Esta é uma versão simplificada do capítulo. Veja também a versão completa.

Busca em vetor ordenado

"Binary search is to algorithms
what a wheel is to mechanics:
It is simple, elegant, and immensely important."
— Udi Manber, *Introduction to Algorithms*

"A good algorithm is like a sharp knife:
it does what it is supposed to do
with a minimum amount of applied effort.
Using the wrong algorithm to solve a problem
is like trying to cut a steak with a screwdriver:
you may eventually get a digestible result,
but you will expend considerably more effort
than necessary,
and the result is unlikely to be aesthetically pleasing."
— Th. Cormen, Ch. Leiserson, R. Rivest,
Introduction to Algorithms

"Binary search is a notoriously tricky algorithm
to program correctly.
It took seventeen years after its invention
until the first *correct* version of binary search
was published!"
— Steven Skiena, *The Algorithm Design Manual*

Este capítulo estuda o seguinte problema de busca: determinar se um dado objeto é elemento de um dado vetor ordenado. Mais especificamente, *dado um inteiro* `x` *e um vetor crescente* `v[0..n-1]` *de inteiros,*

encontrar um índice `m` *tal que* `v[m] == x` .

É claro que algumas instâncias desse problema não têm solução. Esse é o caso, por exemplo, das instâncias em que `n` vale `0`, ou seja, o vetor é vazio. No exemplo abaixo, se `x` vale 555 então `4` é a solução. Se `x` vale 800 ou 1000, não há solução.

0												n-1
111	222	333	444	555	555	666	777	888	888	888	999	999

Vamos examinar dois algoritmos para o problema: um óbvio mas lento e outro menos óbvio mas muito mais rápido.

Para as instâncias que têm solução, nossos algoritmos devem devolver um índice. No caso das instâncias que *não têm* solução, é conveniente que nossos algoritmos também devolvam um índice, mas um que não possa ser confundido com uma solução "válida". Nossa *decisão de projeto*: devolver `-1` nesses casos.

Exercícios 1

1. Escreva uma função que decida se um vetor `v[0..n-1]` está em ordem crescente. Depois, critique o código abaixo.

```
int verifica (int v[], int n) {
   int anterior = v[0], sim = 1;
   for (int i = 1; i < n && sim; i++) {
      if (anterior > v[i]) sim = 0;
      anterior = v[i]; }
   return sim; }
```

2. FORMULAÇÃO SOFISTICADA. Considere a seguinte formulação sofisticada do problema de busca: dado `x` e um vetor crescente `v[0..n-1]`, encontrar um índice `j` no intervalo `0..n` tal que

   `v[j-1] < x ≤ v[j]` .

   (Essa formulação tem muitas vantagens sobre a formulação original.) Mostre que essa formulação do problema é mais geral que a formulação básica dada acima. Mostre como um algoritmo para a formulação sofisticada pode ser usado para resolver a formulação básica. Mostre que a formulação sofisticada sempre tem solução. Em que condições `0` é uma solução do problema? Em que condições `n` é uma solução do problema? [Solução.]

Busca sequencial

Comecemos com um algoritmo óbvio, que examina um a um todos os elementos do vetor. Segue uma implementação do algoritmo:

```
// A função abaixo recebe um inteiro x e um vetor
// crescente v[0..n-1]. Ela devolve um índice m
// em 0..n-1 tal que v[m] == x. Se tal m não existe,
// a função devolve -1.

int
buscaSequencial (int x, int n, int v[]) {
   int m = 0;
   while (/*A*/ m < n && v[m] < x)
      ++m;
   if (m < n && v[m] == x)
      return m;
   else
      return -1;
}
```

O valor de `m` muda a cada iteração, mas as relações `m ≤ n` e `v[m-1] < x` são *invariantes*: elas valem no início de cada iteração. Mais precisamente, essas relações invariantes valem imediatamente antes de cada comparação de `m` com `n` (ponto `A` do código). No começo da primeira iteração, a relação vale se estivermos dispostos a imaginar que `v[-1]` é $-\infty$.

A relação invariante vale, em particular, no início da *última* iteração, quando `m ≥ n` ou `v[m] ≥ x`. Se `m ≥ n`, temos `m == n` e `v[n-1] < x` e a função devolve `-1`. Se `m < n` mas `v[m] > x`, a função devolve `-1`. Se `m < n` e `v[m] == x`, a função devolve `m`. Nos três casos, a função devolve a resposta correta. Essa discussão mostra que a função `buscaSequencial` está correta.

Quantas iterações a função faz? Ou melhor, quantas vezes a função compara `x` com elementos de `v`? No pior caso, `x` é comparado com cada elemento do vetor, e portanto o número de comparações é `n`.

O consumo de tempo da função é proporcional ao número de comparações que envolvem `x`, e portanto proporcional a `n` no pior caso. Assim, se uma busca consome T microssegundos quando o vetor tem N elementos, consumirá $10T$ microssegundos quando o vetor tem $10N$ elementos.

É possível resolver o problema com menos comparações? É possível resolver o problema sem comparar `x` com cada elemento do vetor? A resposta é afirmativa, como veremos a seguir.

