Aula 18 – Algoritmos básicos de busca e classificação

Dentre os vários algoritmos fundamentais, os algoritmos de busca em tabelas e classificação de tabelas estão entre os mais usados. Considere por exemplo um sistema de banco de dados. As operações de busca para se encontrar um determinado elemento, são muito utilizadas. Para que a busca seja rápida é necessário que as tabelas que constituem o banco de dados estejam numa determinada ordem.

Vamos nos concentrar somente na busca e classificação de tabelas em memória, ou seja, busca e classificação em vetores ou matrizes.

Busca em tabelas

Busca sequencial

Consiste em varrer uma tabela a procura de um determinado elemento, verificando ao final se o mesmo foi ou não encontrado.

A função busca abaixo, procura elemento igual a `x` num vetor a de n elementos, devolvendo –1 se não encontrou ou o índice do primeiro elemento igual a `x` no vetor.

```
int busca(int a[], int n, int x) {
   int i;
   for (i = 0; i < n; i++)
     if (a[i] == x) return i;
   /* Foi até o final e não encontrou */
   return -1;
}
```

Existem várias maneiras de se fazer o algoritmo da busca, por exemplo:

```
int busca(int a[], int n, int x) {
   int i = 0;
   while (i < n && a[i] != x) i++;
   /* Verifica se parou porque chegou ao fim ou encontrou igual */
   if (i == n) return -1; /* chegou ao fim */
   return i; /* encontrou um igual */
  }
```

Fica como exercício, reescrever a função busca de outras formas. Usando o próprio comando `for`, usando o `do while`, usando o comando `while`, modificando a comparação, etc.

P102) Programa que testa a função `busca`. Dado `n`, gere uma seqüência de números com `rand()`, imprima a seqüência gerada e efetua várias buscas até ser digitado um número negativo.

```
#include <stdio.h>
#include <stdlib.h>
int busca(int a[], int n, int x) {
   int i;
   for (i = 0; i < n; i++)
     if (a[i] == x) return i;
   /* foi até o final e não encontrou */
   return -1;
}
```

```
void geravet(int v[], int k) {
  /* gera vetor com k elementos usando rand() */
   int i;
   srand(9999);
   for (i = 0; i < k; i++) v[i] = rand();
}

void impvet(int v[], int k)
  /* imprime vetor com k elementos */
  {int i;
   for (i = 0; i < k; i++) printf("%6d", v[i]);
  }

int main() {
  int vet[1000], n, kk, x;
  /* ler n */
  printf("entre com n:");
  scanf("%d", &n);
  /* gera o vetor e imprime */
  geravet(vet, n);
  impvet(vet, n);
  /* ler vários números até encontrar um negativo e procurar no vetor */
  printf("\nentre com o valor a ser procurado:");
  scanf("%d", &x);
  while (x >= 0) {
       if ((kk = busca(vet, n, x)) >= 0)
            printf("*** encontrado na posicao %3d", kk);
       else printf("*** nao encontrado");
       printf("\nentre com o valor a ser procurado:");
       /* lê o próximo */
       scanf("%d", &x);
  }
}
```

Veja o que será impresso:

```
entre com n:20
 32691  1995 26720 28855 23388 19105   163 24288 24657 28851  1499  9771
19920 1363 15073  4003 21514  5633 23815 29153
entre com o valor a ser procurado:21514
*** encontrado na posicao  16
entre com o valor a ser procurado:24288
*** encontrado na posicao   7
entre com o valor a ser procurado:163
*** encontrado na posicao   6
entre com o valor a ser procurado:7777
*** nao encontrado
entre com o valor a ser procurado:1234
*** nao encontrado
entre com o valor a ser procurado:1995
*** encontrado na posicao   1
entre com o valor a ser procurado:-9
```

Busca Sequencial – análise simplificada

Considere a função busca do P102. Quantas vezes a comparação **`(a[i] == x)`** é executada?

Essa comparação **`(a[i] == x)`** representa a quantidade de vezes que a repetição é efetuada. Dizemos então que o tempo que esse algoritmo leva para ser executado é proporcional a quantidade de vezes que essa repetição é executada.

