Estrutura de Dados 2
Roteiro de Laboratório 1 – Aquecimento

1 Crivo de Eratóstenes

O Crivo de Eratóstenes (em inglês *Sieve of Eratosthenes*) é um algoritmo milenar para se encontrar todos os números primos até um limite dado. Seu provável criador foi o matemático grego Eratóstenes, que apresentou o método no século III a.C. Até hoje, este ainda é um dos algoritmos mais eficientes para se encontrar todos os primos pequenos.

Um número primo é um número natural que possui exatamente dois divisores distintos: 1 e o próprio número. Para encontrar todos os primos menores ou iguais a um dado número N, faça:

1. Crie uma lista de números de 2 até N: $2, 3, 4, \ldots, N$.
2. Comece com $i = 2$, o menor número primo.
3. Marque todos os números na lista que são múltiplos de i.
4. Procure o primeiro número na lista maior que i que ainda não foi marcado. Este é o próximo primo i. Repita o passo 3. Se não for possível encontrar um novo número, pare.
5. Quando o algoritmo termina, os números que permaneceram não marcados são os primos entre 2 e N.

Pede-se:

1. Conforme foi feito em aula para o *union-find*, determine quais são as operações fundamentais necessárias para se resolver esse problema. Crie um arquivo `.h` para isolar a definição dessas operações das suas diferentes implementações.

2. Utilizando as operações (funções) definidas no item anterior, crie um programa cliente que recebe como parâmetro o valor N, inicializa e executa o crivo, e exibe os primos no terminal. Crie o seu código de forma que seja fácil modificá-lo para não se realizar mais a exibição (veja itens 5 e 6).

3. Qual é a estrutura de dados mais adequada para se usar na solução? Justifique a sua escolha pensando nas operações mais realizadas no código. Implemente e teste a sua solução usando o tipo `int` como marcador dos números.

4. Repita o item anterior utilizando: (i) um marcador do tipo `char` ou `bool` e (ii) um marcador do tipo *bit*. Nesse último caso será necessário utilizar as operações *bit*-a-*bit* do C para se economizar memória.

5. Utilizando a ferramenta `valgrind`, determine o consumo de memória das três variantes do seu código. *Dica: desabilite a impressão dos primos na tela para evitar a influência dos comandos de* `printf` *no consumo de memória.* O consumo medido era o que você esperava? Se não foi, procure entender o quê aconteceu.

6. Utilizando a ferramenta `time`, determine o tempo de execução das três variantes do seu código para $N = 10^i, i = 3, \ldots, 10$. Nem todas as combinações de implementação e tamanho de N serão possíveis, considerando o limite disponível de memória do computador

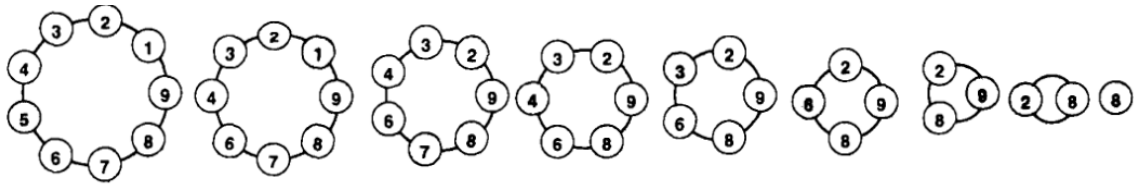
e o consumo de cada implementação. Com as informações obtidas nos itens anteriores, você deve ser capaz de identificar quais combinações não são viáveis sem ter de executar o programa (e travar o computador... :P). Para montar a sua tabela de tempo de execução, use o valor `user` retornado por `time`. Veja uma breve explicação dos três valores de `time` em `https://stackoverflow.com/questions/556405/what-do-real-user-and-sys--mean-in-the-output-of-time1/556411#556411`.

7. *Adiantando a próxima aula*: A partir da tabela de tempo do item anterior, você conseguiria estimar a ordem de crescimento (uma fórmula) para o tempo de execução do algoritmo em função do tamanho da entrada (N)?

Há uma série de alterações que podem ser feitas para melhorar tanto o tempo de execução quanto o consumo de memória do algoritmo. Veja `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes` para vários detalhes. Se estiver com disposição, implemente algumas das melhorias e realize novos testes como indicado anteriormente.

