Trabalho 1 - Entrada/Saída, Controle de Fluxo, Funções e Arrays

Ciência da Computação - Estrturas de Dados I - Prof. Edkallenn

INSTRUÇÕES PARA A RESOLUÇÃO DOS EXERCÍCIOS:

- HÁ TRÊS FORMAS DE ENVIAR O EXERCÍCIO:
  1. Fazer o upload de cada arquivo-fonte (arquivo com a extensão `.c`) na área de envio do BlackBoard
  2. Criar um *repositório no Github* chamado `Lista 1 - Estruturas de Dados - Prof. Edkallenn` e fazer o push dos arquivos-fonte (somente os arquivos `.c`) sem os binários e, de preferência com este arquivo (das questões) incluído. **Você envia, portanto, SOMENTE O LINK DO REPOSITÓRIO.** O repositório não pode ter sido clonado sob pena de **ANULAÇÃO DO EXERCÍCIO.**
  3. Criar um Repl (repositório de projeto no Replit.com) e compartilhar o link com os arquivos do exercício (os arquivos-fonte `.c`). **Você envia, somente o link**.

  ▼ TODOS os arquivos-fonte tem uma forma de serem nomeados. Veja abaixo como cada arquivo deve ser nomeado.
- Os programas **NÃO DEVEM SER COMPACTADOS**. O código-fonte deve ser enviado via upload diretamente na resposta do exercício (arquivo por aquivo) conforme acima.
- **Cada arquivo deve ter o seguinte formato: `ED-lista1N1-questaoXX` onde `XX` é o número da questão correspondente.**
- **IMPORTANTE: NÃO SERÃO ACEITOS TRABALHOS QUE NÃO ESTIVEREM NO FORMATO ACIMA**
- **OBSERVAÇÃO**: **TODOS** os programas entregues devem ter o seguinte cabeçalho:

```
/*
**    Função :
**    Autor :
**    Data  :
**    Observações:
*/
```

Onde deverá estar escrito o que o programa faz, o autor (nome, turma, a data e as
observações que forem pertinentes.

Os trabalhos não serão aceitos após a data SOB HIPÓTESE ALGUMA.

---

1. **Potências Fatoriais**

O fatorial de um número, frequentemente denotado por $n!$, é o produto de todos os números inteiros positivos de `1` a `n`. A ***potência fatorial*** é uma generalização desse conceito. Existem dois tipos de potências fatoriais: **as potências fatoriais crescentes e as potências fatoriais decrescentes**.

- **Potência Fatorial Crescente (*Rising Factorial*)**:
  A
  potência fatorial crescente de um número natural `n`, denotada como $x^{\overline{n}}$, é o produto de `x` com os números inteiros consecutivos, começando por `1` até `n`. Matematicamente, é definida como:

$$x^{\overline{n}} = x \times (x+1) \times (x+2) \times ... \times (x+n-1)$$

  - A leitura da operação acima é a seguinte: "***x** elevado a **n** subindo*"

- **Potência Fatorial Decrescente (*Falling Factorial*)**:
  A
  potência fatorial decrescente de um número natural `n`, denotada como $x^{\underline{n}}$, é o produto de x com os números inteiros consecutivos, começando por n até 1. Matematicamente, é definida como:

$$x^{\underline{n}} = x * (x-1) * (x-2) * ... * (x-n+1)$$

  - A leitura da operação acima é a seguinte: "***x** elevado a **n** caindo*". DICA: É como se fosse um fatorial, mas ao invés de ir até o fim, você pega só os `n` primeiros termos.

Essas operações são úteis em matemática, particularmente em cálculos de permutações, combinações e em diversas fórmulas envolvendo progressões e séries. As potências fatoriais são usadas para descrever a variação de valores em sequências matemáticas.

Alguns exemplos de ambas as operações, a potência fatorial crescente (Rising Factorial) e a potência fatorial decrescente (Falling Factorial) usando um número `x` igual a `5` e `n` igual a `3` para ilustração.

