## Exercício 3: Dinheiro (versão fácil)

### ****Algoritmos Gulosos ou Algoritmos Ambiciosos****



Ao dar o troco, é provável que você queira minimizar o número de moedas que está distribuindo para cada cliente, para não acabar com o estoque (ou irritar o cliente!). Felizmente, a ciência da computação deu aos caixas em todos os lugares maneiras de minimizar o número de moedas devidas: algoritmos ambiciosos, também conhecidos como gulosos ou gananciosos.

De acordo com o Instituto Nacional de Padrões e Tecnologia (NIST), um algoritmo ambicioso é aquele “que sempre pega a melhor solução imediata, ou local, enquanto encontra uma resposta. Algoritmos ambiciosos encontram a solução geral ou globalmente ideal para alguns problemas de otimização, mas podem encontrar soluções menos do que ideais para algumas instâncias de outros problemas.”

O que tudo isso significa? Bem, suponha que um caixa deva a um cliente algum troco e na gaveta desse caixa estejam moedas de 25, 10, 5 e 1 centavo(s). O problema a ser resolvido é decidir quais moedas e quantas de cada uma entregar ao cliente. Pense em um caixa “ganancioso” como alguém que quer tirar o maior proveito possível desse problema com cada moeda que tira da gaveta. Por exemplo, se algum cliente deve pagar 41 centavos, a maior “mordida”(ou seja, melhor “mordida” imediata ou local) que pode ser feita é 25 centavos. (Essa mordida é "melhor" na medida em que nos deixa mais perto de 0 ¢ mais rápido do que qualquer outra moeda faria.) Observe que uma mordida desse tamanho reduziria o que era um problema de 41 ¢ a um problema de 16 ¢, já que 41 - 25 = 16. Ou seja, o restante é um problema semelhante, mas menor. Desnecessário dizer que outra mordida de 25 centavos seria muito grande (supondo que o caixa prefere não perder dinheiro), e assim nosso caixa ganancioso mudaria para uma mordida de 10 centavos, deixando-o com um problema de 6 centavos. Nesse ponto, a ganância pede uma mordida de 5 centavos seguida de uma mordida de 1 centavo, ponto em que o problema é resolvido. O cliente recebe um quarto, um centavo, um centavo e um centavo: quatro moedas no total. Acontece que essa abordagem gananciosa (do algoritmo) não é apenas ótima localmente, mas também globalmente para a moeda dos Estados Unidos (e também da União Europeia). Ou seja, desde que o caixa tenha o suficiente de cada moeda, essa abordagem do maior para o menor renderá o menor número possível de moedas. Quão menor? Bem, diga-nos você!

### ****Detalhes de Implementação****

## Este vídeo irá te ajudar a entender o problema ;)

****Atenção:**** para adicionar legendas ao vídeo clique no botão CC localizado no Player e selecione a opção "Português (Brasil)".

Implemente, em um arquivo chamado ****cash.c**** em um diretório ****~/pset1/cash****, um programa que primeiro pergunta ao usuário quanto dinheiro é devido e depois imprime o número mínimo de moedas com as quais essa mudança pode ser feita.

Use ****get\_float****para obter a entrada do usuário e ****printf**** para gerar sua resposta. Suponha que as únicas moedas disponíveis sejam de 25, 10, 5 e 1 centavo(s).

* Pedimos que você use ****get\_float**** para que possa lidar com reais e centavos, embora sem o cifrão. Em outras palavras, se algum cliente deve R$9.75 (como no caso em que um jornal custa 25 centavos, mas o cliente paga com uma nota de R$10), suponha que a entrada de seu programa será de ****9.75**** e não de ****R$9.75**** ou ****975**** . No entanto, se algum cliente deve exatamente R$9, suponha que a entrada de seu programa será ****9.00**** ou apenas ****9****, mas, novamente, não ****R$9**** ou ****900**** . É claro que, pela natureza dos valores de ponto flutuante, seu programa provavelmente funcionará com entradas como 9.0 e 9.000 também; você não precisa se preocupar em verificar se a entrada do usuário está “formatada” como o dinheiro deveria estar.

Você não precisa tentar verificar se a entrada de um usuário é muito grande para caber em um ****float****. Usar ****get\_float**** sozinho garantirá que a entrada do usuário seja realmente um valor de ponto flutuante (ou integral), mas não que seja não negativo.

Se o usuário não fornecer um valor não negativo, seu programa deve solicitar novamente ao usuário uma quantia válida até que o usuário concorde.

Para que possamos automatizar alguns testes do seu código, certifique-se de que a última linha de outpt do seu programa seja apenas o número mínimo de moedas possível: um inteiro seguido por ****\n****.

Cuidado com a imprecisão inerente aos valores de ponto flutuante. Lembre do ****floats.c**** da aula, em que, se ****x**** é 2 , e ****y**** é 10 , ****x / y****não é precisamente dois décimos! E assim, antes de fazer a alteração, você provavelmente desejará converter os dólares inseridos pelo usuário em centavos (ou seja, de um ****float****para um ****int****) para evitar pequenos erros que poderiam se acumular!

Tome cuidado para arredondar seus centavos até o último centavo mais próximo, por exemplo usando o **round**, que é declarado na ****math.h****. Por exemplo, se o real é um ****float****com input do usuário (por exemplo, ****0.20**** ), então uma linha como:

int centavos = round(reais \* 100);

irá converter com segurança ****0.20**** (ou mesmo 0.2000002980232238769531250 ) em 20.

****Utilize o ponto final ao invés de vírgula!!****

Seu programa deve se comportar de acordo com os exemplos abaixo.

$ ./cash  
Troca devida: 0.41  
****4****

 $ ./cash  
Troca devida: -0.41  
Troca devida: foo  
Troca devida: 0.41  
****4****

## **Como testar seu código no IDE do CS50?**

Seu código funciona conforme prescrito quando você insere:

* ****-1.00**** (ou outros números negativos)?
* ****0.00**** ?
* ****0.01**** (ou outros números positivos)?
* letras ou palavras?
* nenhuma entrada, quando você apenas pressiona Enter?

Execute o seguinte para avaliar se seu código está correto usando **check50**. Mas certifique-se de compilar e testar você mesmo!

check50 cs50/problems/2021/x/cash

Execute o seguinte para avaliar o style do seu código usando ****style50****.

style50 cash.c