AED1 - Aula 04
Recursão, exponencial, análise de desempenho

Exponencial

Projetar um algoritmo que recebe inteiros positivos k e n e devolve k^n.

Como k^n = k * k * k * … * k (n vezes) podemos projetar o seguinte algoritmo iterativo:

```
int expI(int k, int n)
{
   int i, x = 1;
   for (i = 0; i < n; i++)
       x *= k;
   return x;
}
```

Também podemos descrever k^n usando a seguinte regra recursiva:
k^n = { k * k^(n-1), se n > 0,
0, se n = 1. }
Essa regra induz o seguinte algoritmo recursivo:

```
int expR(int k, int n)
{
   if (n == 0)
       return 1;
   return k * expR(k, n - 1);
}
```

Corretude:

- No caso de expR, como é comum com algoritmos recursivos, sua corretude deriva diretamente da implementação da regra em que este se baseia e pode ser verificada usando um prova por indução.
- Já a corretude do algoritmo iterativo expI depende da correta identificação de seu invariante de laço.
    - Neste caso, o invariante x = k^i vale no início de cada iteração.
    - Demonstração por indução:
        - Primeiro verificamos que o invariante vale no início da primeira iteração (caso base).
            - antes da primeira iteração temos x = 1 e i = 0. Como x = 1 = k^0 = k^i, o invariante vale.
        - Então supomos que o invariante vale no início de uma iteração qualquer, ou seja, x = k^i, e mostramos que no início da iteração seguinte ele continua valendo.

            - chamamos de x' e i' as variáveis no início da próxima iteração. Pelo comportamento do algoritmo dentro do laço, temos
                - x' = x * k = k^i * k = k^(i+1)
                - i' = i + 1
            - Portanto, x' = k^(i+1) = k^i', e o invariante vale.
    - Com a validade do invariante demonstrada podemos verificar a corretude do algoritmo, pois no fim do laço temos
        - i = n
    - pelo invariante x = 2^i = 2^n, portanto o algoritmo devolve o resultado correto.

Eficiência:

- De modo complementar, analisar a eficiência de algoritmos iterativos costuma ser fácil, bastando contar o número de vezes que as operações dentro dos laços são realizadas.
    - No caso do algoritmo expI temos apenas um laço que executa n vezes
    - e as operações dentro deste levam tempo constante
    - portanto, seu número de operações é proporcional a n.
- Já o cálculo da eficiência de um algoritmo recursivo depende da resolução de uma função de recorrência que captura a ordem do número de operações que tal algoritmo realiza
    - No caso do algoritmo expR temos a recorrência
        - T(n) = T(n-1) + 1, já que cada chamada da função desencadeia apenas uma outra chamada, com n decrementado de 1, e realiza uma quantidade constante de operações localmente.
        - T(0) = 1, pois o caso base da função recursiva ocorre quando n = 0.
    - Identificada a recorrência, podemos resolvê-la por substituição:

      T(n) = T(n-1) + 1

      (cálculos auxiliares)
      T(n-1) = T(n-2) + 1
      T(n-2) = T(n-3) + 1
      T(n-3) = T(n-4) + 1

      (desenvolvendo)
      T(n) = T(n-1) + 1
      = (T(n-2) + 1) + 1 = T(n-2) + 2
      = (T(n-3) + 2) + 1 = T(n-3) + 3
      = (T(n-4) + 3) + 1 = T(n-4) + 4
      …

(generalizando a regra)
T(n) = T(n-i) + i

(como T(0) = 1, para fazer n-i = 0 escolhemos i = n)
T(n) = T(n-(n)) + n
= T(0) + n
= 1 + n

- portanto, o número de operações realizadas por expR(n) é de ordem n.

Nossos dois algoritmos levam tempo proporcional a n. Será que conseguimos fazer melhor?