- **Potência Fatorial Crescente (*Rising* Factorial)**:
  Como dito na definição acima a potência fatorial crescente é calculada multiplicando um número
  `x` por todos os inteiros consecutivos, começando em `1` até `n`. Aqui está um exemplo usando $x = 5$ e $n = 3$:

  $$5^{\overline{3}} = 5 * (5 + 1) * (5 + 2) = 5 * 6 * 7 = 210$$

  Portanto, $5^{\overline{3}} = 120$

- **Potência Fatorial Decrescente (*Falling* Factorial)**:
  A potência fatorial decrescente é calculada multiplicando um número `x` por todos os inteiros consecutivos, começando em `n` até `1`. Aqui está um exemplo usando $x = 5$ e $n = 3$:

  $$5^{\underline{3}} = 5 * (5 - 1) * (5 - 2) = 5 * 4 * 3 = 60$$

  Portanto, $5^{\underline{3}} = 60$.

Partindo da eplicação acima **escreva uma função em C** chamada `potencia_fatorial_crescente` que recebe dois parâmetros, `x` e `n`, e retorna o resultado da potência fatorial crescente de `x` elevado a `n`. Em seguida, escreva outra função chamada `potencia_fatorial_decrescente` que realiza o mesmo cálculo, mas para a potência fatorial decrescente. Utilize ambas as funções em exemplos (de preferência usando loops). Em seguida monte uma tabela usando um `x` fixo e variando o `n` de 2 até 10 executando ambas as operações. Logo após, faça também o `x` variar de 2 até 10. Exiba todos os resultados na tela. (DESAFIO: Salve os resultados em um arquivo texto).

2. **Fatorial Duplo**

   O **fatorial duplo**, ou *double factorial* em inglês, é uma extensão do conceito de fatorial . Em vez de multiplicar todos os números inteiros positivos até um número n, como no fatorial tradicional $n!$, o fatorial duplo multiplica apenas os números *pares* até n. A notação para o fatorial duplo de n é $n!!$.

A definição matemática do fatorial duplo é dada por:

$$n!! = n \times (n-2) \times (n-4) \times \ldots \times 4 \times 2$$

ou, de forma mais geral, para n par:

$$n!! = n \times (n-2) \times (n-4) \times \ldots \times 6 \times 4 \times 2$$

Para números ímpares, o fatorial duplo é definido de forma similar, mas inclui apenas os números ímpares em vez dos pares.

> 💡 **Exemplo:**
>
> - **Para um número par:**
>
> $$8!! = 8 \times 6 \times 4 \times 2 = 384$$
>
> $$6!! = 6 \times 4 \times 2 = 48$$
>
> - **Para um número ímpar:**
>
> $$7!! = 7 \times 5 \times 3 \times 1 = 105$$

Aplicações:

1. **Combinatória:**
   O fatorial duplo surge frequentemente em problemas de combinatória, especialmente ao contar arranjos alternados e permutações.
2. **Matemática Teórica:**
   É utilizado em algumas fórmulas matemáticas teóricas, especialmente aquelas relacionadas a funções especiais.
3. **Física Teórica:**
   Em algumas áreas da física teórica, o fatorial duplo pode aparecer em expressões matemáticas que modelam fenômenos específicos.

A principal distinção entre o fatorial duplo e o fatorial comum é que o fatorial duplo envolve a multiplicação de números alternados em vez de todos os números inteiros consecutivos.

Para números ímpares, a definição é semelhante, mas inclui apenas os números ímpares em vez dos pares.

Partindo da eplicação acima **escreva uma função em C** chamada `fatorial_duplo` que recebe o parâmetro, `n`, e retorna o resultado do fatorial

duplo de `n`. Em seguida monte uma tabela usando 2 até 20 mostrando, de forma tabulada, obviamente, o resultado do fatorial duplo para cada número. Aproveite e, na mesma tabela mostre o resultado do fatorial normal. Em seguida monte uma nova tabela com uma quarta coluna mostrando a diferença entre o fatorial normal e o fatorial duplo. Exiba todos os resultados na tela. Use arrays (com alocação estática) para armazenar os fatoriais duplos e normais de cada número. (DESAFIO: Salve os resultados em um arquivo texto).

3. **Soma de Divisores Amigáveis:**

   Um divisor próprio de um número `n` é qualquer número inteiro positivo que divide `n` de forma exata, excluindo o próprio `n`. Por exemplo, os divisores próprios de 12 são 1, 2, 3, 4 e 6.

   Dois números inteiros positivos `a` e `b` são considerados **divisores amigáveis** se a soma dos divisores próprios de `a` é igual a `b` e vice-versa. Ou seja, se a soma dos divisores próprios de `a` é igual a `b` e a soma dos divisores próprios de `b` é igual a `a`.

