Algoritmos de Busca em Tabelas

Dentre os vários algoritmos fundamentais, os algoritmos de busca em tabelas estão entre os mais usados. Considere por exemplo um sistema de banco de dados. As operações de busca e recuperação dos dados são bastante frequentes. Considere também os programas "buscadores" da Internet (Google, Yahoo, etc.). Em todos eles são usados de alguma maneira os algoritmos básicos de busca em tabelas ou arquivos.

Vamos estudar os métodos internos de busca, isto é, os dados estão em tabelas na memória.
No caso de arquivos em dispositivos externos (discos, memórias de massa, etc.) há outras variáveis a se considerar. O tempo de acesso pode envolver operações mecânicas (movimento do braço e rotação no caso do disco) que podem em ser muito mais significativos que o tempo gasto pelo algoritmo. Mas de maneira geral, se um algoritmo é bom para se efetuar busca em tabelas na memória, sempre pode ser adaptado para funcionar também com arquivos externos

Tabelas no Python

Uma tabela no Python é uma lista.
O Python já possui como funções intrínsecas, as funções **`count`** e **`index`** cujo objetivo é exatamente procurar e localizar elementos em uma tabela ou lista:

`lst.count(x)` – retorna quantas vezes o elemento **`x`** ocorre na lista **`lst`**.
`lst.index(x)` – retorna o índice da primeira ocorrência de **`x`** na lista **`lst`**. Se não existe x retorna ValueError.
`lst.index(x, i)` – retorna o índice da primeira ocorrência de **`x`** a partir de **`lst[i]`.**
`lst.index(x, i, f)` – retorna o índice da primeira ocorrência de x a partir de **`lst[i]`** até **`lst[f-1]`**.

Possui ainda o operador **`in`** que verifica se determinado elemento está ou não numa tabela:

```
if x in lst: . . .
if x not in lst: . . .
while x in list: . . .
```

O nosso objetivo ao estudar os algoritmos de busca em tabelas, é entender como essas funções e operadores são implementados. A maneira de fazê-lo é percorrer a tabela de alguma forma, procurando o elemento. Vamos então verificar com mais profundidade os algoritmos que fazem a busca, ou os algoritmos usados nestas funções.

Busca sequencial

Consiste em varrer uma tabela a procura de um determinado elemento, verificando ao final se o mesmo foi ou não foi 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. É claro que pode haver mais de um elemento igual a x.

```
def busca(a, x):
    n = len(a)
    for i in range(n):
        if a[i] == x: return i
    # foi até o fim e não achou
    return -1
```

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

```
def busca(a, x):
    n = len(a)
    i = 0
    while i < n and a[i] != x:
        i += 1
    # verifica se parou porque chegou ao fim ou encontrou um igual
    if i == n: return -1 # chegou a fim sem encontrar
    return i # encontrou um igual
```

Fica como exercício, reescrever a função busca de outras formas, usando os comandos for e while.

Busca Sequencial – análise do algoritmo

Considere a função **busca** na primeira versão acima. Quantas vezes a comparação `(a[i] == x)` é executada?

máximo = **n** (quando não encontra ou quando **x = a[n-1])**
mínimo = **1** (quando **x = a[0])**
médio = **(n+1)/2**
Há uma hipótese importante considerada para se chegar ao número médio de (n+1)/2. Temos que supor que a quantidade de comparações pode ser **1, 2, 3,..., n** com a mesma probabilidade.

De qualquer forma, o algoritmo é **O(n)**.
Assim, esse algoritmo pode ser até bom quando n é pequeno, mas quando n é grande, a demora é inevitável. Suponha por exemplo uma tabela com 1.000.000 de elementos. Se cada repetição demorar cerca de 100 microssegundos, o tempo para uma busca poderá chegar a quase 2 minutos.

Uma melhoria pode ser feita na busca sequencial, quando existem elementos que podem ocorrer com maior frequência. Colocá-los no início da tabela, pois serão encontrados com menos comparações.