Máximo = n (quando não encontra o x ou x é igual ao último elemento)
Mínimo = 1 (quando x é igual ao primeiro elemento)
Médio = (n+1)/2, supondo os elementos procurados equiprováveis

Assim, o tempo que esse algoritmo leva, é proporcional a n.

Quando existem elementos que podem ocorrer com maior freqüência, o melhor é colocá-los no início da tabela, pois serão encontrados com menos comparações.

Busca binária em tabela ordenada

Quando a tabela está ordenada, nem sempre é necessário ir até o fim da tabela para concluir que um elemento não está. Ao encontrarmos um elemento maior que o procurado, podemos parar a busca, pois dali para frente todos os demais serão maiores.

No entanto, existe um algoritmo muito melhor quando a tabela está ordenada. Lembre-se de como fazemos para procurar uma palavra no dicionário. Não vamos verificando folha a folha (busca seqüencial) até encontrar a palavra procurada. Ao contrário, abrimos o dicionário mais ou menos no meio e a partir daí só há 3 possibilidades:

a) Encontramos a palavra procurada na página aberta (ou concluímos que ela não está no dicionário, pois se estivesse estaria nesta página).
b) A palavra está na parte esquerda.
c) A palavra está na parte direita.

Se não ocorreu o caso a), repetimos o mesmo processo com as páginas remanescentes onde a palavra tem chance de ser encontrada, isto é, continuamos a busca com um número bem menor de páginas.

Para o caso de uma tabela, isso pode ser sistematizado da seguinte forma:

a) Testa com o elemento do meio da tabela
b) Se o elemento é igual, termina a busca, pois foi encontrado.
c) Se o elemento do meio é maior, repete o processo considerando a tabela do inicio até o elemento do meio menos 1.
d) Se o elemento do meio é menor, repete o processo considerando a tabela do elemento do meio mais 1 até o final.

```
int buscabinaria(int a[], int n, int x) {
   int inicio = 0, final = n-1, meio;
   /* procura enquanto a tabela tem elementos */
   while (inicio <= final) {
         meio = (inicio + final) / 2;
         if (a[meio] == x) return meio;
         if (a[meio] > x) final = meio -1; /* busca na parte de cima */
         else inicio = meio + 1; /* busca na parte de baixo */
   }
   /* foi até o final e não encontrou */
   return -1;
}
```

P102a) Programa que testa a função `buscabinaria`. Dado `n`, gere uma sequência **ordenada** de números com `rand()`, imprima a seqüência gerada e efetue várias buscas até ser digitado um número negativo. Acrescentamos um `printf` na `buscabinaria`, para explicitar cada uma das repetições.

```
#include <stdio.h>
#include <stdlib.h>
int buscabinaria(int a[], int n, int x) {
   int inicio = 0, final = n-1, meio;
   /* procura enquanto a tabela tem elementos */
   while (inicio <= final) {
         printf("\ninicio = %3d final = %3d", inicio, final);
         meio = (inicio + final) / 2;
         if (a[meio] == x) return meio;
         if (a[meio] > x) final = meio -1; /* busca na parte de cima */
         else inicio = meio + 1; /* busca na parte de baixo */
   }
   /* foi até o final e não encontrou */
   return -1;
}

void geravet(int v[], int k) {
  /* gera vetor em ordem crescente com k elementos usando rand() */
   int i;
   srand(9999); v[0] = rand()%100;
   for (i = 1; i < k; i++) v[i] = v[i-1] + rand()%100;
}

void impvet(int v[], int k) {
  /* imprime vetor com k elementos */
   int i;
   for (i = 0; i < k; i++) printf("%6d", v[i]);
}

int main() {
  int vet[1000], n, kk, x;
  /* ler n */
  printf("entre com n:");
  scanf("%d", &n);
  /* gera o vetor e imprime */
  geravet(vet, n);
  impvet(vet, n);
  /* ler vários números até encontrar um negativo e procurar no vetor */
  printf("\nentre com o valor a ser procurado:");
  scanf("%d", &x);
  while (x >= 0) {
       if ((kk = buscabinaria(vet, n, x)) >= 0)
            printf("\n*** encontrado na posicao %3d", kk);
       else printf("\n*** nao encontrado");
       printf("\nentre com o valor a ser procurado:");
       scanf("%d", &x);
  }
}
```

