Complexidade de Algoritmos

Paulino Ng

2020-04-03

Plano da aula

Esta aula apresenta a análise assintótica de alguns algoritmos simples.

1. Análise do algoritmo de impressão recursiva dos valores de uma lista encadeada.
2. Busca em um vetor aleatório
3. Busca em um vetor ordenado
4. Divisão e conquista: soluções recursivas. Equações de recorrência
5. Teorema Mestre para funções recursivas

Exercício discursivo ENADE-2017

Listas lineares armazenam uma coleção de elementos. A seguir, é apresentada a declaração de uma lista simplesmente encadeada.

```
struct ListaEncadeada {
  int dado;
  struct ListaEncadeada *proximo;
};
```

Para imprimir os seus elementos da cauda para a cabeça (do final para o início) de forma eficiente, um algoritmo pode ser escrito da seguinte forma:

```
void mostrar(struct ListaEncadeada *lista) {
  if (lista != NULL) {
    mostrar(lista->proximo);
    printf("%d ", lista->dado);
  }
}
```

Com base no algoritmo apresentado, faça o que se pede nos itens a seguir:

a) Apresente a classe de complexidade do algoritmo, usando a notação Θ.
b) Considerando que já existe implementada uma estrutura de dados do tipo pilha de inteiros - com as operações de empilhar, desempilhar e verificar pilha vazia - reescreva o algoritmo de forma não recursiva, mantendo a mesma complexidade. Seu algoritmo pode ser escrito em português estruturado ou em alguma linguagem de programação, como C, Java ou Pascal.

Solução

a) O algoritmo percorre a lista recursivamente, passando por cada nó uma única vez, logo, o algoritmo é: $\Theta(n)$

b) O "programa" para fazer o que é solicitado, faz um laço onde se desempilha um item, imprime, empilha numa pilha auxiliar. Estas operações são realizadas enquanto a pilha não estiver vazia. Após esvaziar a pilha original, deve-se fazer um novo laço para reempilhar os itens na pilha original.

Busca linear de um valor num vetor

Seja o código:

```
int busca(int *vi, int n, int val) {
  int i;
  for (i = 0; i < n; i++)
    if (vi[i] == val) return i;
  return -1;
}
```

Se o valor `val` fizer parte do vetor, a função retorna o índice do vetor onde está o valor, se não, a função retorna -1.

No melhor caso, o valor está na primeira posição: $f(n) = 1$
No pior caso, o valor está no fim do vetor ou não está no vetor: $f(n) = n$
No caso médio: $f(n) = \frac{n}{2}$
Ou seja, a complexidade do algoritmo é $\mathcal{O}(n)$

Busca de um valor num vetor ordenado (busca binária)

```
int busca2(int *vi, int n, int val) {
  int sup = n-1, inf = 0, meio;
  while (sup >= inf) {
    meio = (sup + inf)/2;
    if (vi[meio] == val) return meio;
    if (val > vi[meio]) inf = meio + 1;
    else                sup = meio - 1;
  }
  return -1;
}
```

A cada passagem no laço, o espaço de busca diminui pela metade.

O espaço fica reduzido a 1 elemento em $\log_2 n$ iterações. Logo:

No melhor caso, $f(n) = 1$

No pior caso, $f(n) = lg\ n$

O caso médio é próximo do pior caso, $f(n) = lg\ n$. *A complexidade é* $\mathcal{O}(lg\ n)$.

Divisão e conquista

Uma técnica comum para a resolução de problemas computacionais é chamada de *divisão e conquista*.

O problema é dividido em problemas menores recursivamente até que sejam tão pequenos que possamos calcular a solução.

Exemplo: cálculo do fatorial recursivo e cálculo do n-ésimo número da sequência de Fibonacci.

Problema do fatorial:

- Entrada: inteiro não negativo n
- Saída: 1 se $n = 0$ e $n.(n-1).....2.1 = n!$ se $n > 0$

Observando que: $n! = n.(n-1)!$ para n>0, podemos calcular o fatorial de forma recursiva.

Cálculo do fatorial

```
int fatorial(int n) {
  if (n == 0) return 1;
  return n * fatorial(n-1);
}
```

- O cálculo do fatorial acima usa a recursão. Podemos transformar a recursão num laço com:

```
int fato(int n) {
  int acc = 1, i;
  for (i = 2; i <= n; i++) acc *= i;
  return acc;
}
```

- A solução não recursiva, em geral, é mais rápida. Apesar de ambas terem a mesma ordem de complexidade, $\Theta(n)$, a recursão custa mais tempo de execução, o que aumenta as constantes no cálculo do tempo.

