Classificação por Intercalação

Este é um bom exemplo de abordagem *top down*, ou de aplicação do princípio da *divisão e conquista*, associado à recursividade.

Ao se observar o andamento do processo sobre a lista, nota-se que a intercalação só começa quando as sublistas tornam-se unitárias. Até lá, *a posição relativa dos elementos não muda*. Ou seja: atinge-se uma situação de n sublistas unitárias (marcadas em itálico no exemplo anterior), cuja a concatenação é a lista origanal.

Classificação por Intercalação

E a intercalação começa juntando pares de elementos, a partir das listas unitárias. Ora, bem, no caso de um vetor, os elementos individuais já estão acessíveis. Logo, pode-se começar a ordenação com meio caminho andado (em relação às listas), intercalando trechos de *um* elemento, depois de *dois, quatro* e assim por diante. Cada trecho sob intercalação é dito “cadeia” ou “cordão” de intercalação. Veja o esquema (K é o comprimento da cadeia).

Classificação por Intercalação

Exemplo: Ordenação de vetor por intercalação

1º passo K = 1 **75 | 25 | 95 | 87 | 64 | 59 | 86 | 40 | 16 | 49**

2º passo K = 2 **25 75 | 87 95 | 59 64 | 40 86 | 16 49**

3º passo K = 4 **25 75 87 95 | 40 59 64 86 | 16 | 49**

4º passo K = cauda **16 25 40 49 59 64 75 86 87 95**

O problema divide-se, então, em duas partes:

Classificação por Intercalação

Para estabelecer as cadeias a intercalar, começa-se com tamanho K = 1. Na primeira passagem, formam-se cadeias de tamanho 2, depois de tamanho 4, 8, etc. Assim na primeira passagem, são intercaladas as partições V[0..0] e V[1..1], V[2..2] e V[3..3], V[4..4] e V[5..5] etc. Na segunda V[0..1] e V[2..3], V[4..5] e V[6..7], etc. Na terceira, V[0..3] e V[4..7], V[8..11] e V[12..15], etc. Regra geral, na passagem i, em que o tamanho da cadeia é $K = 2^{i-1}$, são intercalados os trechos V[j..j+k-1] e V[j+k..j+2*k-1].

Classificação por Intercalação

O problema da intercalação tem solução simples baseada no processo conhecido como *balance line*: percorre-se as duas cadeias a intercalar, usando um cursor para cada uma, copiando para um vetor-resposta sempre o menor elemento dentre os iniciais, avançando-se o cursor apenas da cadeia fornecedora. Ao se esgotar uma das cadeias, a outra é percorrida até o fim, preenchendo-se o vetor-resposta.

As questões adicionais que se colocam são:

1. Como salvar o resultado a cada passagem?

   Só o poderíamos fazer sobre o próprio vetor se fosse adotada uma solução recursiva, como no caso da lista. Mas isto de fato replicaria muitas vezes a área original, pelo salvamento cumulativo de versões parcialmente ordenadas. Melhor intercalar um vetor auxiliar, contendo uma cópia do vetor a ordenar, colocando o resultado no vetor original (vetor-resposta).

2. Como tratar um vetor com tamanho que não é potência de 2?

   Neste caso, sempre restará, ao final, uma cadeia ordenada de tamanho menor que o tamanho corrente da cadeia de intercalação. Esta cadeia é chamada cauda da intercalação (*merge tail*), e tem extensão dada por $n-2^{\lfloor \log_2 n \rfloor}$. No exemplo anterior, restou uma cauda de $10-2^{\lfloor \log_2 10 \rfloor} = 2$ elementos. Como última passagem, quando necessário, procede-se à interca-lação da cauda com o restante do vetor.