Veja o que será impresso. Em especial, veja como as variáveis `inicio` e `final` se comportam dentro da `buscabinaria`.

```
entre com n:15
    91   186   206   261   349   354   417   505   562   613   712   783
803 866   939
entre com o valor a ser procurado:354

inicio =   0 final =  14
inicio =   0 final =   6
inicio =   4 final =   6
*** encontrado na posicao   5
entre com o valor a ser procurado:803

inicio =   0 final =  14
inicio =   8 final =  14
inicio =  12 final =  14
inicio =  12 final =  12
*** encontrado na posicao  12
entre com o valor a ser procurado:500

inicio =   0 final =  14
inicio =   0 final =   6
inicio =   4 final =   6
inicio =   6 final =   6
*** nao encontrado
entre com o valor a ser procurado:613

inicio =   0 final =  14
inicio =   8 final =  14
inicio =   8 final =  10
*** encontrado na posicao   9
entre com o valor a ser procurado:12345

inicio =   0 final =  14
inicio =   8 final =  14
inicio =  12 final =  14
inicio =  14 final =  14
*** nao encontrado
entre com o valor a ser procurado:939

inicio =   0 final =  14
inicio =   8 final =  14
inicio =  12 final =  14
inicio =  14 final =  14
*** encontrado na posicao  14
entre com o valor a ser procurado:-1
```

Exercícios:

Considere a seguinte tabela ordenada, onde vamos procurar elementos usando a busca binária:

```
2 5 7 11 13 17 25
```

1) Diga quantas comparações serão necessárias para procurar cada um dos 7 elementos da tabela?

2) Diga quantas comparações serão necessárias para procurar os seguintes números que não estão na tabela 12, 28, 1, 75, 8?

E se a tabela tivesse os seguintes elementos:

```
2 5 7 11 13 17 25 32 35 39
```

1) Diga quantas comparações serão necessárias para procurar cada um dos 10 elementos da tabela?

2) Diga quantas comparações serão necessárias para procurar os seguintes números que não estão na tabela 12, 28, 1, 75, 8?

Busca binária - análise simplificada

A comparação `(a[meio] == x)` é bastante significativa para a análise do tempo que esse algoritmo demora, pois representa a quantidade de repetições que serão feitas.
O tempo consumido pelo algoritmo é proporcional à quantidade de repetições (comando **`while`**).
Como a cada repetição uma comparação é feita, o tempo consumido será proporcional à quantidade de comparações.

Quantas vezes a comparação `(a[meio] == x)` é efetuada?

Mínimo = 1 (encontra na primeira)
Máximo = ?
Médio = ?

Note que a cada iteração a tabela fica dividida ao meio. Assim, o tamanho da tabela é:
`N,N/2,N/4,N/8,...`
A busca terminará quando o tamanho da tabela chegar a zero, ou seja, no menor `k` tal que $N < 2^k$. Portanto o número máximo é um número próximo de `lg N` ($\lg N = \log_2 N$).

Uma maneira um pouco mais consistente. Vamos analisar quando `N` é da forma 2^k-1 (1,3,7,15,31,...)
Para este caso, a tabela fica sempre dividida em:
`N, N/2, N/4, N/8, ..., 15,7,3,1` – sempre com a divisão por 2 arredondada para baixo.
Serão feitas exatamente `k` repetições até que a tabela tenha `1` só elemento.
Como estamos interessados num limitante superior, caso N não seja desta forma, podemos considerar N o menor N' tal que N'>N e que seja da forma 2^k-1.

N = 2^k-1 então o número máximo de repetições será `k = lg (N+1)`.

Assim, o algoritmo é `O(lg N)`.

É um resultado surpreendente. Suponha uma tabela de `1.000.000` de elementos. O número máximo de comparações será `lg (1.000.001) = 20`. Compare com a busca seqüencial, onde o máximo de comparações seria `1.000.000` e o médio seria `500.000`. Veja abaixo alguns valores típicos para tabelas grandes.

N	lg (N+1)
100	7
1.000	10
10.000	14
100.000	17
1.000.000	20
10.000.000	24
100.000.000	27
1.000.000.000	30

Será que o número médio é muito diferente da média entre o máximo e o mínimo?
Vamos calculá-lo, supondo que sempre encontramos o elemento procurado. Note que quando não encontramos o elemento procurado, o número de comparações é igual ao máximo. Assim, no caso geral, a média estará entre o máximo e a média supondo que sempre vamos encontrar o elemento.