Algoritmos recursivos

- Muitos algoritmos úteis são recursivos na própria estrutura, por exemplo nos algoritmos para dados em árvore. Este algoritmos seguem a formulação *dividir-e-conquistar*: quebra-se o problema em subproblemas semelhantes ao problema original com tamanho menor e a composição das soluções resolvem o problema maior. Esta divisão dos problemas é feita recursivamente até se ter problemas básicos cuja solução possa ser facilmente calculada (sem novas quebras). A solução do problema original é obtida da composição das soluções básicas.
- O paradigma *dividir-e-conquistar* leva aos seguintes 3 passos:
  - **Dividir** o problema em vários (ou um) problemas menores do mesmo tipo;
  - **Conquistar** os subproblemas resolvendo-os recursivamente. Se o subproblema for pequeno o suficiente, resolvê-lo diretamente; e
  - **Combinar** as soluções dos subproblemas até resolver o problema original.

Teorema mestre para funções recursivas

Soluções recursivas frequentemente resultam na equação de recorrência a seguir no cálculo da complexidade do algoritmo.

Onde $a \geq 1$ e $b > 1$ são constantes e $f(n)$ é uma função assintoticamente positiva, $T(n)$ é uma medida de complexidade definida sobre os inteiros. A solução da equação de recorrência:

$T(n) = aT(\frac{n}{b}) + f(n)$,

É:

1. $T(n) = \Theta(n^{\log_b a})$, se $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$;
2. $T(n) = \Theta(n^{\log_b a} \log n)$, se $f(n) = \Theta(n^{\log_b a})$; e
3. $T(n) = \Theta(f(n))$, se $f(n) = \Omega(n^{\log_b a + \epsilon})$para alguma constante $\epsilon > 0$, e se $af(\frac{b}{b}) \leq cf(n)$ para alguma constante $c < 1$ e todo n a partir de um valor suficientemente grande.

- A equação de recorrência diz que o problema foi dividido em a subproblemas de tamanho $\frac{n}{b}$ cada um.
- Os problemas são resolvidos recursivamente em tempo $T(\frac{n}{b})$ cada um.
- A função $f(n)$ descreve o custo de dividir o problema em subproblemas e combinar os resultados de cada subproblema.

Busca binária recursiva

- Problema:
  - Entradas: vetor vi, índice inicial, índice final, valor buscado
  - Saída: índice do elemento que casa com o valor buscado, ou -1 se o valor não estiver no vetor.

```
int busca2_rec(int *vi, int ini, int fim, int val) {
  if (ini > fim) return -1;
  int meio = (ini + fim)/2;
  if (vi[meio] == val) return meio;
  if (val > vi[meio]) return busca2_rec(vi, meio + 1, fim,
  return busca2_rec(vi, ini, meio - 1, val);
}
```

Use o teorema mestre para mostrar que a complexidade da busca binária recursiva é $\Theta(\log n)$. Sem usar o teorema mestre, refaça a análise.

Recursividade vs. Iteratividade

- Na recursividade, a função se chama para resolver os problemas menores. Cada chamada de função cria um contexto na memória para calcular a função. Estes contextos só são recolhidos no final do cálculo. A quantidade de contextos cresce com a complexidade do algoritmo, na maioria das vezes. Isto faz com que, em geral, uma implementação recursiva seja mais lenta do que sua implementação iterativa. Além disso, muitas vezes ao transformar um algoritmo recursivo em iterativo, é possível modificá-lo em mudar sua complexidade.
- O caso mais típico é o cálculo do enésimo elemento da sequência de Fibonacci. O algoritmo recursivo:

```
int fibo_rec(int n) {
  if (n < 2) return n;
  return fibo_rec(n-1) + fibo_rec(n-2);
}
```

- Curto, elegante e que segue exatamente a definição matemática tem complexidade $\Theta(2^n)$.