Classificação por Intercalação

Com base no que foi discutido, codifique uma função que receba um vetor (de inteiros) e o número de elementos no mesmo e através do método *merge sort* ordene de forma crescente os elementos do vetor.

```
void merge_sort (int v[], int n) {
   int v_aux[n], tam_cadeia=1, esq, dir, IND, i;
   while (tam_cadeia<=n/2) {
      for (i=0; i<n; v_aux[i]=v[i++]);
      esq=IND=0;
      dir=esq+tam_cadeia-1;
      while (dir<n-tam_cadeia) {
         intercala (v, v_aux, esq, dir, esq+tam_cadeia,
    dir+tam_cadeia);
         esq+=2*tam_cadeia;
         dir+=2*tam_cadeia; }
      tam_cadeia*=2; }
   for (i=0; i<n; v_aux[i]=v[i++]);
   if (tam_cadeia!=n)
      intercala (v, v_aux, 0, esq-1, esq, n-1); }
```

```
void intercala (int v[], int v_aux[], int limesqesq, int limesqdir,
int limdiresq, int limdirdir) {
   int deve_continuar=1, esq_menor, IND=limesqesq;
   while (deve_continuar)   {
      esq_menor=v_aux[limesqesq]<v_aux[limdiresq];
      v[IND++]=esq_menor?v_aux[limesqesq++]:
      v_aux[limdiresq++];
      deve_continuar=limesqesq<=limesqdir&&
      limdiresq<=limdirdir;
   }
   while (limesqesq<=limesqdir)
      v[IND++]=v_aux[limesqesq++];
   while (limdiresq<=limdirdir)
      v[IND++]=v_aux[limdiresq++];
}
```

Classificação por Intercalação

Quanto à complexidade do algoritmo apresentado no slide anterior, em uma análise superficial, pode ser determinada se considerarmos o seguinte: tam_cadeia, atualizada por duplicações sucessivas, assume valores do conjunto [1,2,4,8,16...$\lfloor n/2 \rfloor$], sendo a repetição principal controlada pela condição tam_cadeia <= n/2, o que a qualifica como O (log n). Em cada passagem, cada elemento do vetor é copiado uma vez e intercalando uma vez (na função intercala).

Classificação por Intercalação

O *enquanto* intermediário, i.e, o segundo *enquanto* do algoritmo principal, apenas distribui o processamento sobre os sucessivos subvetores. Isto acarreta na máximo 2n movimentos de dados em cada fase. Logo, o procedimento todo é da ordem de 2n log n, ou seja, O (n log n). Como uma análise mais profunda fugiria do escopo desta disciplina, ficaremos apenas neste nível de analise.

Classificação por Intercalação

É importante perceber que, quando o tamanho de uma lista *n* é pequeno, uma classificação **O**(n²) é em geral mais eficiente do que uma classificação **O**(n log n). Isto acontece porque usualmente as classificações **O**(n²) são muito simples de programar e exigem bem poucas ações além de comparações e trocas em cada passagem. Por causa dessa baixa sobrecarga, a constante de proporcionalidade é bem pequena. Em geral, uma classificação **O**(n log n) é muito complexa e emprega um grande

Classificação por Intercalação

número de operações adicionais em cada passagem para diminuir o número das passagens subseqüentes. Sendo assim, sua constante de proporcionalidade é maior. Quando *n* é grande, n^2 supera n log n, de modo que as constantes de proporcionalidade não desempenham um papel importante na determinação da classificação mais veloz. Entretanto, quando *n* é pequeno, n^2 não é muito maior que n log n de modo que uma grande diferença nessas constantes freqüentemente faz com que a classificação **O**(n^2) seja mais rápida.

Métodos de Pesquisa

Objetivos e Caracterizações

Para que se possa falar em algoritmos de pesquisa, é necessário inicialmente introduzir a noção de mapeamento que é uma das mais primitivas em programação. Refere-se a uma regra de associação entre os valores de um conjunto (domínio) e os valores de outro (imagem). Escreve-se

$$m: S \rightarrow T$$

para se declarar que m é um mapeamento do conjunto S para o conjunto T. Obtém-se um valor de T aplicando-se m(i), onde $i \in S$. Os vetores e matrizes são os casos típicos. Aí os índices são normalmente objetos simples ou no máximo tuplas homogenias (pares, triplas, etc) de valores simples.

Objetivos e Caracterizações

As estruturas chamadas **tabelas** são a realização da idéia genérica de mapeamento, em que os valores do domínio podem ser quaisquer. A organização das tabelas pode se reduzir aos arranjos, mas para se falar em tabelas, usa-se uma terminologia específica:

tabela: uma coleção de *entradas*;

entrada: um conjunto de *campos*, formando um *registro*, ou *linha*, da tabela;

chave: um campo escolhido para *identificar* a entrada.

Como pode-se perceber uma operação importante é a busca de uma entrada dado o valor da chave.

Objetivos e Caracterizações

A tabela, como mapeamento, poderia ser operada, por exemplo, para uma consulta, da seguinte forma:

E = T[C];

Considerando E: entrada, T:tabela; C:chave.

