Diminuição e conquista

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Recursividade (Revisão)

Dados com autorreferência

- Lista
- Árvores
- Números Naturais
- ...

Quando projetamos algoritmos que processam dados com autorreferência colocamos diretamente uma chamada recursiva onde existe autorreferência na definição do dado.

- Lembrando que este tipo de recursão é chamada de **recursão natural**
- As demais chamadas recursivas são chamadas de **recursão generativa**

E os arranjos?

Embora não tenhamos definido arranjos como tendo autorreferências, nós aplicamos o mesmo processo de projeto de algoritmos compondo um arranjo com um número natural (que restringe o tamanho do arranjo) e fazendo a recursão baseado no tamanho (avançando o início ou diminuindo o fim).

Diminuição e conquista

Projetar algoritmos para dados com autorreferência nos leva naturalmente a uma técnica de projeto de algoritmo chamada **Diminuição e conquista**:

- Reduzir o problema para uma instância menor do mesmo problema
- Resolver a instância menor (usando a mesma técnica)
- Estender a solução da instância menor para a instância original

Observações

Embora tenhamos "derivado" esta técnica a partir do conceito de autorreferência, ela poder ser aplicada a qualquer tipo de dado!

- De fato nós já vimos esta técnica com outro nome: abordagem incremental (ou ainda, indutiva).

Todos os algoritmos que projetamos foram recursivos.

Todos os algoritmos que projetamos diminuíam o tamanho do problema em 1.

Perguntas

1) Esta técnica sempre funciona? Ou seja, se eu aplicar a ideia de diminuir e conquistar eu consigo projetar um algoritmo para resolver qualquer problema?
2) Os algoritmos projetados com essa técnica são sempre recursivos?
3) Os algoritmos projetados com essa técnica são eficientes?
4) Podemos diminuir o tamanho do problema em mais de uma unidade?

Aplicabilidade

Quando podemos aplicar a técnica de projeto diminuição e conquista?

- Quando conseguimos resolver a instância menor (caso base)
- Quando a solução da instância menor pode ser estendida para a solução da instância original

Aplicabilidade

Exemplos positivos

- Busca
- Ordenação
- $\ldots$

Exemplos negativos

- Encontrar os divisores de um número natural
  - Saber os divisores de 9 $(9, 3, 1)$ não ajuda e encontrar os divisores de 10 $(10, 5, 2, 1)$...
  - Mas nesse caso podemos formular o problema de forma diferente: encontrar os divisores de n que são menores ou iguais a x

Formas de implementação

Como podemos implementar os algoritmos de diminuição e conquista?

Talvez o jeito mais natural seja partir do problema original, diminuir o problema, resolver o problema menor e depois estender a solução! Esta forma de implementação é chamada de *top-down* – de cima para baixo.

No entanto, para alguns problemas também é possível fazer uma implementação *bottom-up* - de baixo para cima. O algoritmo parte da solução de um problema trivial e estende iterativamente a solução para um problema maior, até chegar na solução do problema original.

Formas de implementação

Apesar de a abordagem *top-down* levar naturalmente a uma implementação recursiva e a abordagem *bottom-up* levar naturalmente a uma implementação iterativa, é possível implementar as duas abordagens tanto de forma recursiva quanto de forma iterativa.

Em geral, as implementações recursivas são mais simples mas podem consumir mais memória ou mesmo mais tempo.

Nas implementações iterativas precisamos declarar uma invariante de laço que relaciona o estado da função com o subproblema resolvido.

