CES12 - Lab 2024
Paradigmas de Programação
Moedas de Troco

Implementar

3 algoritmos devem ser implementados: DC, PD e Greedy
Há uma classe derivada para cada algoritmo, fornecida com métodos em branco, e estas classes herdam a mesma classe `TrocoSolverAbstract`:

```
class TrocoSolverAbstract {
public:

      virtual void solve(const std::vector<unsigned int> &denom,
                         unsigned int value,
                         std::vector<unsigned int> &coins)=0;

      virtual std::string getName() { return "ABSTRACT"; }

      virtual long countRecursiveCalls() { return 0; }

};
```

Os métodos com cabeçalho sublinhado já foram implementados no arquivo header .h fornecido.
Portanto o aluno deverá implementar o método `solve` das 3 classes acima, uma para cada algoritmo.
O método `solve` recebe um vetor com as denominações das moedas, um inteiro que é o valor do troco, e deve preencher o vetor coins, com o número de moedas de cada denominação. Os índices do vetor coin devem corresponder aos do vetor denom.
E.g:
Entrada:

- Denominacoes [1 5 10] centavos
- Valor: 28

Saída

- Coins: [3 1 2]
    - Significa 3 moedas de 1c + 1 moeda de 5c + 2 moedas de 10c Totalizando 28 centavos como requisitado.

O método `countRecursiveCalls()` retorna zero por default. Mas para a classe `DivConquer`, ele foi reimplementado para retornar o atributo `recursiveCalls`, que vocês devem incrementar.

Testes

Testes são repetidos separadamente para cada algoritmo. /0, /1 ou /2 são sufixos no nome dos testes, significando PD (prog. dinâmica), GR (Greedy), ou DC (Divide and Conquer). (com a exceção do teste separado para DC (TrocoDCTest), que está marcado com /0 ao invés de /2)

Há uma suíte de testes triviais, que podem ser executados no início do desenvolvimento, e depois, todos os valores de troco até um determinado valor são testados sistematicamente.

O algoritmo DC possui um teste separado que termina em um valor de troco menor, pois é o mais lento.

Saída dos testes

Exemplo de arquivo:

```
1,0.002382,1,0
2,0.0018,2,0
3,0.00133,3,0
4,0.0014,4,0
5,0.001516,1,0
6,0.001563,2,0
7,0.001617,3,0
8,0.001642,4,0
9,0.001667,5,0
.
.
.
-1, 2.23476,0 , 0
```

Todas as linhas tem o formato:
<valor do troco>,<tempo de execução em ms>,<número de moedas>,<contagem de chamadas recursivas>

Exceto a última linha, indexada com -1:
-1, <tempo de execução de todos os testes mostrados nesse arquivo>, 0, 0
São 4 colunas na última linha apenas para harmonizar com o restante do arquivo e facilitar a leitura. Note que o tempo de execução total inclui as chamadas de escrita no arquivo para as linhas.

Todos os arquivos são apagados e reescritos a cada execução dos testes.

Para o algoritmo GREEDY (“GR”) os testes NÃO testam se o número de moedas para o troco é mínimo. Para “GR” os testes apenas checam se a soma dos valores das moedas escolhidas é igual ao valor requisitado.
No entanto, a saída dos testes contém o número de moedas de troco para cada valor de troco, e o aluno deve então comparar os diferentes algoritmos - lembre que o guloso não é ótimo - e há um teste onde o GR deve perder.

Executando alguns testes (filtros)

```
$ ./labTrocotests --gtest_filter=*TrocoTrivial*
```
Executa apenas os testes triviais, com um numero limitado de instancias

Para testar apenas um algoritmo:
```
$ ./labTrocotests --gtest_filter=*/0*
$ ./labTrocotests --gtest_filter=*/1*
$ ./labTrocotests --gtest_filter=*/2*
```
Testam respectivamente apenas PD, GR, ou DC
(com a exceção do teste separado para DC (TrocoDCTest), que está marcado com /0 ao invés de /2)

Ou combinando as 2 ideias acima
```
$ ./labTrocotests --gtest_filter=*TrocoTrivial*/0*
```
Recomendado ao iniciar o desenvolvimento de cada um dos algoritmos.

UPDATE: Comando para executar o PD sem o teste TrocoDCTest, ou seja, isolando o PD do DC.
A ordem /0, /1, /2 depende da ordem das declarações, o ano que vem corrijo. Para não bagunçar a submissão de vocês, podem usar o seguinte comando para executar os testes do PD sem executar o teste separado para o DC.
```
$  ./labTrocotests --gtest_filter=*/0*-*TrocoDCTest*
```
A sintaxe corresponde a um padrão positivo, um sinal ‘-’, e um padrão negativo

Relatório

Compare os algoritmos em relação a tempo de execução e otimalidade.