   Por exemplo:

   - Os divisores próprios de 220 são 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 e 110. A soma desses divisores é 284.
   - Os divisores próprios de 284 são 1, 2, 4, 71 e 142. A soma desses divisores é 220.

   > 💡 Portanto, 220 e 284 são um par de números amigáveis.

   Seguindo a ideia acima, escreva uma função em C que verifica se dois números são amigáveis. Esta função deve exibir na tela os divisores de ambos os números, assim como a soma desses divisores.

   Além disso, utilizando a função acima, escreva outra função que verifica todos os pares de números amigáveis até um determinado valor `n` que será fornecido pelo usuário.

4. **Número Automórfico:**

   Um **número automórfico** é um número que, quando elevado ao quadrado, produz um número cujos dígitos finais são iguais aos dígitos do número original. Por exemplo, 5 é um número automórfico porque $5^2 = 25$, e 25

tem os mesmos dígitos finais que 5. Ou seja, um número é automórfico se, e somente se, seu quadrado terminar com os mesmos dígitos do próprio número.

Escreva uma função em C chamada `verificar_numero_automorfico` que verifica se um dado número inteiro `n` é automórfico ou não. Usando a função mostre todos os números automórficos entre 2 e 1000. Guarde-os em um array usando alocação dinâmica.

O exercício pede para você criar uma função em C chamada `eh_numero_automorfico` que recebe um número inteiro positivo como entrada e verifica se esse número é um número automórfico ou não.

> 💡 Por exemplo:
>
> - Para o número 5, a função deve retornar verdadeiro, pois $5^2 = 25$ e os últimos dígitos correspondem a 5.
> - Para o número 6, a função deve retornar falso, pois $6^2 = 36$ e os últimos dígitos não correspondem a 6.

Em seguida mostrar todos os automóficos entre 2 e um número `n` digitado pelo usuário. Guarde esses números em um array alocado dinamicamente e em seguida exiba-os, todos.

5. **Soma de Fatoriais Inversos:**

   Escreva uma função em C chamada `soma_fatoriais_inversos` que recebe um número inteiro positivo `n` como entrada e retorna a soma dos fatoriais inversos dos números de 1 a `n`. O fatorial inverso de um número `x` é definido como $\frac{1}{x!}$.

   A **soma de fatoriais inversos** é uma soma de frações onde os denominadores são fatoriais. A fórmula geral é:

   $$S = \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!}$$

   O exercício solicita que você crie uma função em C chamada `soma_fatoriais_inversos` que recebe um número inteiro positivo `n` como entrada e calcula a soma dos fatoriais inversos até o termo `n`.

   Por exemplo:

   - Para $n = 3$, a função deve retornar $\frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} = 1 + \frac{1}{2} + \frac{1}{6} = \frac{11}{6}$

6. **Número de Carmichael:**

   Um **número de Carmichael** é um número composto `n` que satisfaz a propriedade de que $a^n \equiv a(\mod n)$ para todos os inteiros `a`, onde `a` é co-primo com `n`. Em outras palavras, para um número de Carmichael, a congruência de Fermat é sempre verdadeira. Escreva uma função em C chamada `verificar_numero_carmichael` que verifica se um dado número inteiro `n` é um número de Carmichael ou não. Em seguida exiba todos os números de Carmichael de 2 até o número `n` digitado pelo usuário.

   Explicando melhor,

   Um número de Carmichael é um número composto n que satisfaz a congruência modular $a^n \equiv a(\mod n)$ para todo a inteiro coprimo a n.

   A função que você precisa criar em C, chamada `eh_numero_carmichael`, deve receber um número inteiro positivo `n` como entrada e verificar se ele é um número de Carmichael ou não.

   > 💡 Por exemplo:
   >
   > - Para $n = 561$, a função deve retornar **verdadeiro**, pois $2^5 61 \equiv 2(\mod 561)$, $3^{561} \equiv 3(\mod 561)$, $4^{561} \equiv 4(\mod 561)$, e assim por diante para todos os inteiros coprimos a 561.
   > - Para $n = 10$, a função deve retornar **falso**, pois 10 não satisfaz a congruência $a^{10} \equiv a(\mod 10)$ para todos os inteiros coprimos a 10.