Contém

```python
# Produz True se v está em lst;
# False caso contrário
def contem(lst: Lista, v: int) -> bool:
    if lst is None:
        return False
    else:
        if v == lst.primeiro:
            return True
        else:
            return contem(lst.resto, v)

# Fizemos a análise do tempo de execução
# durante a aula.
```

```python
# Produz True se v está em lst;
# False caso contrário
def contem(lst: Lista, v: int) -> bool:
    # Invariante:
    # v não é igual ao valor de nenhum
    # link antes de p
    p = lst
    while p != None
        if v == p.primeiro:
            return True
        p = p.resto
    return False
```

Inverte

```python
# Cria uma nova lista com os elementos
# de lst de tras para frente.
def inverte(lst: Lista) -> Lista:
    if lst is None:
        return None
    else:
        return link_fim(lst.primeiro,
                        inverte(lst.resto))

# Fizemos a análise do tempo de execução
# durante a aula.
```

```python
# Cria uma nova lista com os elementos
# de lst de tras para frente.
def inverte(lst: Lista) -> Lista:
    # Invariante:
    # inv contem os elementos em ordem
    # invertida que vieram antes de p.
    inv = None
    p = lst
    while p != None:
        inv = Link(p.primeiro, inv)
        p = p.resto
    return inv
```

Eficiência

Como calcular o tempo de execução e consumo de memória de um algoritmo diminuir para conquistar? Depende da abordagem!

- Iterativa, da forma que fizemos para os outros algoritmos iterativos
- Recursiva, usando equações de recorrências

Obs: fizemos as análises durante a aula.

Diminuição

Podemos diminuir o tamanho do problema em mais de uma unidade? Sim!

- Diminuição por uma constante
- Diminuição por um fator constante
- Diminuição variável

Inverte

Projete um algoritmo que inverta os valores de um arranjo (sem criar um novo arranjo).

Inverte

Primeira tentativa:

- Entrada: Um arranjo A e um natural n tal que $0 \leq n \leq A.length$
- Diminuir: A e $n - 1$
- Caso base: $n \leq 1$
- Estender: mover $A[n-1]$ para o início

Problemas:

- Tempo de $O(n^2)$!
- Estender a solução é $O(n)$

```python
# Inverte as posições dos elementos de A.
# O primeiro troca com o último,
# o segundo com o penúltimo,
# e assim por diante.
# Requer que 0 <= n <= len(A).
def inverte(A: List[int], n: int):
    if n <= 1:
        return
    else:
        inverte(A, n - 1)
        # faça o donwload do material
        # para ver o código completo
        move_inicio(A, n - 1)
```

Inverte

Segunda tentativa:

- Entrada: Um arranjo A e dois números naturais a e b, tal que $0 \leq a \leq b \leq A.length$
- Diminuir: A, $a + 1$ e $b - 1$
- Caso base: $b - a \leq 1$
- Estender: trocar $A[a]$ e $A[b - 1]$

Tempo de execução:

- $O(n)$
- Estender a solução é $O(1)$

```python
# Inverte as posições dos elementos
# de A[a : b].
# A[a] troca com A[b-1],
# A[a+1] troca com A[b-2],
# e assim por diante.
# Requer que 0 <= a <= b <= len(A).
def inverte2(A: List[int], a: int, b: int):
    if (b - a) <= 1:
        return
    else:
        inverte2(A, a + 1, b - 1)
        A[a], A[b - 1] = A[b - 1], A[a]
```

Contém

Projete um algoritmo que verifique se um dado valor está em um arranjo ordenado.

Contém

Ideia: Ver se o elemento está no "meio" do arranjo e se não estiver procurar em uma das outras duas metades.

- Entrada: Um arranjo A, um valor v e dois números naturais a e b, tal que $0 \leq a \leq b \leq A.length$
- Diminuir: A, a e $b = meio$ ou $A, a = meio + 1$ e b
- Caso base: $a == b$
- Estender: nada

Tempo de execução:

- $O(\lg n)$ (fizemos a análise durante a aula)

```python
# Produz True se v está em A[a : b];
# False caso contrário.
# Requer que 0 <= a <= b <= len(A).
def busca_binaria(v: int, A: List[int], a: int, b: int)
    if a == b:
        return False
    else:
        meio = (a + b) // 2
        if v == A[meio]:
            return True
        elif v < A[meio]:
            return busca_binaria(v, A, a, meio)
        else:
            return busca_binaria(v, A, meio + 1, b)
```

Referências

Referências

- Divide-and-conquer algorithm - Wikipedia
- Abordagem top-down e bottom-up - Wikipedia