Recursividade[1][2]

Algoritmos recursivos são parte do cotidiano da computação. Por um lado, a forma concisa de definir soluções em formato recursivo atrai o interesse de desenvolvedores interessados em evitar códigos iterativos complicados[3]. Por outro, a recursão é uma ferramenta que deve ser evitada em alguns contextos de aplicação, como discutiremos neste roteiro.

1 Fundamentos teóricos

Estrutura geral – os componentes principais de um algoritmo recursivo são as definições da **estrutura recursiva** e do(s) **caso**(s) **base**[4]. Em geral, a estrutura recursiva é definida em função de um caso anterior ou de um sub-caso. Respectivamente, os casos bases para estes dois modelos são o caso inicial (quando não existe mais caso anterior) ou o caso atômico (quando não é mais possível subdividir o problema).

Operacionalização – internamente, algoritmos recursivos são operacionalizados através do bloco denominado **pilha**, localizado no segmento de memória reservado pela aplicação para alocação dinâmica. A cada chamada, o estado das variáveis pertencentes ao escopo local da função é salvo, sendo recuperado quando os níveis recursivos abaixo do nível atual retornam. Por este motivo, é possível que algoritmos recursivos gerem um **estouro de pilha**[5], isto é, um uso excessivo de memória que extrapola o limite permitido pelo sistema operacional para cada aplicação.

Complexidade teórica – a análise da complexidade assintótica de algoritmos recursivos requer ferramentas matemáticas mais avançadas que a análise de algoritmos iterativos[6]. Em linhas gerais, algoritmos recursivos do modelo **dividir-para-conquistar** apresentam-se como os mais eficientes, enquanto algoritmos que apresentam **intersecção entre subproblemas** apresentam-se como os mais ineficientes.

2 Laboratório 1

Neste laboratório, você deverá implementar o algoritmo da busca sequencial de forma recursiva, respeitando o protótipo abaixo. Os arquivos relacionados a esta implementação terão como nome base o mesmo nome da função (ex.: `sequencial_recursiva.cpp`). Este trabalho deverá ser entregue até o fim da aula de hoje.

```
1 int sequencial_recursiva(int v[], int chave, int tamanho);
```

[1] Roteiro de estudo fornecido na disciplina de Estruturas de Dados Básicas 1 (EDB1) da Universidade Federal do Rio Grande do Norte (UFRN).

[2] Autor: Leonardo Bezerra (`leobezerra@imd.ufrn.br`).

[3] Em geral, é possível escrever versões iterativas de algoritmos recursivos, porém é comum que estas versões sejam de mais difícil entendimento que os algoritmos originais.

[4] Note que é possível utilizar funções recursivas sem caso base, porém sua utilidade é nula uma vez que este será um algoritmo infinito.

[5] As duas principais razões para estouros de pilha são algoritmos cuja profundidade se alonga bastante antes que um caso base seja atingido, ou algoritmos que alocam muita memória localmente a cada nível da recursão (muitas variáveis locais ou vetores alocados estaticamente demasiadamente grandes).

[6] Para algoritmos que trabalham com subcasos do problema original, é possível realizar a análise através do teorema central. Para casos mais simples, é possível realizar esta análise através dos métodos iterativo ou da árvore.

3 Visão crítica

Em geral, o uso de algoritmos recursivos deve ser ponderado em função da categoria na qual o algoritmo se encaixe:

- **Algoritmos dividir-para-conquistar** particionam o problema em subproblemas e, por vezes, apresentam ganhos em termos de complexidade assintótica. Em geral, algoritmos dividir-para-conquistar podem ser implementados de forma recursiva com mais simplicidade do que de forma iterativa. Um exemplo de algoritmo dividir-para-conquistar é a *busca binária*, que consegue apresentar complexidade assintótica reduzida em relação à *busca sequencial.*

- **Intersecções entre subproblemas** são a principal causa de ineficiência de algoritmos recursivos. Nestas situações, múltiplas chamadas da função recursiva para uma mesma entrada passam a ocorrer, tornando o algoritmo recursivo redundante e ineficiente. Um exemplo de algoritmo recursivo com intersecção entre subproblemas é o algoritmo para calcular *o n-ésimo termo da sequência de Fibonacci.* A alternativa mais eficaz para contornar este tipo de situação é o uso de *programação dinâmica.*

- **Recursão de cauda** é o termo utilizado para descrever algoritmos recursivos onde a chamada à função recursiva ocorre como último passo da função. Algoritmos recursivos deste tipo são considerados ineficientes, mas a tradução do código recursivo para o modelo iterativo é feita de forma automática por muitos compiladores. Assim, para se beneficiar desta situação, busca-se converter algoritmos recursivos que não apresentem recursão de cauda em versões que apresentem esta característica.

- **Demais casos** devem ser analisados individualmente. Em geral, a opção por implementações recursivas se dá quando a implementação iterativa é complicada, dificultando a compreensão e manutenção do código. No entanto, sempre que houver a possibilidade de escrever um código iterativo claro, este deve ser preferido ao código recursivo.

4 Laboratório 2

Neste laboratório, você deverá implementar uma versão recursiva do algoritmo de busca binária para sequências bitônicas. Especificamente, uma sequência bitônica é uma sequência de n elementos

$$\{A_1, A_2, \ldots, A_k, A_{k+1}, A_{k+2}, \ldots, A_n\} \tag{1.1}$$

tal que:

- a subsequência $\{A_1, A_2, \ldots, A_k\}$ está em ordem crescente;
- a subsequência $\{A_k, A_{k+1}, \ldots, A_n\}$ está em ordem decrescente.

A função implementada deverá respeitar o protótipo abaixo. Os arquivos relacionados a esta implementação terão como nome base o termo `bitonica` (ex.: `bitonica.cpp`). Note que, para implementar uma versão da busca binária de modo recursivo para sequências bitônicas, o primeiro passo é entender a implementação recursiva da busca binária para sequências tradicionais.

```
1 int binaria_bitonica(int v[], int chave, int tamanho);
```