Suponha agora que a probabilidade do elemento procurado ser a[i] é pi.
Σ pi (0<=i<n) é a probabilidade de x estar na tabela.
Σ pi < 1 se existe alguma chance de x não estar na tabela.
1 - Σ pi é a probabilidade de x não estar na tabela.

O número médio de comparações será então:
1.p0 + 2.p1 + 3.p2 + ... + n.p(n-1) + n.(1 - Σ pi)

Se os elementos mais procurados estão no início da tabela, o número médio de comparações é menor. Basta verificar a fórmula acima.

Busca sequencial em tabela ordenada

Quando a tabela está ordenada (por exemplo, em ordem crescente dos elementos), uma melhoria pode ser feita. Se durante o processo de busca, encontrarmos um elemento que já é maior que x, não adianta continuar procurando, pois todos os outros serão também maiores.

```
def busca(a, x):
    n = len(a)
    for i in range(n):
        if a[i] == x: return i  # encontrou
        if a[i] > x: return -1  # não adianta continuar procurando
    # foi até o final e não encontrou
    return -1
```

Ou ainda,

```
def busca(a, x):
    n = len(a)
    i = 0
    while i < n and a[i] < x: i += 1
    # verifica se parou porque chegou ao fim ou encontrou um igual
    if i == n: return -1    # chegou ao final
    if a[i] == x: return i  # achou
    return -1  # encontrou um maior então não adianta procurar mais
```

Porém, existe uma solução muito melhor quando a tabela está ordenada.

Busca binária em tabela ordenada

Lembre-se de como fazemos para procurar uma palavra no dicionário. Não vamos verificando folha a folha (busca sequencial). Ao contrário, abrimos o dicionário mais ou menos no meio e daí só há três possibilidades:

a) A palavra procurada está na página aberta
b) A palavra está na parte esquerda do dicionário
c) A palavra está na parte direita do dicionário

Caso não ocorra a situação (a), repetimos o mesmo processo com as páginas da parte direita ou da parte esquerda, onde a palavra tem chance de ser encontrada. Em cada comparação eliminamos cerca de metade das páginas do dicionário.

Em se tratando de uma tabela, isso pode ser sistematizado da seguinte forma:

a) Testa com o elemento do **meio** da tabela;
b) Se for igual, termina o algoritmo porque encontrou o elemento;
c) Se o elemento do meio é maior, repete o processo considerando a tabela do inicio até o **meio** – 1;
d) Se o elemento do meio é menor, repete o processo considerando a tabela do **meio** + 1 até o final;

Se o elemento não está na tabela, chegará o momento em que a tabela restante terá zero elementos que é o outro critério de parada do algoritmo.
Observe que se ocorreu a situação (c) ou (d), a tabela foi reduzida à metade.
Vamos então ao algoritmo:

```
def buscabinaria(a, x):
    n = len(a)
    inicio = 0
    final = n - 1
    # Enquanto a tabela tem elementos continue procurando
    while inicio <= final:
        meio = (inicio + final) // 2
        if a[meio] == x: return meio # encontrou o procurado
        # verifica se continua a procura na parte superior ou inferior
        if x < a[meio]: final = meio -1 # parte superior
        else: inicio = meio + 1         # parte inferior
    # Se chegou aqui é porque esgotou as possibilidades e não encontrou
    return -1
```

Exercício:

Considere a seguinte tabela ordenada:

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

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

2) Diga quantas comparações com elementos da tabela 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:

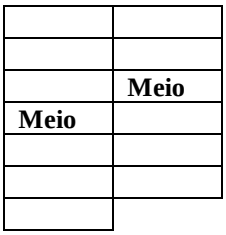
```
2 5 7 11 13 17 25 32 35 39
```

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

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

O meio da tabela

Quando a quantidade de elementos da tabela **`(n = final – inicio + 1)`** é ímpar, o resultado de **`(inicio + final)//2`** fornece exatamente o elemento do meio da tabela. Na próxima repetição temos então 2 tabelas exatamente com **`n // 2`** elementos.
Quanto **`n`** é par, isso não ocorre e a parte inferior da tabela fica com **`n/2`** elementos enquanto que a parte superior fica com **`(n // 2 – 1)`** elementos.



