Técnicas de Projeto de Algoritmos I

Rômulo César Silva

Unioeste

Julho de 2021

unioeste
Universidade Estadual do Oeste do Paraná

Sumário

Técnicas de Projeto de Algoritmos

A rigor, um mesmo problema pode ser resolvido de diferentes maneiras. Cada maneira equivale a um algoritmo diferente. Mas como obter um algoritmo?
Existem diferentes técnicas de projeto de algoritmos:

- algumas técnicas surgem a partir da observação de padrões de soluções em diferentes problemas e respectivos algoritmos.
- nenhuma técnica é aplicável a qualquer problema
- o uso de uma técnica não garante a obtenção de um algoritmo eficiente

Técnicas de Projeto de Algoritmos

Técnicas de Projeto de Algoritmos

As técnicas aqui apresentadas servem como referencial na obtenção de algoritmos para resolução de diversos problemas.

Assim:

- A técnica representa uma forma de abordagem ao problema visando obter uma solução (algoritmo)
- São apresentadas situações ou características típicas de problemas em que a técnica pode ser aplicada

Projeto por Indução

Projeto de Algoritmo por Indução

Um projeto de algoritmo por indução funciona como uma analogia à indução matemática, baseado em 2 princípios:

1. é possível solucionar o problema para uma instância pequena (caso base)
2. uma solução para todo o problema pode ser construída a partir de soluções de problemas menores (passo da indução)

Assim, a aplicação da técnica concentra-se em reduzir o problema a um problema menor ou a um conjunto de problemas menores, da mesma natureza do problema original.

Projeto por Indução

A técnica resulta em algoritmos recursivos tal que:

- a condição de parada corresponde aos casos base da indução
- as chamadas recursivas correspondem à aplicação da hipótese de indução
- o processo de obtenção da resposta para o caso geral do problema corresponde ao passo da indução.

Projeto por Indução

Benefícios:

- por se basear em uma técnica de demonstração, se aplicada corretamente, a técnica já corresponde à prova de corretude do algoritmo.
- a complexidade de tempo pode ser expressa em uma recorrência
- embora haja situações em que o uso de recursão não seja indicado, ainda assim, o emprego da técnica permite a obtenção de um algoritmo que pode ser facilmente convertido em um algoritmo iterativo.

Projeto por Indução

Observação

Em alguns problemas, a técnica de projeto por indução pode ser aplicada de diferentes maneiras. Dependendo da forma que a indução é feita, obtêm-se em algoritmos diferentes, porém todos corretos. Porém a diferença significativa entre eles pode estar na complexidade de tempo.

Apresentaremos a seguir um exemplo de problema para exemplificar o uso da técnica e também como o modo de usar a indução pode resultar em um algoritmo com complexidade de tempo consideravelmente menor.

Projeto por Indução - exemplo

Problema do Cálculo de Polinômios

Dada uma sequência de números reais $a_n, a_{n-1}, ..., a_1, a_0$, e um número real x, calcular o valor do polinômio

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

Projeto por Indução - exemplo (1ª abordagem)

1ª abordagem: $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ pode ser calculado indutivamente da seguinte maneira:

- caso base: $n = 0$. A solução é $P(n) = a_0$
- hipótese de indução: suponha que dado n, sabe-se calcular $P_{n-1}(x) = a_{n-1} x^{n-1} + ... + ...a_1 x + a_0$.
- passo da indução: Logo, para calcular $P_n(x)$ é só calcular x^n e multiplicar por a_n e em seguida somar com $P_{n-1}(x)$. Isto é, $P_n(x) = P_{n-1}(x) + a_n x^n$.