Supondo que temos N elementos e que a probabilidade de procurar cada um é sempre 1/N.
Vamos considerar novamente $N=2^k-1$ pelo mesmo motivo anterior.
Como fazemos 1 comparação na tabela de N elementos, 2 comparações em 2 tabelas de N/2 elementos, 3 comparações em 4 tabelas de N/4 elementos, 4 comparações em 8 tabelas de N/8 elementos, e assim por diante.

$= 1.1/N + 2.2/N + 3.4/N + ... + k.2^{k-1}/N$

$= 1/N \; . \; \sum i.2^{i-1} \; (i=1,k)$

$= 1/N \; . \; ((k-1).2^k + 1)$ (a prova por indução está abaixo)

Como $N=2^k-1$ então $k=\lg(N+1)$

$= 1/N \; . \; ((\lg(N+1) - 1) \; . \; (N+1) + 1)$

$= 1/N \; . \; ((N+1).\lg(N+1) - N)$

$= (N+1)/N \; . \; \lg(N+1) - 1 \sim \lg(N+1) - 1$

Resultado novamente surpreendente. A média é muito próxima do máximo.

Prova por indução: $\sum i.2^{i-1} \; (i=1,k) = (k-1).2^k + 1$

Verdade para $k=1$
Supondo verdade para k, vamos calcular para $k+1$.

$\sum i.2^{i-1} \; (i=1,k+1) = \sum i.2^{i-1} \; (i=1,k) + (k+1).2^k$
$= (k-1).2^k + 1 + (k+1).2^k$
$= k.2^{k+1} + 1$

Tabelas estáticas e tabelas dinâmicas

A busca binária é um algoritmo extremamente eficiente. Entretanto há um problema a se considerar no caso real. As tabelas muitas vezes não são estáticas, ou seja, durante o processamento ocorrem inserções e remoções de elementos (tabelas dinâmicas). Para manter a tabela ordenada, seriam necessárias deslocamentos de grandes partes da tabela para acomodar as inserções e remoções de elementos.

Portanto outros algoritmos são necessários e que têm que aliar a eficiência da busca binária com inserções e remoções. Não serão estudados neste curso, mas apenas como comentário, são algoritmos de hash (que dividem a tabela em subtabelas lógicas) e que usam estrutura de dados ligada (listas ligadas e árvores).

Classificação de tabelas

Classificação – método da seleção

O algoritmo imediato para se ordenar uma tabela é o seguinte:

a) Determinar o mínimo a partir do primeiro e trocar com o primeiro
b) Determinar o mínimo a partir do segundo e trocar com o segundo
c) Determinar o mínimo a partir do terceiro e trocar com o terceiro
:
:
x) Determinar o mínimo a partir do (n-1)-ésimo e trocar com o (n-1)-ésimo

Exemplo numa tabela de 5 elementos:

```
6    2    2    2
8    8    4    4
4    4    8    5
2    6    6    6
5    5    5    8
```

P103) Programa que dado n, gere uma seqüência com n elementos usando rand(), imprima a seqüência gerada, ordene a mesma pelo método da seleção e imprima a seqüência ordenada.

```
#include <stdio.h>
#include <stdlib.h>

void troca (int *x, int *y)
  {int aux;
   aux = *x; *x = *y; *y = aux;
  }

void ordena(int a[], int n)
  {int i, j, imin;
   /* varrer o vetor de a[o] até a[n-2] (penúltimo) */
   for (i = 0; i < n - 1; i++)
        {/* achar o mínimo a partir de a[i] */
         imin = i;
         for (j = i + 1; j < n; j++)
               if (a[imin] > a[j]) imin = j;
         /* troca a[imin] com a[i] */
         troca(&a[imin], &a[i]);
        }
   }

void geravet(int v[], int k)
  /* gera vetor com k elementos usando rand() */
  {int i;
   srand(9999);
   for (i = 0; i < k; i++) v[i] = rand()%1000;
  }

void impvet(int v[], int k)
  /* imprime vetor com k elementos */
  {int i;
   for (i = 0; i < k; i++) printf("%5d", v[i]);
  }

int main()
 {int vet[1000], n, kk, x;
  /* ler n */
  printf("entre com n:");
  scanf("%d", &n);
```

```
  /* gera o vetor e imprime */
  geravet(vet, n);
  printf("\n****** vetor gerado ******\n");
  impvet(vet, n);
  /* ordena o vetor */
  ordena(vet, n);
  /* imprime vetor ordenado */
  printf("\n****** vetor ordenado ******\n");
  impvet(vet, n);
 }
```