   Dois números inteiros são considerados "**coprimos**" se o único divisor comum entre eles for 1. Em outras palavras, *eles não têm nenhum fator primo em comum*.

💡 Por exemplo:

- Os números 15 e 28 são coprimos porque o único divisor comum entre eles é 1.
- Os números 12 e 25 são coprimos porque não têm nenhum divisor comum além de 1.

No contexto do exercício sobre números de Carmichael, quando dizemos "inteiros coprimos a n", estamos nos referindo a inteiros que não têm nenhum fator comum com n além de 1. Isso é importante porque a congruência modular $a^n \equiv a(\mod n)$ precisa ser verdadeira para todos os inteiros coprimos a n para que n seja considerado um número de Carmichael.

💡 Para deixar ainda mais claro:

- Dois números **não são coprimos** quando possuem pelo menos um divisor comum além do número 1. Aqui estão alguns exemplos:
  - 15 e 9 não são coprimos, pois têm 3 como divisor comum.
  - 21 e 14 não são coprimos, pois têm 7 como divisor comum.
  - 8 e 12 não são coprimos, pois têm 2 como divisor comum.
  - 30 e 45 não são coprimos, pois têm 3 e 5 como divisores comuns.

7. **Números Felizes:**

   Um **número feliz** é definido pelo seguinte processo: comece com qualquer número inteiro positivo e substitua o número pela soma dos quadrados dos seus dígitos. Repita o processo até que o número se torne 1 ou entre em um

ciclo infinito que não inclui 1. Se o processo terminar com 1, então o número é considerado feliz; caso contrário, é infeliz.

> 💡 Por exemplo:
>
> - 19 é um número feliz porque $1^2 + 9^2 = 82$, $8^2 + 2^2 = 68$, $6^2 + 8^2 = 100$ e $1^2 + 0^2 + 0^2 = 1$.
> - 4 não é um número feliz porque $4^2 = 16$ e $1^2 + 6^2 = 37$, $3^2 + 7^2 = 58$, $5^2 + 8^2 = 89$ e $8^2 + 9^2 = 145$, $1^2 + 4^2 + 5^2 = 42$ e $4^2 + 2^2 = 20$, $2^2 + 0^2 = 4$, que entra em um ciclo infinito sem nunca chegar a 1.

- Escreva uma função em C chamada `eh_numero_feliz` que recebe um número inteiro positivo `n` como entrada e retorna verdadeiro se `n` for um número feliz, e falso caso contrário. Use um array para checar se o número já foi "visitado" em um "percurso feliz". Se determinado número já foi visitado, então é um ciclo infinito e o número não é feliz.

8. **Números de Armstrong:**

   Um **número de Armstrong** é um número inteiro positivo no qual a soma dos cubos dos seus dígitos é igual ao próprio número. Por exemplo, 153 é um número de Armstrong porque $1^3 + 5^3 + 3^3 = 153$.

   Escreva uma função em C chamada `eh_numero_armstrong` que recebe um número inteiro positivo `n` como entrada e retorna verdadeiro se `n` for um número de Armstrong, e falso caso contrário. Usando esta função mostre todos os números de Armstrong entre 1 e 100.

9. **Números Perfeitos:**

   Um **número perfeito** é um número em que a soma de todos os seus divisores naturais próprios (excluindo ele mesmo) é igual ao próprio número

Por exemplo:

- O 1º número perfeito é 6 (soma dos divisores, menos o próprio número: 1 + 2 + 3).
- O 2º número perfeito é 28 (soma dos divisores: 1 + 2 + 4 + 7 + 14)
- E assim por diante.

Escreva uma função em C chamada `eh_numero_perfeito` que recebe um número inteiro positivo `n` como entrada e retorna verdadeiro se `n` for um número perfeito, e falso caso contrário. Usando esta função, exiba todos os números perfeitos entre 1 e 100000.

10. **Série de Fibonacci com Restrição Máxima**

    A série de Fibonacci é uma sequência de números em que cada número subsequente é a soma dos dois números anteriores. Começando com 0 e 1, os primeiros 10 termos serão: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

    Escreva uma função em C chamada `fibonacci_max` que recebe um número inteiro `max` como entrada e retorna um array contendo todos os números da série de Fibonacci que sejam menores ou iguais a `max`. O array retornado deve terminar com um marcador de posição especial, como -1. Use alocação dinâmica.