Projeto por Indução - exemplo (1ª abordagem)

Esse projeto por indução conduz ao algoritmo recursivo abaixo:

```
1:  function CalculaPolinomio(A, x)
2:      ▷ Entrada: coefecientes A = a_n, a_{n-1}, ..., a_1, a_0 e real x
3:      ▷ Saída: P_n(x)
4:      if n = 0 then
5:          return a_0
6:      else
7:          A' ← a_{n-1}, ..., a_1, a_0
8:          P' ← CalculaPolinomio(A',x)          ▷ Cálculo de P_{n-1}(x)
9:          aux ← 1
10:         for i ← 1, ..., n do                 ▷ Cálculo de x^n
11:             aux ← aux * x
12:         end for
13:         P ← P' + a_n * aux
14:         return P
15:     end if
16: end function
```

Projeto por Indução - exemplo (1ª abordagem)

Calculando a complexidade de tempo, usando a análise simplificada, levando conta apenas operações aritméticas de multiplicação e adição, pode-se expressá-la na recorrência:

$$T(n) = \begin{cases} 0 & \text{se} \quad n = 0 \\ T(n-1) + n + 1 & \text{se} \quad n > 0 \end{cases}$$

considerando que são n multiplicações e 1 adição feitas e mais o custo da chamada recursiva que tem tamanho $n-1$. Resolvendo a recorrência chega-se que $T(n) = \frac{(n+1)n}{2} + n$ ou seja $T(n) \in \Theta(n^2)$, sendo portanto um algoritmo de complexidade quadrática.

Projeto por Indução - exemplo (2ª abordagem)

2ª abordagem: considere agora o mesmo problema (Cálculo de Polinômios), mas usando:

- hipótese de indução: suponha que sabe-se calcular $P_{n-1}(x) = a_{n-1}x^{n-1} + ... + a_1x + a_0$ e também o valor de x^{n-1}.
- passo da indução: calculando x^n como sendo a multiplicação de x^{n-1} por x e em seguida calculando $P_n(x) = x^n * a_n + P_{n-1}$.

Projeto por Indução - exemplo (2ª abordagem)

Essa nova indução conduz ao seguinte algoritmo recursivo:

```
 1: function CalculaPolinomio(A, x)
 2:     ▷ Entrada: coefecientes A = a_n, a_{n-1}, ..., a_1, a_0 e real x
 3:     ▷ Saída: P_n(x) e x^n
 4:     if n = 0 then
 5:         return (a_0, 1)
 6:     else
 7:         A' ← a_{n-1}, ..., a_1, a_0
 8:         P', aux ← CalculaPolinomio(A',x)        ▷ Cálculo de P_{n-1}(x) e de x^{n-1}
 9:         aux ← aux * x
10:         P ← P' + a_n * aux
11:         return (P, aux)
12:     end if
13: end function
```

Projeto por Indução - exemplo (2ª abordagem)

Calculando a nova complexidade de tempo, obtém-se a recorrência:

$$T(n) = \begin{cases} 0 & \text{se} \quad n = 0 \\ T(n-1) + 3 & \text{se} \quad n > 0 \end{cases}$$

considerando que são 2 multiplicações e 1 adição feitas e mais o custo da chamada recursiva que tem tamanho $n - 1$. Resolvendo a recorrência chega-se que $T(n) = 3n$ ou seja, sendo portanto um algoritmo de complexidade linear.

Projeto por Indução - exemplo (3ª abordagem)

3ª abordagem: considere novamente o mesmo problema (Cálculo de Polinômios), mas usando:

- hipótese de indução: suponha que sabe-se calcular $P'_{n-1}(x) = a_n x^{n-1} + a_{n-1} x^{n-2} + ... + a_1$.
- passo da indução: assim $P_n(x)$ pode ser calculado como $P_n(x) = xP'_{n-1}(x) + a_0$.

Projeto por Indução - exemplo (3ª abordagem)

Essa nova indução conduz ao seguinte algoritmo recursivo:

```
1:  function CalculaPolinomio(A, x)
2:      ▷ Entrada: coefecientes A = a_n, a_{n-1}, ..., a_1, a_0 e real x
3:      ▷ Saída: P_n(x)
4:      if n = 0 then
5:          return a_0
6:      else
7:          A' ← a_n, a_{n-1}, ..., a_1
8:          P' ← CalculaPolinomio(A',x)          ▷ Cálculo de P_{n-1}(x)
9:          P ← x * P' + a_0
10:         return P
11:     end if
12: end function
```

Projeto por Indução - exemplo (3ª abordagem)

Calculando a nova complexidade de tempo, obtém-se a recorrência:

$$T(n) = \begin{cases} 0 & \text{se} \quad n = 0 \\ T(n-1) + 2 & \text{se} \quad n > 0 \end{cases}$$

considerando que são 1 multiplicação e 1 adição feitas e mais o custo da chamada recursiva que tem tamanho $n - 1$. Resolvendo a recorrência chega-se que $T(n) = 2n$ ou seja, sendo portanto um algoritmo de complexidade linear, com um ganho na complexidade em relação ao algoritmo anterior.