Um exemplo de execução do programa:

```
entre com n:100

****** vetor gerado ******
  691  995  720  855  388  105  163  288  657  851  499  771  920  363   73    3
  514  633  815  153  899  757  934  575  387  841  748  872  726  818  172   62
  473  640   93  724  734  620  136   86  370  610  740  421  605  119   84  631
  217  740  580  286  974  436   40  335  526  923  918  511  973  535  892  945
  115  515  269   43   93  929  984  330  688  488  961  266  152   14  100  899
  264  344   56    8   50  200   88  274  554  135  976   20  998  435  966  592
  209  658  158  643
****** vetor ordenado ******
    3    8   14   20   40   43   50   56   62   73   84   86   88   93   93  100
  105  115  119  135  136  152  153  158  163  172  200  209  217  264  266  269
  274  286  288  330  335  344  363  370  387  388  421  435  436  473  488  499
  511  514  515  526  535  554  575  580  592  605  610  620  631  633  640  643
  657  658  688  691  720  724  726  734  740  740  748  757  771  815  818  841
  851  855  872  892  899  899  918  920  923  929  934  945  961  966  973  974
  976  984  995  998
```

Classificação – método da seleção – análise simplificada

O número de trocas é sempre n – 1. Não seria necessário trocar se o mínimo fosse o próprio elemento, mas para fazer isso teríamos de qualquer forma fazer outra comparação.

O número de comparações `(a[imin] > a[j])` é sempre (n-1)+(n-2)+...+2+1 = n(n-1)/2.

Esse número é exatamente a quantidade de repetições efetuadas. Portanto o tempo que esse algoritmo leva é proporcional a esse número, ou seja, o tempo é proporcional a n^2.

Ordenação - método bubble (da bolha)

Outro método para fazer a ordenação é pegar cada um dos elementos a partir do segundo (a[1] até a[n-1]) e subi-los até que encontrem o seu lugar.

```
6    6    4    4    4    2    2    2
8    4    6    6    2    4    4    4
4    8    8    2    6    6    6    5
2    2    2    8    8    8    5    6
5    5    5    5    5    5    8    8
```

Observe como cada elemento sobe até encontrar o seu lugar.

P104) Programa que dado n, gere uma seqüência com n elementos usando `rand()`, imprima a seqüência gerada, ordene a mesma pelo método da bolha e imprima a seqüência ordenada.

```
#include <stdio.h>
#include <stdlib.h>

void troca (int *x, int *y)
  {int aux;
   aux = *x; *x = *y; *y = aux;
  }

void bubble(int a[], int n)
  {int i, j;
   /* subir com a[i], i = 1, 2, . . ., n-1*/
   for (i = 1; i < n; i++)
        {/* suba com a[i] até encontrar um menor ou chegar em a[0] */
         j = i;
         while (j > 0 && a[j] < a[j-1])
              {/* troca a[imin] com a[i] */
               troca(&a[j], &a[j-1]);
               /* continua subindo */
               j--;
              }
        }
  }

void geravet(int v[], int k)
  /* gera vetor com k elementos usando rand() */
  {int i;
   srand(9999);
   for (i = 0; i < k; i++) v[i] = rand()%1000;
  }

void impvet(int v[], int k)
  /* imprime vetor com k elementos */
  {int i;
   for (i = 0; i < k; i++) printf("%5d", v[i]);
  }

int main()
 {int vet[1000], n, kk, x;
  /* ler n */
  printf("entre com n:");
  scanf("%d", &n);
  /* gera o vetor e imprime */
  geravet(vet, n);
  printf("\n****** vetor gerado ******\n");
  impvet(vet, n);
  /* ordena o vetor */
  bubble(vet, n);
  /* imprime vetor ordenado */
  printf("\n****** vetor ordenado ******\n");
  impvet(vet, n);
 }
```