Na verdade, não é nem necessário se escolher exatamente o elemento do meio. O elemento do meio é a melhor escolha, pois as duas partes da tabela ficam reduzidas quase que exatamente à metade.

Outra forma de busca binária

Outra forma de se pensar o algoritmo de busca binária, é usar o fato que a tabela tem um primeiro elemento (**`Base`**) e um tamanho (**`N`**). A cada repetição compara-se o elemento procurado com o elemento médio da tabela (**`x == a[Base + N // 2]?`**). Se encontrar termina. Se **`x > a[Base + N // 2]`** o elemento procurado deve estar acima e a nova base fica sendo (**`Base + N // 2 + 1`**), enquanto o novo valor de **`N`** fica em (**`N // 2`**) se **`N`** é ímpar ou (**`N // 2 - 1`**) se **`N`** é par. Se **`x < a[Base + N // 2]`** o elemento procurado deve estar abaixo e a **`Base`** permanece a mesma enquanto **`N`** fica em (**`N // 2`**). Inicialmente, a **`Base`** é zero e o algoritmo termina quando **`N`** fica zero, isto é, a tabela terminou e não encontramos o elemento procurado.

```
# Procura X em L[Base], ..., L[Base + N - 1]
# Busca binária em tabela ordenada - versão não recursiva
def BBNR(L, X, Base, N):
    # Continua a busca enquanto N > 0
    while N > 0:
```

```
        # meio da tabela
        meio = Base + N // 2
        if L[meio] == X: return meio
        # Cálculo do tamanho da parte de cima e da parte de baixo
        if N % 2 == 0:
            # N é par
            tam_inf, tam_sup = N // 2, N // 2 - 1
        else:
            # N é impar
            tam_inf, tam_sup = N // 2, N // 2
        # Continua a busca na parte de cima ou de baixo.
        if X > L[meio]:
            Base, N = meio + 1, tam_sup
        else:
            N = tam_inf
    # Se saiu do while - não encontrou
    return -1
```

Busca binária recursiva

O algoritmo de busca binária pode também ser implementado de forma recursiva.
Observe que a cada repetição, realizam-se as mesmas operações numa tabela menor.
A versão recursiva é especialmente interessante. Comparamos com o elemento médio da tabela. Se for igual, a busca termina. Se for maior, fazemos a busca binária na tabela superior, senão fazemos busca binária na tabela inferior. A busca termina quando a tabela tiver zero elementos. Observe que a tabela superior vai do elemento médio mais 1 até o final e a tabela inferior vai do início até o elemento médio menos 1.

```
def BBR(a, inicio, final, x):
    if inicio > final: return -1 # não encontrou
    meio = (inicio + final) // 2
    if a[meio] == x: return meio  # encontrou
    # procura na parte superior ou inferior da tabela
    if a[meio] > x: return BBR(a, inicio, meio - 1, x)
    return BBR(a, meio + 1, final, x)
```

Com a função acima, a chamada inicial seria **`BBR(v, 0, n-1, x)`**.

Busca binária recursiva – outra forma

Usando a versão com o primeiro elemento (**`Base`**) e um tamanho (**`N`**) de forma recursiva:

```
# Procura X em L[Base], ..., L[Base + N - 1]
# Busca binária em tabela ordenada - versão recursiva
def BBR(L, X, Base, N):
    if N == 0: return -1 # terminou a busca
    # meio da tabela
    meio = Base + N // 2
    if L[meio] == X: return meio
    # Cálculo do tamanho da parte de cima e da parte de baixo
    if N % 2 == 0:
        # N é par
        tam_inf, tam_sup = N // 2, N // 2 - 1
    else:
        # N é impar
        tam_inf, tam_sup = N // 2, N // 2
```

```
    # Continua a busca na parte de cima ou de baixo.
    if X > L[meio]: return BBR(L, X, meio + 1, tam_sup)
    return BBR(L, X, Base, tam_inf)
```

