxx` (aqui e no restante da seção) deve ser substituído pelos ra's dos participantes do grupo, separados por *underline*.

2. Ao descompactar o arquivo `xxxxx.zip`, uma única pasta chamada `xxxxx`, contendo todos os arquivos fontes do seu projeto, deve ser criada.

3. Dentro da pasta `xxxxx`, obrigatoriamente, devem existir os seguintes arquivos:

   - `bignumber.h` um arquivo de *header* de C contendo toda a interface pública do seu tipo BigNumber.
   - `bignumber.c` um arquivo de C contendo a implementação da sua interface pública.
   - `client.c` um arquivo de C contendo a função `main()` e que é responsável por usar a sua biblioteca `bignumber.h` para resolver o problema do projeto.
   - `makefile` um arquivo de configuração que permita o programa `make` compilar corretamente o seu programa quando o seguinte comando for digitado dentro do diretório `xxxxx`: `make`.
   - `.git` o diretório que armazena as informações do repositório *git* do seu projeto.
   - Um arquivo `pdf` chamado `README`, com no máximo 3 páginas, contendo um pequeno relatório. Este relatório deve conter o nome e ra de todos os membros do grupo. Além disso, deve conter:
     - Explicação da organização do código.
     - Explicação de quais funcionalidades foram implementadas corretamente.
     - Explicação de quais funcionalidades extras foram implementadas.
     - Qual a interface pública do seu tipo `BigNumber` (basta as assinaturas).
     - Mencione qualquer algoritmo ou estrutura de dados avançada que tenha sido empregada para melhorar o tempo de execução do seu código. Além disso, diga como/onde usou.
     - Diga, de forma geral, como foi a divisão de trabalho dentro da equipe, i.e., quem fez o quê?
   - Não há nenhum problema em dividir o seu programa em outros arquivos, essa é apenas a divisão mínima.

4. O seu makefile deve compilar o seu programa usando o programa `gcc` com as seguintes flags: `-std=c99 -Wall -Wextra -Wvla -g`

5. As únicas bibliotecas permitidas para o desenvolvimento do projeto são as bibliotecas da linguagem C (`stdio.h`, `string.h`, `stdlib.h`, `math.h`, `limits.h` e etc).

4 Avaliação

Para a composição da nota final do trabalho, serão levados em contas os seguintes aspectos:

- Correção das operações de soma, subtração, multiplicação e divisão, que serão averiguadas de forma automatizada. Cada operação implementada corretamente terá um peso na composição da nota final. É preferível que o seu programa suporte apenas as operações de soma e subtração, mas que funcione corretamente para 100% dos casos de teste, do que "suportar as 4 operações" e falhar em 40% dos casos de teste.

- Desempenho do programa (tempo de execução).

- Vazamento de memória: vazamentos de memória serão verificados com o seguinte comando `valgrind --leak-check=yes cliente`. Programas com vazamento de memória serão penalizados.
- Organização e legibilidade do código: indentação correta, modularização (i.e., divisão em funções, em arquivos, criação de tipos e etc.), nomenclatura adequada (para variáveis e funções) e comentários.
- A avaliação de cada membro da equipe será individual, o que pode levar a notas distintas para membros da mesma equipe. Essa avaliação personalizada será baseada na descrição das funcionalidades implementadas por cada membro da equipe, descrita no relatório, e pelos logs do *git* (`git log`).

Lembre-se: Plágios serão severamente punidos com a reprovação na disciplina.

4.1 Política de atrasos

Não serão aceitos trabalhos fora do prazo.

4.2 Bônus

4.2.1 Resto de Divisão – Bônus de 0.5 na P2

Receberão esse bônus todos os membros da equipe cujo programa, além de suportar os 4 operadores básicos ($+$, $-$, $\times$, $\div$), suportar o operador % (resto de divisão).

Para receber esse bônus a funcionalidade deve passar em todos os casos de testes.

4.2.2 Exponenciação Rápida – Bônus de 0.5 na P2

Receberão esse bônus todos os membros da equipe cujo programa, além de suportar os 4 operadores básicos ($+$, $-$, $\times$, $\div$), suportar o operador ^ (potenciação).

Sejam a e b o primeiro e segundo número fornecido ao programa, respectivamente, onde $b \geq 0$. Caso a terceira linha contenha o operador ^, é esperado que o programa produza o valor a^b. Ademais, o seu programa deve, **obrigatoriamente**, produzir esse resultado usando o método abaixo conhecido com *exponenciação rápida*:

$$a^b = \begin{cases} 1, & \text{se } b = 0 \\ (a^{b/2})^2, & \text{se } b > 0 \text{ e } b \text{ é par} \\ aa^{b-1}, & \text{se } b > 0 \text{ e } b \text{ é ímpar} \end{cases}$$

Para receber esse bônus a funcionalidade deve passar em todos os casos de testes.