    Por exemplo, se `max` for 15, a função deve retornar o array {0, 1, 1, 2, 3, 5, 8, 13, -1}.

    Você pode incluir testes para verificar se a função está funcionando corretamente. Em seguida exibe o array.

11. **Permutação Circular**

    Uma permutação circular de uma sequência de números é obtida movendo cada número para a posição seguinte, e movendo o último número para a primeira posição. Por exemplo, a permutação circular de (1, 2, 3, 4) é (4, 1, 2, 3).

    Escreva uma função em C chamada `permutacao_circular` que recebe um array de inteiros e seu tamanho como entrada, e verifica se ele é uma permutação circular de outro array de inteiros do mesmo

tamanho. A função deve retornar 1 se for uma permutação circular e 0 caso contrário. Use alocação dinâmica.

12. DESAFIO: Crescimento Populacional (opcional) Adaptado por Neilor Tonin, URI Brasil (beecrowd 1160)

Mariazinha quer resolver um problema interessante. Dadas as informações de população e a taxa de crescimento de duas cidades quaisquer (A e B), ela gostaria de saber quantos anos levará para que a cidade menor (sempre é a cidade A) ultrapasse a cidade B em população. Claro que ela quer saber apenas para as cidades cuja taxa de crescimento da cidade A é maior do que a taxa de crescimento da cidade B, portanto, previamente já separou para você apenas os casos de teste que tem a taxa de crescimento maior para a cidade A. Sua tarefa é construir um programa que apresente o tempo em anos para cada caso de teste.

Porém, em alguns casos o tempo pode ser muito grande, e Mariazinha não se interessa em saber exatamente o tempo para estes casos. Basta que você informe, nesta situação, a mensagem "Mais de 1 século.".

Entrada

A primeira linha da entrada contém um único inteiro **T**, indicando o número de casos de teste (1 ≤ **T** ≤ 3000). Cada caso de teste contém 4 números: dois inteiros **PA** e **PB** (100 ≤ **PA** < 1000000, **PA** < **PB** ≤ 1000000) indicando respectivamente a população de A e B, e dois valores **G1** e **G2** (0.1 ≤ **G1** ≤ 10.0, 0.0 ≤ **G2** ≤ 10.0, **G2** < **G1**) com um digito após o ponto decimal cada, indicando respectivamente o crescimento populacional de A e B (em percentual).

Atenção: A população é sempre um valor inteiro, portanto, um crescimento de 2.5 % sobre uma população de 100 pessoas resultará em 102 pessoas, e não 102.5 pessoas, enquanto um crescimento de 2.5% sobre uma população de 1000 pessoas resultará em 1025 pessoas. Além disso, não utilize variáveis de precisão simples para as taxas de crescimento.

Saída

Imprima, para cada caso de teste, quantos anos levará para que a cidade A ultrapasse a cidade B em número de habitantes. Obs.: se o tempo for mais do que 100 anos o programa deve apresentar a mensagem: Mais de 1 século.

Neste caso, acredito que seja melhor interromper o programa imediatamente após passar de 100 anos, caso contrário você poderá receber como resposta da submissão deste problema "Time Limit Exceeded".

Exemplo de Entrada	Exemplo de Saída

`6`
`100 150 1.0 0`
`90000 120000 5.5 3.5`
`56700 72000 5.2 3.0`
`123 2000 3.0 2.0`
`100000 110000 1.5 0.5`
`62422 484317 3.1 1.0`
	`51 anos.`
	`16 anos.`
	`12 anos.`
	`Mais de 1 século.`
	`10 anos.`
	`100 anos.`

A última questão (12) é **um desafio** de modo que não é obrigatória!

🖖🏾 Bom Exercício! 😘

DICA: É IMPORTANTE QUE TODOS OS EXERCÍCIOS SEJAM REALIZADOS! E QUE VOCÊ MESMO FAÇA OS EXERCÍCIOS. SE NÃO CONEGUIR FAZER TODOS, ENVIE TODOS AQUELES QUE VOCÊ CONSEGUIR. NÃO DEIXE DE ENVIAR O TRABALHO. SUA NOTA DEPENDE ELE.

▼ Para tirar a **raiz quadrada** de um número use a função **sqrt(x)** e para potenciação use a função **pow(x,y)**

👩🏾‍🏫Prof. Ed

_____

- @Prof. Edkallenn
- @edkallenn
- @Edkallenn Lima