1. (peso 1) Plote os resultados, para cada valor de troco, de i) tempo de execução e ii) número de moedas de troco, que pode ser ótimo ou não; *de forma a embasar o argumento das próximas respostas*.

    a. Mantenha legibilidade e clareza em indicar legendas, eixos, labels, etc para compreensão do gráfico.
2. (peso 2) *Com base nos gráficos da pergunta anterior*, explique brevemente porque os algoritmos são mais rápidos ou mais lentos, e porque são ótimos ou não, *considerando as diferenças entre os paradigmas e como estas diferenças se aplicam na estrutura do problema específico*.
3. (peso 1) O algoritmo de Divisão e Conquista é claramente mais lento do que os outros. Mas isso é devido a uma limitação inerente ao paradigma em geral, ou se deve a alguma especificidade na estrutura do problema em questão? Forneça um exemplo de problema que pode ser resolvido rapidamente por Divisão e Conquista, e detalhe porque a estrutura deste problema permite essa solução rápida enquanto o nosso problema não permite.

Nota

(peso 6) Testes: cada teste (exceto os testes em TrocoTrivialTests) que passa vale um ponto. Nota 10 significa passar em todos os testes, segue a proporção.

(peso 4. A soma do peso definido nas questões vale 10 neste quesito) Perguntas do Relatório. Não valem respostas sem embasamento em resultados da sua implementação.

A nota calculada assim comporá a nota da Parte I (Troco) do lab de Paradigmas de Programação.

Dicas

Contando as chamadas recursivas e implementando a recursão

Ao incrementar a variável `recursiveCalls`, ela precisa ser zerada antes da próxima chamada. Portanto, é preciso um segundo método interno para implementar a recursão de verdade, separando inicialização da contagem e variáveis auxiliares e da solução recursiva em si mesma. Eu implementei com o seguinte cabeçalho:

```
private:
void TrocoSolverDivConquer::solveRecursive(
           const std::vector<unsigned int> &denom,
           unsigned int value,
           std::vector<unsigned int> &coins,
           int &q)
```

Note que o valor de retorno q é necessário para parar a recursão, mas não faz parte da interface externa que o testador usa.

Adaptando os algoritmos de Troco

Nós precisamos da quantidade de moedas para cada denominação e não apenas a quantidade total de moedas

- No algoritmo PD, a tabela que está nos slides contém o valor da última moeda usada para cada valor de troco. Seria mais fácil no nosso caso, se armazenasse, para cada valor de troco, o índice no vetor de denominações da última moeda usada, que pode ser usado diretamente para incrementar o vetor coins.
    - Assim, evitam-se buscas no vetor de denominações, que acrescentariam um outro fator na complexidade.
    - Exemplo: supondo que o vetor de denominações seja [1 2 5 10 25 50]. Ao invés de armazenar o valor 10, armazenamos o índice 3. E podemos com esse índice incrementar a posição 3 correspondente no vetor coins que conta as moedas usadas.

FAQ

O vetor de denominações sempre inclui moedas de 1 centavo? Pelo que vi nas aulas do Alonso, esse parece ser o básico para cada algoritmo.

Sem moedas de 1 centavo, para alguns valores de troco não haverá solução. O que não impede que os algoritmos executem, apenas pode acontecer de não haver solução.
Mas, (só para este problema que é simples), não precisa que o algoritmo funcione para outros casos além dos testes.

O vetor de moedas sempre vai ser crescente?

Acho que em todos os testes que fiz é crescente. Se precisar ordenar, O(nlogn) não é nada. Mas, de novo, (só para este problema que é simples), não precisa que o algoritmo funcione para outros casos além dos testes.

Já perceberam que dá pra transformar o problema do troco em uma variante do problema da mochila? O troco não é “fácil”. Se fosse, colocaria testes na casa de milhões pra vcs...

Apesar de serem realizados 24 testes, 15 deles em TrocoTrivialTests, são gerados apenas 5 arquivos para serem analisados e comparados?

É isso mesmo. BR200 já plota todos os valores de 1 a 200 centavos. GRLooses faz o mesmo, mas com um vetor de denominações onde o GR nem sempre chega no ótimo. Quanto ao DC, é testado separado porque demoraria muito se for até 200.

Os testes triviais são sanity e casos extremos, pra ter certeza que cobriu todos os casos: e.g, quando a resposta é uma vez todas as denominações; ou apenas uma moeda; ou valor 1 centavo.
Os testes verdadeiros de 'stress' são os que varrem os valores e geram os arquivos.

Bloopers de anos anteriores

- Atenção à escrita:
    - usem a palavra "**denominação**" para evitar expressões longas e confusas como 'quantidades de valores de moedas", "mais valores", etc.. Exemplo: "As denominações das moedas de Real são: 1; 0,50; 0.25; 0.10; 0.05 e 0.01."
    - 'exponencialmente' indica ordem exponencial de complexidade, não "muito grande";
- O greedy não precisa testar todas as denominações, basta sair do loop quando o valor resultante chega a zero. Portanto, não necessariamente ele é mais lento para 1 ou 2 centavos. Cuidado para não confundir o algoritmo com a sua implementação.
- O tamanho da pilha de recursão é a altura da árvore de recursão. O número de chamadas recursivas é o número de nós na árvore de recursão.

Versão 20240515

- Adcionado o script zipaParaEntregar, igual ao dos labs anteriores, apenas faltava uma cópia aqui
- Update CMakeFile.txt
    - excluindo a receita antiga.
    - C++ standard default = C++20