Observe que foi possível utilizar indução de 3 maneiras diferentes de indução para o mesmo problema, cada uma gerando um algoritmo de complexidade diferente.

Projeto por Indução - exemplo 2

Problema do Cálculo da Altura de Árvores Binárias

Dada uma árvore binária A, calcular sua altura h.

Lembrete: A altura de uma árvore binária corresponde ao número de níveis existentes abaixo da raiz.

Projeto por Indução - Altura de árvore binária

Considerando uma árvore binária de n nós, o projeto por indução pode ser aplicado da seguinte maneira:

- caso base: $n = 0$. Nesse caso, $h = 0$
- hipotése de indução: sabe-se calcular as alturas das subárvores esquerda (h_e) e direita (h_d) de uma árvore binária A com $n > 0$ nós.
- passo da indução: a altura da árvore A pode ser calculada por $h_A = max(h_e, h_d) + 1$

Projeto por Indução - Altura de árvore binária

Essa indução conduz ao seguinte algoritmo recursivo:

```
1:  function CalculaAltura(A)
2:      ▷ Entrada: árvore binária A
3:      ▷ Saída: altura de A
4:      if n = 0 then
5:          return 0
6:      else
7:          h_e ← CalculaAltura(A.esq)          ▷ Cálculo de h_e
8:          h_d ← CalculaAltura(A.dir)          ▷ Cálculo de h_d
9:          if h_e > h_d then
10:             return h_e + 1
11:         else
12:             return h_d + 1
13:         end if
14:     end if
15: end function
```

Projeto por Indução - Altura de árvore binária

Calculando a complexidade de tempo, que pode ser expressa na seguinte recorrência:

$$T(n) = \begin{cases} 0 & \text{se} \quad n = 0 \\ T(n_e) + T(n_d) + \Theta(1) & \text{se} \quad n > 0 \end{cases}$$

onde:

- n_e e n_d são os números de nós das subárvores esquerda e direita, respectivamente
- $\Theta(1)$ corresponde às operações de atribuição de valores a h_e e h_d, e também a execução do comando *if*

Projeto por Indução - Altura de árvore binária

O pior caso do algoritmo ocorre quando a cada nível que se desce na árvore, sempre uma das subárvores é vazia, caso em que o número de nós da subárvore não vazia é $n-1$ e $T(n)$ pode ser escrita como $T(n) = T(n-1) + \Theta(1)$. Assim, $T(n) = \sum_{i=1}^{n} \Theta(1) = \Theta(n)$. E portanto, o algoritmo para o pior caso tem comportamento linear.

Projeto por Indução - Erros comuns

Da mesma forma que na demonstração por indução, erros análogos podem ser cometidos ao usar a técnica de projeto por indução:

- esquecer o caso base: no caso de algoritmos recursivos, a condição de parada é fundamental para o funcionamento correto do algoritmo.
- generalizar uma solução para o tamanho de entrada n a partir de uma instância específica (caso especial) ao invés de uma instância arbitrária

Projeto por Divisão e Conquista

Técnica de projeto de algoritmos inspirada em tática de guerra, com as seguintes características:

- divide o problema original em subproblemas da mesma natureza
- resolve o problema a partir da combinação de soluções parciais de um ou mais subproblemas, que são obtidos recursivamente
- baseado no princípio da indução
- gera algoritmos recursivos
- sua diferença para o projeto por indução simples está na forma da indução: na indução simples o algoritmo funciona de maneira incremental (a partir da retirada ou inclusão de um único elemento) enquanto na divisão e conquista o conjunto de entrada inicial é repartido em tamanhos menores.