Árvore de chamadas do fibo_rec()

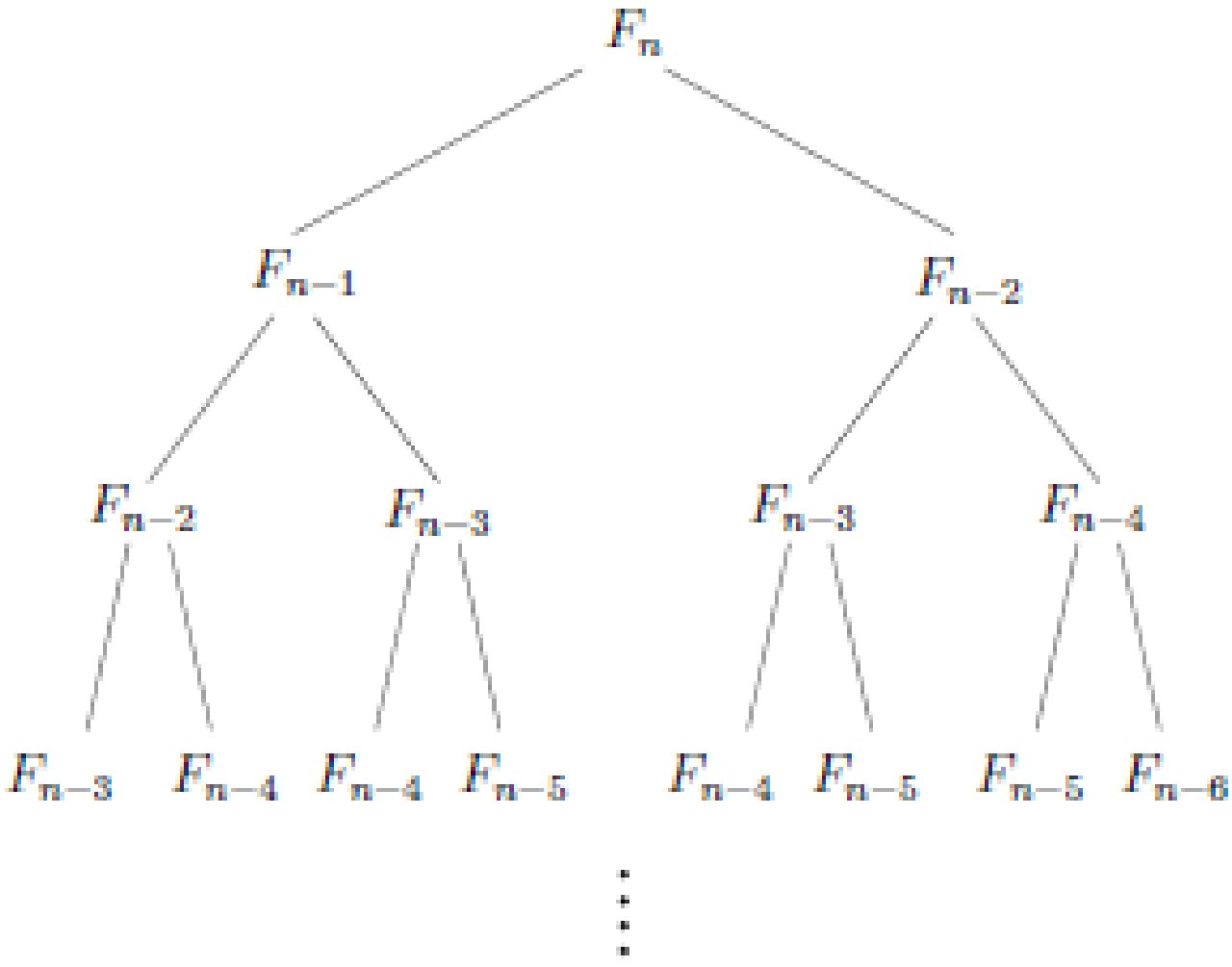

Figure 1: Árvore de chamadas

(cont.)

- É fácil reescrever de maneira iterativa este cálculo com:

```
int fibo_iter(int n) {
  if (n < 2) return n;
  int val1 = 1, val2 = 0, val;
  for ( ; n > 1; n--) {
    val = val1 + val2;
    val2 = val1;
    val1 = val;
  }
  return val;
}
```

- Este algoritmo iterativo tem complexidade $\Theta(n)$.

(cont.)

Para os alunos de programação funcional que possam achar que devido a isto, algoritmos iterativos são sempre melhores. Observe que é possível escrever um `fibo_rec()` com complexidade $\Theta(n)$.

```
int fibo_aux(int n, int val1, int val2) {
  if (n == 0) return 0;
  if (n == 1) return val1;
  return fibo_aux(n-1, val1+val2, val1);
}
int fibo_rec2(int n) {
  return fibo_aux(n, 1, 0);
}
```

Obs.: A função recursiva fibo_aux() tem a vantagem de ter recursividade terminal. Recursividade terminal ocorre quando a última instrução da função recursiva, a que faz o retorno, só faz a única chamada recursiva. Neste caso o compilador não precisa gerar código para uma chamada de função, ele pode fazer simplesmente um ´GOTO´ para a função. Evita-se deste modo a criação dos contextos da