- Pensando no caso especial em que n é múltiplo de 2 podemos calcular k^n com a seguinte regra:
  - k^n = k^(n/2) * k^(n/2)
  - note que os dois termos multiplicados são iguais, o que sugere um algoritmo recursivo que realiza apenas uma chamada recursiva, com n dividido por 2, e depois multiplica o resultado desta chamada.
- No entanto, o algoritmo tem que tomar algum cuidado quando n não é par. Considerando que a operação n/2 corresponde ao piso da divisão, para n ímpar temos a regra:
  - k^n = k^(n/2) * k^(n/2) * k
- Por fim, o caso base continua ocorrendo quando n = 0.
- Assim, temos o algoritmo:

```
int expR2(int k, int n)
{
   int x;
   if (n == 0)
       return 1;
   x = expR2(k, n / 2);
   x *= x;
   if (n % 2 == 0)
       return x;
   return k * x;
}
```

A corretude deste algoritmo deriva diretamente da corretude da regra que o motivou. Já sua eficiência guarda uma grata surpresa, que deriva do fato dele dividir n por 2 a cada chamada recursiva. Para calcular a ordem do número de operações que ele realiza vamos usar a recorrência:

- T(n) = T(n/2) + 1, já que cada chamada da função desencadeia apenas uma outra chamada, com n dividido por 2.
- T(0) = 1.

- Resolução da recorrência por substituição:
  T(n) = T(n/2) + 1

  (cálculos auxiliares)
  T(n/2) = T(n/4) + 1
  T(n/4) = T(n/8) + 1
  T(n/16) = T(n/32) + 1

  (desenvolvendo)
  T(n) = T(n/2) + 1
  = (T(n/4) + 1) + 1 = T(n/4) + 2
  = (T(n/8) + 2) + 1 = T(n/8) + 3
  = (T(n/16) + 3) + 1 = T(n/16) + 4
  = (T(n/32) + 4) + 1 = T(n/32) + 5

  (sucessivas divisões por 2 nos sugerem ficar atento ao aparecimento de funções logarítmicas e exponenciais. Note que)
  T(n) = T(n/2) + 1 = T(n/2^1) + 1
  = T(n/4) + 2 = T(n/2^2) + 2
  = T(n/8) + 3 = T(n/2^3) + 3
  = T(n/16) + 4 = T(n/2^4) + 4
  = T(n/32) + 5 = T(n/2^5) + 5
  …
  (generalizando a regra)
  T(n) = T(n/2^i) + i

  (como T(0) = 1, para fazer n/2^i = 0, lembrando que se trata do piso da divisão, escolhemos i = lg n + 1)
  T(n) = T(n/2^(lg n + 1)) + (lg n + 1)
  = T(n/2n) + lg n + 1
  = T(0) + lg n + 1
  = 1 + lg n + 1
  = lg n + 2
- portanto, o número de operações realizadas por expR2(n) é de ordem lg n.
  - note que este algoritmo é muito mais eficiente que os anteriores para valores grandes de n.

Por fim, fica o seguinte exercício:
- projetar um algoritmo iterativo que seja igualmente eficiente para o cálculo da exponencial.

Comentários sobre corretude e eficiência de algoritmos

Corretude de algoritmos:

- Em algoritmos recursivos costuma ser uma análise mais direta, bastando:
  - mostrar que o algoritmo devolve o valor correto no caso base,
  - supondo que o algoritmo encontra o valor correto quando a entrada é menor que n, mostrar que ele devolve o valor correto para n.
- Em algoritmos iterativos geralmente é necessário utilizar invariantes:
  - um invariante é uma relação ou propriedade que depende dos valores das variáveis do algoritmo e se mantém válida ao longo das iterações de um laço.
  - encontrando os invariantes corretos, a análise de corretude é semelhante a de algoritmos recursivos.

Eficiência de algoritmos:

- Em algoritmos iterativos costuma ser bem direta, bastando somar operações realizadas dentro de um laço e multiplicar o resultado pelo número de iterações do laço.
- Em algoritmos recursivos geralmente é necessário usar uma fórmula de recorrência:
  - Por exemplo, a recorrência $T(n) = T(n-1) + 1$ e $T(0) = 1$, captura a ordem de operações realizada por um algoritmo que, em cada chamada recursiva, realiza um número constante de operações e faz uma chamada recursiva com a entrada reduzida de uma unidade, e que faz um número constante de operações quando atinge o caso base n=0.
  - Outro exemplo, a recorrência $T(n) = 2T(n/2) + 1$ e $T(1) = 1$, captura a eficiência de um algoritmo que, em cada chamada recursiva, realiza um número constante de operações e faz duas chamadas recursivas com a entrada reduzida pela metade, e que faz um número constante de operações no caso base n=1.
  - De posse da recorrência, pode-se simular seu comportamento, para tentar encontrar uma função (n, n^2, log n, 2^n, etc) que descreva bem o comportamento da recorrência conforme varia o tamanho da entrada n.
  - Esta função descreve a ordem do número de operações realizada pelo algoritmo recursivo.