Projeto por Divisão e Conquista

Esquematicamente um algoritmo usando a técnica de Divisão e Conquista tem o seguinte esquema:

```
1:  function DivisaoeConquista(x)
2:      ▷ Entrada: instância x do problema de tamanho n
3:      ▷ Saída: solução y para instância x
4:      if n é pequeno then                                   ▷ caso base
5:          return SolucaoCasoBase(x)
6:      else
7:          ▷ Divisão
8:          Divida a instância x de tamanho n em k instâncias menores x_1, x_2, ..., x_k
9:          for i = 1, ..., k do
10:             y_i ← DivisaoeConquista(x_i)
11:         end for
12:         ▷ Conquista
13:         y ← CombinaSolucoes(y_1, y_2, ..., y_k)
14:         return y
15:     end if
16: end function
```

Projeto por Divisão e Conquista

Em um projeto por Divisão e Conquista:

- os subproblemas são sempre indepentendentes, isto é, não há sobreposição entre suas subsoluções
- em geral é obtido um algoritmo simples, claro, robusto e elegante
- dependendo do tamanho da entrada do problema e da pilha de chamadas recursivas geradas pode ocorrer queda no desempenho ou estouro de pilha durante a execução do algoritmo

Projeto por Divisão e Conquista

A complexidade de tempo de um projeto de Divisão e Conquista pode ser expressa pela recorrência:
$T(n) = aT(\frac{n}{b}) + f(n)$
onde:

- a corresponde ao número de chamadas recursivas do algoritmo
- b corresponde ao tamanho das instâncias dos subproblemas
- $f(n)$ corresponde ao custo da combinação das subsoluções para compor a solução global

Projeto por Divisão e Conquista - Busca Binária

A Busca Binária é um exemplo de projeto por divisão e conquista.

Problema de determinar a posição de um elemento em um vetor ordenado

Dado um vetor ordenado $vet[0..n-1]$ com n números reais e um real x, determinar a posição i, tal que $0 \leq i \leq n-1$ e $vet[i] = x$ ou que o elemento não pertence ao vetor.

Projeto por Divisão e Conquista - Busca Binária

Como o vetor está ordenado, pode-se usar o seguinte argumento indutivo: é possível descartar a análise da metade dos elementos apenas a partir da comparação com o elemento localizado na posição central. Caso seja o elemento procurado seja maior, procura-se apenas na segunda metade; caso seja menor, procura-se apenas na primeira metade.

Projeto por Divisão e Conquista - Busca Binária

```
1:  function BuscaBinaria(vet, e, d, x)
2:      ▷ Entrada: vetor ordenado vet[0..n − 1], e e d são índices de esquerda e
    direita respectivamente, e x é o elemento procurado
3:      ▷ Saída: posição i tal que vet[i] = x ou −1 caso x não pertença ao vetor
4:      if e = d − 1 then                ▷ caso base: um único elemento a ser analisado
5:          if vet[d] = x then
6:              return d
7:          else
8:              return −1                              ▷ elemento não encontrado
9:          end if
10:     else
11:         m ← (e + d)/2                              ▷ cálculo da posição central
12:         if vet[m] < x then
13:             return BuscaBinaria(vet,e,m,x)
14:             return BuscaBinaria(vet,m,d,x)
15:         end if
16:     end if
17: end function
```

Projeto por Divisão e Conquista - Busca Binária

Implementado da forma anterior, o algoritmo considera que o vetor se inicia na posição 0 (zero), e a chamada inicial do algoritmo anterior deve ser $BuscaBinaria(vet, -1, n, x)$, isto é, os índices esquerda (-1) e direita (n) estão fora dos limites do vetor.
No caso da Busca Binária, não há necessidade de combinação de subsoluções, pois a resposta corresponde diretamente ao resultado da chamada recursiva executada.