2 Problema de Josefo

O Problema de Josefo (em inglês *Josephus Problem*) foi apresentado pelo historiador Flávio Josefo no século I d.C. Suponha que N pessoas decidiram eleger um líder entre si. A eleição é feita da seguinte forma: as pessoas se organizam em um círculo e a M-ésima pessoa é eliminada, fechando o círculo quando alguém sai. O problema é descobrir quem será o líder, isto é, a última pessoa sobrando. (Uma pessoa com inclinação matemática pode determinar de antemão qual posição tomar no círculo para garantir a sua eleição.) Além disso, nós também queremos saber a *ordem* em que as pessoas são eliminadas. Por exemplo, na figura abaixo, se $N = 9$ e $M = 5$, as pessoas são eliminadas na ordem 5 1 7 4 3 6 9 2, e 8 é o líder escolhido.



Pede-se:

1. Conforme feito no problema anterior, determine quais são as operações fundamentais necessárias para se resolver esse problema e crie um arquivo `.h` adequado.

2. Crie um programa cliente que recebe como parâmetros os valores N e M, inicializa o círculo e exibe a sequência de pessoas eliminadas no terminal.

3. Qual é a estrutura de dados mais adequada para se usar na solução? Implemente e teste essa estrutura.

4. É possível prover uma variante da estrutura que é mais rápida e/ou consome menos memória? Tente fazer pelo menos uma variante.

5. Avalie o consumo de memória das suas implementações para um valor razoavelmente grande de N.

6. Com $M = 10$, avalie o desempenho das suas implementações em termo de tempo de execução para $N = 10^i, i = 3, \ldots, 9$.

7. *Adiantando a próxima aula*: A partir da tabela de tempo do item anterior, você conseguiria estimar a ordem de crescimento (uma fórmula) para o tempo de execução do algoritmo em função dos parâmetros (M e N)?

Maiores informações sobre esse problema em `https://en.wikipedia.org/wiki/Josephus_problem`.

3 Dijkstra's 2-stack

Um uso clássico de uma estrutura de dados do tipo pilha é a avaliação de expressões aritméticas. Uma expressão *pós-fixada*, isto é, uma expressão aonde os operandos aparecem antes do operador, pode ser facilmente avaliada utilizando-se uma pilha de números. Processando a entrada, sempre que um número é lido, ele é empilhado no topo da pilha. Quando um operador é lido, os dois primeiros valores da pilha são desempilhados, a operação é realizada e o resultado é empilhado. Ao final da avaliação, a pilha possui um único valor que é o resultado total da expressão. Por exemplo, a expressão `5 9 8 + 4 6 * * 7 + *` tem como resultado o valor `2075`.

Quando usamos uma notação infixada é necessário empregar parênteses para evitar ambiguidades. Por exemplo, a expressão anterior em notação infixada fica como abaixo.

```
( 5 * ( ( ( 9 + 8 ) * ( 4 * 6 ) ) + 7 ) )
```

O venerável programador E. W. Dijkstra propôs um algoritmo para a avaliação de expressões infixadas que utiliza duas pilhas, uma para os operandos e outra para os operadores. Uma expressão é formada por parênteses, operadores e operandos (números). Processando a expressão da esquerda para a direita, assumindo que essas entidades são separadas por espaço, as pilhas são manipuladas segundo quatro possíveis casos, como a seguir:

- Empilhe operandos na pilha de operandos.
- Empilhe operadores na pilha de operadores.
- Ignore abre parênteses `'('`.
- Ao encontrar um fecha parênteses `')'`, desempilhe um operador, desempilhe dois operandos, realize a operação e empilhe o resultado na pilha de operandos.

Assista o vídeo em `https://algs4.cs.princeton.edu/lectures/13DemoDijkstraTwoStack.mov` para ajudar no entendimento do funcionamento do algoritmo.

Pede-se:

1. Projete e implemente um Tipo Abstrato de Dados (TAD) *pilha*, utilizando a estrutura de dados interna que achar mais adequada. Note que foi pedido um TAD, o que implica que a implementação deve estar toda no arquivo `.c` e a estrutura da pilha deve ser opaca. O tipo armazenado na pilha deve ser facilmente modificável, para permitir o uso do TAD com diferentes tipos de elementos.

2. Crie uma pilha de operandos contendo `double`s e uma pilha de operadores contendo `char`s. Duplicação manual do código do TAD pilha através de *copy-paste* para lidar com esses dois tipos não é uma solução aceitável.

3. Utilizando as pilhas do item anterior, implemente o algoritmo proposto por Dijkstra e teste-o.