4.2.3 Karatsuba – Bônus de 1.0 na P2

Para as o operações de soma e subtração, o melhor algoritmo que conhecemos é realmente aquele aprendido na escola. Para a multiplicação a história é outra: existem algoritmos mais eficientes do que aquele aprendido na escola. Um deles é o algoritmo de Karatsuba[6].

Receberão esse bônus todos os membros da equipe cujo programa realizar a operação de multiplicação $\times$ usando o algoritmo de Karatsuba.

Para receber esse bônus a funcionalidade deve passar em todos os casos de testes (com números positivos e negativos).

[6] https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/karatsuba.html

5 Dicas

Sugerimos que você crie um tipo de dados `BigNumber` para armazenar os dígitos e qualquer outro metadados necessário pelo seu "numerão". Isso facilitará a modularização do seu código e na definição de boas, úteis e versáteis interfaces. Uma boa modularização também levará a uma menor ocorrência de bugs e uma melhor divisão de trabalho na equipe.

Sugerimos também que você crie funções para criar um `BigNumber` a partir de um inteiro (`int`) ou a partir de uma string (`char []`) e para imprimir o `BigNumber` armazenado. Isso facilitará bastante os seus testes.

Crie funções para cada operação (soma, subtração e multiplicação, divisão) que será realizada sobre os `BigNumbers`.

Para a implementação das operações propriamente ditas, utilize os algoritmos de soma e subtração que você está acostumado (papel e lápis). Eles são surpreendentemente eficientes, mesmo para números extremamente grandes. Note que, uma vez implementada as funcionalidades de soma e subtração, o trabalho de implementar as funcionalidades de multiplicação e divisão é trivial.

Evite duplicar dados e tome cuidado com a passagem de parâmetros. Em C, toda passagem de parâmetro é por valor, ou seja, uma cópia do dado é feita. Uma forma de mitigar esse problema é passar endereços de memória das estruturas grandes (um endereço de memória é um dado pequeno – 64 bits). Copiar um `struct` com um vetor estático como membro requer uma operação de cópia de todos os elementos do vetor (se o vetor foi alocado dinamicamente, então apenas endereços são copiados).

Você pode usar o programa `gprof` [7] para analisar o tempo gasto em cada etapa de execução do seu programa, para tentar identificar gargalos e, assim, estudar melhorias para otimizar esses pontos. O uso do `gprof` não será ensinado ao longo do curso. É uma ferramenta simples e existem diversos bons materiais na web ensinando a usá-lo.

Você pode verificar vazamentos de memória no seu programa com o uso do programa `valgrind`. O professor Rafael Schouery escreveu um tutorial bem simples sobre o seu uso[8].

5.1 Sugestão de passos para o desenvolvimento

1. Planeje-se para o desenvolvimento desse trabalho. Esse não é o tipo de trabalho que você e o seu grupo conseguirá realizar "focando na última semana". Surgirá dúvidas que deverão ser discutidas comigo, *bugs* difíceis de serem encontrados por mentes cansadas e etc.

2. Avance devagar, seguindo a abordagem de *Baby Steps*. Certifique-se de que cada etapa do programa está completamente funcional antes de prosseguir. Apenas passe para o próximo passo quando tiver 100% de certeza de que o código atual está correto e bem testado.

   Ignorar esse conselho, inevitavelmente, resultará em um código com tantos *bugs* que será praticamente impossível corrigi-los.

3. Comece trabalhando apenas com números positivos.

4. Crie o tipo `BigNumber` e funções para ler e escrever tal número. Teste à exaustão. Quando estiver confiante de que tais funções estão 100% corretas, siga para o próximo passo.

5. Crie funções para criar um elemento do tipo `BigNumber` de um `int` e de um vetor de caracteres. Teste!

[7] https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html
[8] https://www.ic.unicamp.br/en/~rafael/materiais/valgrind.html

6. Implemente a operação de soma e teste-a à exaustão!

   Um dos critérios analisados na avaliação do seu programa é se a funcionalidade de soma funciona, pelo menos, quando todos os números envolvidos são positivos.

7. Faça a subtração. Primeiramente trabalhe apenas os casos de $a-b$ onde $a > b$. Teste à exaustão!

   Um dos critérios analisados será se a sua subtração funciona quando todos os números envolvidos são positivos.

8. Em seguida você vai ter que adicionar o suporte aos números negativos.

9. Faça testes exaustivos na soma com números negativos e positivos. Perceba que:

   - $a+b$ com $a \geq 0$ e $b < 0$ é igual a $a-|b|$.
   - $a-b$ com $b < 0$ é igual a $a+|b|$.
   - $a+b$ com $a < 0$ e $b < 0$ é igual a $-(|a|+|b|)$.
   - ...

10. Note que utilizando essas transformações é possível utilizar o código que você já tinha feito apenas para números positivos para tratar o caso de números negativos também!

11. Tendo as operações de soma e subtração preparadas, o trabalho acabou, a menos que o seu grupo esteja almejando algum bônus.

12. Continue em passo de bebê até o final do trabalho!

Divirta-se!