Diversas estratégias são propostas para implementação da operação de pesquisa, levando em conta aspectos das operações usuárias e da representação física da tabela. Veremos agora os dois métodos mais utilizados para implementação de pesquisa em tabelas, a pesquisa seqüencial e a pesquisa binária.

Pesquisa Seqüencial

A pesquisa seqüencial é o método mais simples. Consiste na mera varredura serial, entrada por entrada, devolvendo-se o índice da entrada cuja chave for igual à chave fornecida como argumento da pesquisa. Ou devolvendo -1, convencionalmente, caso a chave buscada não seja localizada, tendo-se comparado todas as chaves, até o fim da tabela. O algoritmo a seguir nos mostra uma função que efetua uma busca seqüencial em uma tabela.

```
int pesq (tabela T, chave C)
{
   int i;
   for (i=0; i<T.N; i++)
        if (T.TAB [i].CH == C)
                return i;
   retorne -1;
}
```

Pesquisa Seqüencial

A complexidade do algoritmo acima é **O**(n). O desempenho da pesquisa seqüencial pode melhorar um pouco se a tabela estiver *ordenad*a em função da chave: pode-se interromper a pesquisa assim que se alcançar uma entrada com chave maior do que a pesquisada, significando ser desnecessário prosseguir até o fim da tabela. O pior caso (busca da última chave) continua **O**(n). No algoritmo a seguir visualizamos um exemplo de uma função que efetua a busca seqüencial em uma tabela ordenada em função da chave.

```
int pesq_seq (tabela T, chave C)
{
  int i;
  for (i=0; i<T.N; i++)
     if (T.TAB [i].CH >= C)
          if (T.TAB [i].CH == C)
               return i;
          else
               retorne -1;
}
```

Pesquisa Binária

Sobre a tabela ordenada é possível obter-se um algoritmo bem mais eficiente. A idéia do mesmo pode ser assimilada pelo seguinte exemplo: ao procurar uma palavra no dicionário (que é uma tabela ordenada), começa-se em qualquer ponto do mesmo. Se a página aberta contiver a palavra, a busca terminou; senão, a palavra pode estar antes ou depois dessa página, conforme ela seja lexicograficamente menor ou maior que as palavras dessa página.

UNIVASF

Pesquisa Binária

De modo que novamente se abre o dicionário no setor adequado, anterior ou posterior, repetindo-se este processo até encontrar a palavra. No algoritmo apresentado agora, a primeira comparação de chaves é feita no meio da tabela. Se não for encontrada aí a chave procurada, pode-se abandonar metade da tabela, repetindo o processo com a divisão da outra metade pelo meio, até ser encontrada a chave ou ter-se uma metade constituída de apenas uma entrada, caracterizando-se assim a ausência da chave na tabela.

```
int pesq_bin (tabela T, chave C)  {
    int meio, PRIM, ULT, achou;
    PRIM = 0; ULT = T.N - 1; achou = 0;
    while (PRIM <= ULT && !achou) {
        meio = (ULT + PRIM) / 2;
        if (C== T.TAB [meio].CH)
            achou = 1;
        else
            if (C > T.TAB[meio].CH)
                PRIM = meio + 1;    //busca na parte final
            else
                ULT = meio- 1;      //busca na parte inicial
    }
     if (achou)
        return meio;
    else
        return- 1;
}
```

Pesquisa Binária

O algoritmo anterior nos mostra uma função que implementa a pesquisa binária de forma iterativa. Nota-se que, a cada comparação, o universo de chaves a comparar é reduzido à metade e que o pior caso é quando a busca prossegue até a subtabela pesquisada ter só um elemento (encontrando-se ou não a chave procurada). Para isso acontecer, o tamanho da tabela vai-ser reduzido de *n* para *n/2, (n/2)/2*, ... até *1*. Logo, a complexidade dessa solução é **O**(log n).

Pesquisa Binária - Exercício

Com base no algoritmo anterior, defina um TAD tabela e codifique um programa em C, o qual deve se utilizar da operação pesq_bin do TAD definido.

Pesquisa Binária

A solução recursiva é natural nesse caso, pois a metade da tabela é também uma tabela passível de operação estritamente similar à tabela inteira. Para o reaproveitamento recursivo da função, informa-se os índices PRIM e ULT, que se referem à primeira e à última entrada do trecho onde se efetua a busca. A invocação inicial deve ser pesq (T, C, 0, T.N-1). Como exercício, construa uma função recursiva que implemente a pesquisa binária.