Projeto por Divisão e Conquista - Busca Binária

A recorrência que expressa a complexidade do algoritmo é:

$$T(n) = \begin{cases} \Theta(1) & \text{se} \quad n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{se} \quad n > 1 \end{cases}$$

pois:

- quando há um único elemento a ser analisado ($n = 1$), corresponde ao custo do *if* e uma comparação, sendo portanto $\Theta(1)$
- apesar de existirem 2 chamadas recursivas no algoritmo, apenas uma delas é realmente invocada, dependendo se $vet[m] < x$ ou não. E além disso, o tamanho do conjunto a ser analisado reduze-se à metade do original. Assim $T(\frac{n}{2}) + \Theta(1)$ representa o custo da chamada recursiva e o custo do teste $vet[m] < x$

Resolvendo a recorrência

$$T(n) = \begin{cases} \Theta(1) & \text{se} \quad n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{se} \quad n > 1 \end{cases}$$

usando o método Mestre, obtém-se que $T(n) \in \Theta(lg\, n)$, e portanto, o algoritmo da busca binária tem complexidade logaritmica.

Projeto por Divisão e Conquista - Mergesort

O algoritmo de ordenação *Mergesort* é um exemplo típico da estratégia de divisão e conquista:

1. divida o vetor a ser ordenado em 2 metades, e as ordene separadamente.
2. depois faça a interlação das metades ordenadas para obter o vetor completamente ordenado (conquista)

Projeto por Divisão e Conquista - Mergesort

```
1:  procedure MERGESORT(vet, e, d)
2:      ▷ Entrada: vetor vet[0..n − 1], e e d são índices de esquerda e direita
    respectivamente
3:      ▷ Saída: vetor ordenado crescentemente
4:      if e < d − 1 then
5:          m ← (e + d)/2                          ▷ cálculo da posição central
6:          MergeSort(vet, e, m)
7:          MergeSort(vet, m, d)
8:          Intercala(vet, e, m, d)
9:      end if
10: end procedure
```

Projeto por Divisão e Conquista - Mergesort

```
1:  procedure INTERCALA(vet, e, m, d)
2:      ▷ Entrada: vet[e..m..d], estando ordenados vet[e..m] e vet[m..d]
3:      ▷ Saída: vetor vet[e..d] ordenado crescentemente
4:      i ← e, j ← m, k ← 0
5:      while i < m and j < d do
6:          if vet[i] ≤ vet[j] then          ▷ copia para vetor auxiliar intercalando
7:              vet_aux[k] ← vet[i] , i ← i + 1
8:          else
9:              vet_aux[k] ← vet[j] , j ← j + 1
10:         end if
11:         k ← k + 1
12:     end while
13:     while i < m do          ▷ copia restante da 1ª metade caso tenha sobrado
14:         vet_aux[k] ← vet[i]
15:         k ← k + 1 , i ← i + 1
16:     end while
17:     while j < d do          ▷ copia restante da 2ª metade caso tenha sobrado
18:         vet_aux[k] ← vet[j]
19:         k ← k + 1, j ← j + 1
20:     end while
21:     for i = e, ..., d − 1 do          ▷ copia de volta para vetor original
22:         vet[i] ← vet_aux[i − e]
23:     end for
24: end procedure
```

Projeto por Divisão e Conquista - Mergesort

A recorrência que expressa a complexidade do algoritmo do *Mergesort* é:

$$T(n) = \begin{cases} 0 & \text{se} \quad n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{se} \quad n > 1 \end{cases}$$

pois:

- quando há um único elemento ($e < d - 1$), não há nada ser feito
- são feitas 2 chamadas recursivas, cada uma lidando com aproximadamente a metade do tamanho orginal, dependendo apenas se n é par ou ímpar.
- a complexidade do procedimento `Intercala` é $\Theta(n)$, pois apesar de ter vários laços *while*, eles fazem conjuntamente a cópia dos n elementos para o vetor auxiliar, e depois o último *for* copia-os de volta, por cima dos elementos no vetor original.

