Paradigmas de Projeto de Algoritmos

Recursividade, Tentativa e Erro, Divisão e Conquista

Referência: Projeto de Algoritmos, Nivio Ziviani, Capítulo 2

O projeto de algoritmos requer abordagens adequadas: a forma como um algoritmo aborda um problema pode levar a um desempenho ineficiente. Nosso objetivo é rever os principais paradigmas a serem seguidos durante o projeto de algoritmos.

Não existe um paradigma que seja o melhor entre todos. Diferentes paradigmas são adequados para diferentes problemas.

1. Recursividade

- Um procedimento que chama a si mesmo, direta ou indiretamente, é dito ser **recursivo**.
- Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.
- Exemplo: árvores binárias de pesquisa
    - Todos os registros com chaves menores estão na sub-árvore esquerda
    - Todos os registros com chaves maiores estão na sub-árvore direita
    - Caminhamento central é um procedimento recursivo

```
CENTRAL(p)
  if p != nil
    CENTRAL(p.esq)
    Visita nó     ->Faz algum processamento
    CENTRAL(p.dir)
```

Implementação de recursividade:

- Usa-se uma pilha para armazenar os dados usados em cada chamada de um procedimento que ainda não terminou;
- Todos os dados não globais vão para a pilha, registrando o estado corrente da computação;
- Quando uma ativação anterior prossegue, os dados da pilha são recuperados.
- No caso do caminhamento central:
    - Para cada chamada recursiva, o valor de p e o endereço de retorno da chamada recursiva são armazenados na pilha.
    - Quando encontra p=nil o procedimento retorna para quem chamou utilizando o endereço de retorno que está no topo da pilha.

Problema de terminação em procedimentos recursivos:

- Procedimentos recursivos introduzem a possibilidade de iterações que podem não terminar
    - Existe a necessidade de considerar o problema de terminação.
- É necessário mostrar que o nível mais profundo de recursão é finito, e também possa ser mantido pequeno, pois cada ativação recursiva usa uma parcela de memória para acomodar as variáveis.

Quando não usar recursividade:

- Nem todo problema de natureza recursiva deve ser resolvido com um algoritmo recursivo.
- Estes podem ser caracterizados pelo esquema P ≡ if B then (S, P).
- Tais programas são facilmente transformáveis em uma versão não recursiva P ≡ (x: = x_0; while B do S).
- Recursividade também é ruim quando causa o cálculo repetido de resultados.
- Exemplo: números de Fibonacci

```
Fibonacci(n)
    if n < 2
        return n
    else
        return Fibonacci(n-1) + Fibonacci(n-2)
```

Programa ineficiente. Recalcula o mesmo valor várias vezes.

```
Fibonacci(n)
    if (n == 0) return 0
    i = 1
    j = 1
    sum = 0
    for k = 3 até n
        sum = i + j
        i = j
        j = sum
    return j
```

Programa eficiente.

Recursão de cauda:

Encontra elemento em uma lista encadeada:

```
find(p, val)
    if (p == NULL) return NULL
    else if (p.key == val) return p
    else return find(p.next, val)
```

- Uma função é dita recursiva em cauda quando sua chamada recursiva é a última operação feita na função.
- Compiladores eficientes transformam automaticamente a recursão de cauda em versões iterativas.
- A recursão de cauda é uma técnica de recursão que faz menos uso de memória durante o processo de empilhamento, o que a torna mais rápida que a recursão comum.
    - na recursão comum é necessário guardar a posição do código onde foi feita a chamada

    - na recursão de cauda isso não é necessário: a chamada recursiva é a última operação da função
- O Fibonacci **não** é uma função recursiva em cauda! Ele ainda precisa realizar uma soma após o retorno da recursão.

Fatorial recursivo:

```
fat(n)
    if (n == 1)
        return 1
    else
        return n * fat(n - 1)
```

Fatorial recursivo em cauda:

```
tail_fat(n)
    return fat_aux(n, 1)

fat_aux(num, aux)
    if (num == 1)
        return aux
    else
        return fat_aux((num - 1), (aux * num));
```

Conclusão:

- Técnica bastante adequada para expressar algoritmos que são definidos recursivamente.
- No entanto, deve ser usada com muito cuidado.
- Na maior parte dos casos funciona mais como uma técnica conceitual do que uma técnica computacional.
- Ao se fazer a análise de um algoritmo recursivo, deve-se também analisar o crescimento da pilha.
- A recursão em cauda ajuda a melhorar o desempenho de algoritmos recursivos
    - especialmente últil em linguagens funcionais que não possuem estruturas de repetição

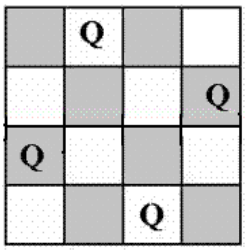
2. Tentativa e erro (backtracking)

- Tentativa e erro: decompor o processo em um número finito de sub-tarefas parciais que devem ser exploradas exaustivamente.
- O processo de tentativa gradualmente constrói e percorre uma árvore de sub-tarefas.
- Algoritmos tentativa e erro não seguem uma regra fixa de computação:
    - Passos em direção à solução final são tentados e registrados.
    - Caso esses passos tomados não levem à solução final, eles podem ser retirados e apagados do registro.

Exemplo: passeio do cavalo
- Tabuleiro com n × n posições: cavalo se movimenta segundo regras do xadrez.
- Problema: partindo da posição (x0, y0), encontrar, se existir, um passeio do cavalo que visita todos os pontos do tabuleiro uma única vez.

Exemplo: N rainhas
- Colocar n rainhas em um tabuleiro n x n, de forma que nenhuma rainha possa se atacar.

Solução: Vamos inserir as rainhas uma a uma, em diferentes colunas.
- 1) Se todas as rainhas estão inseridas a solução foi encontrada;
- 2) Insira uma rainha na próxima linha válida (começando a partir da primeira linha) da primeira coluna sem rainha e volte para o passo 1;
- 3) Se essa linha não existir, faça um backtrack (a última rainha inserida será colocada em outra linha);



- Técnica usada quando não se sabe exatamente que caminho seguir para encontrar uma solução.
- Não garante a solução ótima.
- Essa técnica pode ser vista ainda como uma variante da recursividade
- Ao se fazer a análise de um algoritmo que usa backtracking, deve-se também analisar o crescimento do espaço de soluções.