Projeto de Algoritmos com Recursão Generativa

Fundamentos de Algoritmos

INF05008

Projetando Algoritmos

- **Análise e definição de dados:**
  - A escolha de **representação dos dados** para um problema afeta a nossa **forma de pensar sobre o processo**
  - Às vezes, a **descrição** de um processo **sugere uma representação específica** para os dados
  - Em outras situações, é possível, e até recomendável, **explorar alternativas**

Projetando Algoritmos

- **Contrato, propósito e cabeçalho:**
  - O passo generativo **não tem conexão com a estrutura** da definição de dados
  - Logo, além de dizer **o quê** a função faz, também devemos explicar, em geral, **como** ela o faz

Projetando Algoritmos

- **Exemplos:**
  - Antes, bastava relação entre entrada e saída nos exemplos
  - Agora, devem ilustrar **como a função trabalha com uma entrada específica**
  - Para algumas funções, isso é fácil de se fazer (função que movimenta bola, por exemplo)
  - Para outras, o processo é baseado em uma **idéia que não é trivial** (explicação baseada em um bom exemplo - como uma figura para o *quick sort*)

Projetando Algoritmos

- **Template:**

```
(define (função-generativa-recursiva problema)
  (cond
    [(solução-trivial-existe? problema)
     (determinar-solução problema)]
    [else
     (combinar-soluções
       ... problema ...
     (função-generativa-recursiva (gen-prob-1 problema))
     ...
     (função-generativa-recursiva (gen-prob-n problema)))]))
```

Projetando Algoritmos

- **Definição:** cada função que aparece no *template* tem o propósito de nos lembrar que devemos pensar sobre as seguintes questões:

  1. O quê é um problema de solução trivial? Qual é a solução correspondente?

Projetando Algoritmos

- **Definição:** cada função que aparece no *template* tem o propósito de nos lembrar que devemos pensar sobre as seguintes questões:

  1. O quê é um problema de solução trivial? Qual é a solução correspondente?
  2. Como dividimos um problema original em novos problemas, os quais são resolvidos mais facilmente do que o problema original? Quantos novos problemas devem ser gerados?

Projetando Algoritmos

- **Definição:** cada função que aparece no *template* tem o propósito de nos lembrar que devemos pensar sobre as seguintes questões:

  1. O quê é um problema de solução trivial? Qual é a solução correspondente?
  2. Como dividimos um problema original em novos problemas, os quais são resolvidos mais facilmente do que o problema original? Quantos novos problemas devem ser gerados?
  3. A solução do problema dado é a mesma solução para o(s) novo(s) subproblema(s)? Ou temos de combinar as soluções para criar a solução do problema original? E se for o caso, precisamos de algo do problema original?

Projetando Algoritmos

- **Definição:** cada função que aparece no *template* tem o propósito de nos lembrar que devemos pensar sobre as seguintes questões:

  1. O quê é um problema de solução trivial? Qual é a solução correspondente?
  2. Como dividimos um problema original em novos problemas, os quais são resolvidos mais facilmente do que o problema original? Quantos novos problemas devem ser gerados?
  3. A solução do problema dado é a mesma solução para o(s) novo(s) subproblema(s)? Ou temos de combinar as soluções para criar a solução do problema original? E se for o caso, precisamos de algo do problema original?

  As respostas a essas perguntas são dadas em termos da representação de dados escolhida!

Projetando Algoritmos

- **Testes:**
  - Depois de termos uma definição completa, devemos testá-la
  - Lembre-se de que **testes não garantem que função funciona corretamente para todas as possíveis entradas**!
  - Mas devemos selecionar testes que **aumentem a confiança** no correto funcionamento do programa

Exercícios

1. Responda às questões anteriores para o problema de modelar o movimento de uma bola sobre uma tela até que ela saia dos limites

2. Responda às questões anteriores para o problema *quick-sort*

Terminação

- Em **recursão estrutural**, cada chamada recursiva opera com uma **parte do dado original** (menor, portanto). Dessa forma, **em algum momento, a função consome um dado atômico e pode parar**

- Em **recursão generativa**, cada chamada recursiva opera com um **problema obtido a partir do problema original** (que é um novo dado)

- **A análise de terminação deve ser mais cuidadosa!**

Exemplo de Erro de Terminação

```
;; menores : lista-de-números número -> lista-de-números
;; Cria lista com todos os elementos de ldn que são menores
;; ou iguais a n

(define (menores n ldn)
 (cond
   [(empty? ldn) empty]
   [else
     (cond
       [(< (first ldn) n) (cons (first ldn)
                                (menores n (rest ldn)))]
       [else (menores n (rest ldn))])]))
```

- Pequeno erro na geração de problema pode levar a **não terminação**: por exemplo, $<=$ no lugar de $<$

Terminação

- Essa função produz `(list 5)` quando aplicada aos parâmetros `5` e `(list 5)`
- Se `quick-sort` for chamada para resolver o problema `(list 5)`, a execução de `(menores 5 (list 5))` produzirá o **mesmo problema original**, o qual é passado na recursão generativa
- Dessa forma, **`quick-sort` não produz nenhum resultado!**

Terminação

```
   (quick-sort (list 5))
= (append (quick-sort (menores 5 (list 5)))
          (list 5)
          (quick-sort (maiores 5 (list 5))))
= (append (quick-sort (list 5))
          (list 5)
          (quick-sort (maiores 5 (list 5))))
...
```

Terminação

- O projeto de algoritmos deste tipo precisa incluir mais um passo: **um argumento para terminação**
  - Ele explica porquê o processo sempre produz uma saída para qualquer entrada e como a função implementa essa idéia
  - Ou então avisa as situações nas quais a função não termina
- Para `quick-sort`, o argumento pode ser o seguinte:
  A cada passo, quick-sort particiona a lista em duas sub-listas usando as funções menores e maiores. Cada função produz uma lista que tem menos elementos do que o número de entrada, mesmo que este esteja presente na lista. Assim, cada chamada recursiva de quick-sort consome uma lista estritamente menor que a lista dada.
- Sem esse argumento, o algoritmo pode ser considerado **incompleto**

Exercícios

1. Desenvolva a função `mdc-e` que, dados dois números, retorna o máximo divisor comum (MDC) entre eles. Use recursão estrutural.

2. Desenvolva a função `mdc-g` que, dados dois números retorna o MDC entre eles, usando recursão generativa.

   Um *insight* matemático para achar o MDC de dois números:

   (MDC menor maior) = (MDC menor (remainder maior menor))