Busca binária recursiva – com sub-listas do Python

Uma versão recursiva pode ser implementada usando o recurso de sub-listas do Python. Ocorre que neste caso perdemos a informação do índice na tabela em que está o elemento. Além disso, há um problema de uso de memória excessivo porque a cada nova chamada, novas listas são criadas.

```
def BBsub(a, x):
    n = len(a)
    if n == 0: return False
    meio = (n - 1) // 2
    if x == a[meio]: return True
    if x > a[meio]: return BBsub(a[meio+1:], x)
    return BBsub(a[:meio], x)
```

Exercício:

1) Considerando agora a versão acima (**Outra formas de busca binária)**, escreva a função **`def BBR2(a, base, n):`** , de forma recursiva. Neste caso a chamada inicial da função seria **`BBR2(v, 0, n)`**.

Busca binária - Como extrair mais informações do algoritmo

Podemos extrair mais informações do algoritmo de busca binária, além do resultado se achou ou não achou. Quando não encontramos o elemento, a busca termina, nas proximidades de onde o elemento deveria estar. Suponha que desejamos encontrar o lugar onde o elemento deveria estar se fossemos inseri-lo na tabela. Exemplos:

i	0	1	2	3	4	5	6	7	8
A[i]	12	15	18	18	18	20	25	38	44

```
binariaP(23, A, 9) devolve 6
binariaP(17, A, 9) devolve 2
binariaP(15, A, 9) devolve 1
binariaP(18, A, 9) devolve 2
binariaP(55, A, 9) devolve 9
binariaP(12, A, 9) devolve 0
binariaP(10, A, 9) devolve 0
```

De uma forma geral, se **`binariaP(x, a, n)`** devolve `k`, então de `a[0]` até `a[k-1]` todos são menores que `x` e de `a[k]` até `a[n-1]` todos são maiores ou iguais a `x`.

Esta versão tem duas aplicações interessantes:

1) Se é necessário inserir um elemento que ainda não está na tabela, a busca devolve exatamente o lugar em que ele deverá ser inserido. É claro que para isso vamos ter que deslocar todos abaixo dele para abrir espaço.
2) Quando há elementos repetidos na tabela, a busca devolve exatamente o primeiro igual. Todos os outros iguais estarão após ele.

```
# procura x em a[0] até a[n-1] e devolve:
#   0 se x<= a[0]
#   n se x > a[n-1]
#   R se a[R-1] < x <= a[R]
def binariaP(x, a):
    n = len(a)
    # se x está fora da tabela
    if x <= a[0]: return 0
    if x > a[n-1]: return n
    # x pode estar dentro da tabela
    L = 0
    R = n-1
    while R - L > 1:
        M = (R + L) // 2
        if x <= a[M]: R = M
        else: L = M
    return R
```

Para saber se achou z na lista b ou não:

```
k = binariaP(z, b)
m = len(b)
if k < m and b[k] == z: # achou
else: # não achou
```

ou

```
if k == m or b[k] != z: # não achou
else: # achou
```

Busca binária - análise simplificada

Considere agora a primeira versão da busca binária:

```
def buscabinaria(a, x):
    n = len(a)
    inicio = 0
    final = n - 1
    # Enquanto a tabela tem elementos continue procurando
    while inicio <= final:
        meio = (inicio + final) // 2
        if a[meio] == x: return meio # encontrou o procurado
        # verifica se continua a procura na parte superior ou inferior
        if x < a[meio]: final = meio -1 # parte superior
        else: inicio = meio + 1         # parte inferior
    # Se chegou aqui é porque esgotou as possibilidades e não encontrou
    return -1
```

A comparação `a[meio] == x` é o comando significativo para a análise do tempo que esse algoritmo demora, pois representa a quantidade de repetições que serão feitas até o final do algoritmo.
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 = ?