Nada de novo na saída do programa.

```
entre com n:100

****** vetor gerado ******
  691  995  720  855  388  105  163  288  657  851  499  771  920  363   73    3
  514  633  815  153  899  757  934  575  387  841  748  872  726  818  172   62
  473  640   93  724  734  620  136   86  370  610  740  421  605  119   84  631
  217  740  580  286  974  436   40  335  526  923  918  511  973  535  892  945
  115  515  269   43   93  929  984  330  688  488  961  266  152   14  100  899
  264  344   56    8   50  200   88  274  554  135  976   20  998  435  966  592
  209  658  158  643
****** vetor ordenado ******
    3    8   14   20   40   43   50   56   62   73   84   86   88   93   93  100
  105  115  119  135  136  152  153  158  163  172  200  209  217  264  266  269
  274  286  288  330  335  344  363  370  387  388  421  435  436  473  488  499
  511  514  515  526  535  554  575  580  592  605  610  620  631  633  640  643
  657  658  688  691  720  724  726  734  740  740  748  757  771  815  818  841
  851  855  872  892  899  899  918  920  923  929  934  945  961  966  973  974
  976  984  995  998
```

Exercícios

Considere as 120 (5!) permutações de 1 2 3 4 5:

1) **Encontre algumas que precisem exatamente de 5 trocas para classificá-la pelo método bubble.**
2) **Idem para 7 trocas**
3) **Qual a seqüência que possui o número máximo de trocas e quantas trocas são necessárias?**

Ordenação - método bubble (da bolha) – análise simplificada

O comportamento do método bubble depende se a tabela está mais ou menos ordenada, mas em média, o seu tempo também é proporcional a n^2 como no método anterior.

Quantas vezes o comando `troca(&a[j], &a[j-1])` é executado???
No pior caso, quando a seqüência está invertida, é executado i vezes para cada valor de i = 1, 2, 3, ..., n-1.
Portanto 1 + 2 + 3 + ... + n – 1 = n.(n – 1)/2.

Portanto , no pior caso o bubble é $O(n^2)$.

E quanto ao número médio?

Inversões

Seja $P = a_1\ a_2\ \ldots\ a_n$, uma permutação de $1\ 2\ \ldots\ n$.
O par (i,j) é uma inversão quando $i<j$ e $a_i>a_j$.
Exemplo: $1\ 3\ 5\ 4\ 2$ tem 4 inversões: $(3,2)\ (5,4)\ (5,2)$ e $(4,2)$

No método bubble o número de trocas é igual ao número de inversões da seqüência.
O algoritmo se resume a:

```
for (i = 1; i < n; i++) {
   // elimine as inversões de a[i] até a[0]
   . . .
}
```

Veja também o exemplo:

```
6 8 4 2 5 – elimine as inversões do 8 – 0
```

```
6 8 4 2 5 – elimine as inversões do 4 – 2
4 6 8 2 5 – elimine as inversões do 2 – 3
2 4 6 8 5 – elimine as inversões do 5 – 2
2 4 5 6 8
```

Total de 7 inversões que é exatamente a quantidade de inversões na seqüência.

Para calcularmos então o número de trocas do bubble, basta calcular o número de inversões da seqüência.
É equivalente a calcular o número de inversões de uma permutação de `1 2 ... n`.

Qual o número médio de inversões em todas as seqüências possíveis?
É equivalente a calcular o número médio de inversões em todas as permutações de 1 2 ... n.

Não vamos demonstrar, mas esse número é n.(n-1)/4.

Note que é exatamente a média entre o mínimo e o máximo.

Portanto o bubble é $O(n^2)$.

Ordenação - outros métodos

Existem vários outros métodos melhores que os métodos acima que não veremos neste curso.
Podemos citar:
Quick – que usa o fato de existir algoritmo rápido para particionar a tabela.
Merge – que usa o fato de existir algoritmo rápido para intercalar 2 seqüências ordenadas.
Heap – que organiza a tabela como uma árvore hierárquica.

Nesses métodos, o tempo é proporcional a $(n \cdot \log n)$