Projeto por Divisão e Conquista - Mergesort

Considerando que as metades do vetor são iguais, a recorrência *Mergesort* pode ser escrita por:

$$T(n) = \begin{cases} 0 & \text{se} \quad n = 1 \\ 2\,T(\frac{n}{2}) + \Theta(n) & \text{se} \quad n > 1 \end{cases}$$

que resolvendo pelo método Mestre, obtém que $T(n) \in \Theta(n\,lg\,n)$.

Projeto por Divisão e Conquista - Exponenciação

Problema do Cálculo de Exponenciação

Dado real a e inteiro $n \geq 0$, calcular a^n

O algoritmo recursivo obtido a partir de um projeto por indução simples é:

```
1: function EXPONENCIACAO(a, n)
2:     ▷ Entrada: real a e inteiro n ≥ 0
3:     ▷ Saída: a^n
4:     if n = 0 then
5:         return 1
6:     else
7:         return a * Exponenciacao(a, n − 1)
8:     end if
9: end function
```

Projeto por Divisão e Conquista - Exponenciação

O algoritmo por indução simples tem a complexidade expressa por $T(n) = T(n-1) + 1$, ou seja, $T(n) \in \Theta(n)$.

Dividir e conquistar

Será possível aplicar a estratégia de divisão e conquista ao problema do cálculo da exponenciação???

Projeto por Divisão e Conquista - Exponenciação

Aplicando a técnica de divisão e conquista, a indução poderia ser feita da seguinte maneira:

- caso base: $n = 0$, a solução é $a^n = 1$
- hipótese de indução: sabe-se calcular a^k para qualquer $k < n$.
- passo da indução: particularmente sabe-se calcular a^k para $k = \lfloor \frac{n}{2} \rfloor$. Assim

$$a^n = \begin{cases} (a^{\lfloor \frac{n}{2} \rfloor})^2 & \text{se} \quad n \text{ par} \\ a \times (a^{\lfloor \frac{n}{2} \rfloor})^2 & \text{se} \quad n \text{ ímpar} \end{cases}$$

Projeto por Divisão e Conquista - Exponenciação

A indução anterior conduz ao seguinte algoritmo recursivo:

```
1:  function ExponenciacaoDivConq(a, n)
2:      ▷ Entrada: real a e inteiro n ≥ 0
3:      ▷ Saída: a^n
4:      if n = 0 then
5:          return 1
6:      else
7:          aux ← ExponenciacaoDivConq(a, n div 2)        ▷ Calcula a^⌊n/2⌋
8:          aux ← aux × aux
9:          if (n mod 2) = 1 then                          ▷ n é ímpar
10:             aux ← a × aux
11:         end if
12:         return aux
13:     end if
14: end function
```

Projeto por Divisão e Conquista - Exponenciação

A recorrência que expressa a complexidade do algoritmo de divisão e conquista da exponenciação é:

$$T(n) = \begin{cases} 0 & \text{se} \quad n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + \Theta(1) & \text{se} \quad n > 0 \end{cases}$$

pois:

- $n = 0$, a solução é direta ($a^n = 1$)
- há uma única chamada recursiva sobre $\lfloor \frac{n}{2} \rfloor$.
- as operações de multiplicação e o teste de se n é ímpar, caso em que é feita uma multiplicação adicional, tem custo $\Theta(1)$

Resolvendo a recorrência pelo método Mestre, obtém-se que $T(n) \in \Theta(lg\ n)$, e portanto, o algoritmo tem complexidade de tempo logaritmica.

Bibliografia I

[Cormen 1997] Cormen,T.; Leiserson, C.; Rivest, R.
Introduction to Algorithms. McGrawHill, New York, 1997.

[Dasgupta 2009] Sanjoy Dasgupta; Christos Papadimitriou; Umesh Vazirani.
Algoritmos. McGrawHill, São Paulo, 2009.

[Feofiloff 2009] Paulo Feofiloff.
Algoritmos em linguagem C. Elsivier, Rio de Janeiro, 2009.

[Manber 1989] Udi Manber.
Introduction to Algorithms: A Creative Approach.
Addison-Wesley, 1989.

[Ziviani 2011] Nivio Ziviani.
Projeto de Algoritmos: com implementação em Java e C++.
Cengage Learning, São Paulo, 2011.