Número máximo de comparações

Note que a cada iteração a tabela fica dividida ao meio. A nova busca é feita numa tabela que tem no máximo a metade dos elementos.

Veja o exemplo para N = 11 (impar) – faremos no máximo 4 comparações.

Comparação	Tamanho Máximo da Tabela
1°	11
2°	5
3°	2
4°	1

Agora para N = 10 (par) – faremos no máximo 4 comparações.

Comparação	Tamanho Máximo da Tabela
1°	10
2°	4 ou 5
3°	1 ou 2
4°	1

Agora para N = 31 (impar e potência de 2 menos 1) – faremos no máximo 5 comparações.

Comparação	Tamanho Máximo da Tabela
1°	31
2°	15
3°	7
4°	3
5°	1

No caso geral

Comparação	Tamanho Máximo da Tabela
1°	N
2°	N / 2
3°	N / 4
4°	N / 8
...	...
(k + 1) °	$N / 2^k$

Enquanto não se encontra o elemento procurado, as comparações continuam até o valor k tal que:

$2^k \leq N < 2^{(k+1)}$

Ou $k \leq \lg N < k + 1$ (lg N é o log de N na base 2)

Ou ainda (somando – k – lg N em todos os membros): $- \lg N \leq - k < - \lg N + 1$

Ou (multiplicando por -1) $\lg N - 1 < k \leq \lg N$

Como estamos interessados no valor (k + 1) termos: $\lg N < (k + 1) \leq 1 + \lg N$

Assim o número máximo de comparações está entre lg N e 1 + lg N

Assim, o algoritmo é `O(lg N)` , ou `O(log 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 seria `1.000.000` e mesmo 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

Número médio de comparações

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 $N = 2^k - 1$.
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, a média é calculada então pela somatória:

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

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

$= 1/N \; . \; ((k-1).2^k + 1)$ (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 $k+1$.

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

Exercícios:

1. Dada uma tabela com 5 elementos x1,x2,x3,x4,x5 com as seguintes probabilidades de busca:

elemento	x1	x2	x3	x4	x5
probabilidade	0.5	0.25	0.15	0.08	0.02

a) Calcule o número médio de comparações no algoritmo de busca seqüencial para a tabela com os elementos na seguinte ordem:

a1) x1, x2, x3, x4, x5
a2) x5, x4, x3, x2, x1
a3) x3, x2, x5, x1, x4

b) Idem supondo a seguinte distribuição

elemento	x1	x2	x3	x4	x5
probabilidade	0.4	0.2	0.1	0.1	0.05

Neste caso o elemento procurado pode não estar na tabela

Considere a seguinte tabela ordenada:

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

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

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

2. Considere a seguinte modificação do algoritmo busca binária.

```
def buscabinaria(a, x):
    n = len(a)
    inicio = 0
    final = n - 1
    # Enquanto a tabela tem elementos continue procurando
    while inicio <= final:
        meio = entre(inicio, final)
        if a[meio] == x: return meio # encontrou o procurado
        # verifica se continua a procura na parte superior ou inferior
        if x < a[meio]: final = meio -1 # parte superior
        else: inicio = meio + 1         # parte inferior
    # Se chegou aqui é porque esgotou as possibilidades e não encontrou
```

```
    return -1
```

A função `entre(inicio,final)` é uma função que devolve um número aleatório entre `inicio` e `fim`.

a) Está correta?
b) Diga qual o número mínimo e o número máximo de vezes que a comparação (v[i] == x) é efetuada e em qual situação ocorre.
c) O que acontece se a função devolver `n/3`?

3. Escreva um algoritmo de busca ternária, isto é, a cada passo calcular `m1=n/2` e `m2=2*n/3`. A tabela então fica dividida em 3 partes (esquerda, meio e direita). Daí basta comparar com m1 e m2. Se não for igual o elemento deve estar em uma das 3 partes. A cada passo a tabela fica dividida por 3.

4. A busca ternária é melhor que a busca binária?