Exercícios 2

1. Discuta a seguinte versão da função `buscaSequencial`:

```
int buscaSequencial (int x, int n, int v[]) {
   int m;
   for (m = 0; m < n; m++)
      if (x >= v[m]) break;
   if (m < n && v[m] == x) return m;
   else return -1; }
```

2. Critique a seguinte versão da função `buscaSequencial`:

```
int busca (int x, int n, int v[]) {
   int m = 0;
   while (v[m] < x && m < n) ++m;
   if (v[m] == x) return m;
   else return -1; }
```

3. Discuta a seguinte versão recursiva da função `buscaSequencial`:

```
int buscaR (int x, int n, int v[]) {
   if (n == 0) return -1;
   if (x == v[n-1]) return n-1;
   return buscaR (x, n-1, v); }
```

4. Escreva uma versão da `buscaSequencial` para a formulação sofisticada do problema de busca.

Busca binária

Existe um algoritmo muito mais rápido que a busca sequencial. Ele é análogo ao método que se usa para encontrar um nome em uma lista telefônica. É claro que essa ideia só funciona porque o vetor está *ordenado*.

```
// A função abaixo recebe um inteiro x e um vetor
// crescente v[0..n-1]. Ela devolve um índice m
// tal que v[m] == x ou devolve -1 se tal m não
```

```
// existe.

int
buscaBinaria (int x, int n, int v[]) {
   int e, m, d;                              // 1
   e = 0; d = n-1;                           // 2
   while (e <= d) {                          // 3
      m = (e + d)/2;                         // 4
      if (v[m] == x) return m;               // 5
      if (v[m] < x) e = m + 1;               // 6
      else d = m - 1;                        // 7
   }                                         // 8
   return -1;                                // 9
}
```

Os nomes das variáveis não foram escolhidos por acaso: `e` lembra "esquerda", `m` lembra "meio" e `d` lembra "direita". O resultado da divisão por `2` na expressão `(e+d)/2` é automaticamente [truncado](), pois as variáveis são do tipo `int`. Por exemplo, se `e` vale `3` e `d` vale `6`, então `(e+d)/2` vale `4`.

0			e			d						n-1
111	222	333	444	555	555	666	777	888	888	888	999	999

A ideia da busca binária (= *binary search*) é um verdadeiro [ovo de Colombo](). Essa ideia é o ponto de partida de algoritmos eficientes para muitos problemas.

Exercícios 3

1. Responda as seguintes perguntas sobre a função `buscaBinaria`: (1) Que acontece se "`while (e <= d)`" for trocado por "`while (e < d)`"? (2) Que acontece se "`while (e <= d)`" for trocado por "`while (e <= d+1)`"? (3) Que acontece se "`e = m+1`" for trocado por "`e = m`"? E se for trocado por "`e = m-1`"? (4) Que acontece se "`d = m-1`" for trocado por "`d = m`" ou por "`d = m+1`"?
2. Suponha que `v[i] = i` para cada `i`. Execute `buscaBinaria` com `n = 9`, e com vários valores de `x`. Repita o exercício com `n = 14` e `x = 9`. Repita o exercício com `n = 15` e `x = 9`.
3. Execute a função `buscaBinaria` com `n = 16`. Quais os possíveis valores de `m` na primeira iteração? Quais os possíveis valores de `m` na segunda iteração? Na terceira? Na quarta?
4. Confira a validade da seguinte afirmação: se `n+1` é uma potência de `2`, a expressão `(e+d)` é sempre divisível por `2`, quaisquer que sejam `v` e `x`.

A função buscaBinaria está correta?

Para entender a função `buscaBinaria`, basta [verificar]() que no início de cada repetição do `while`, imediatamente antes da comparação de `e` com `d`, vale a relação

`v[e-1] < x < v[d+1] .`

Essa relação é, portanto, *invariante*. Para que o invariante valha no início da *primeira* iteração basta imaginar que `v[-1]` vale $-\infty$ e que `v[n]` vale $+\infty$. Esse jogo de imaginar faz sentido porque o vetor é crescente.

Se a execução da função termina na linha `5`, o índice `m` é obviamente uma solução do problema. Suponha agora que a execução termina na linha `9`. Então a última iteração começou (na linha `3`) com `e > d`. Em virtude do invariante, temos `v[e-1] < v[d+1]` e portanto `e-1 < d+1`, uma vez que o vetor é crescente. Logo, `e == d+1`. A relação `v[e-1] < x < v[d+1]` garante agora que `x` está estritamente entre dois elementos consecutivos do vetor. Como o vetor é crescente, concluímos que `x` é diferente de todos os elementos do vetor. Portanto, ao devolver `-1` a função está se comportando como prometeu.

Resta verificar que a execução da função termina. Suponha que a execução não é interrompida na linha `5`. Como o valor da diferença `d - e` [diminui a cada iteração](), a diferença fica estritamente negativa depois de algumas iterações e então o processo iterativo termina na linha `9`.

Exercícios 4

1. INVARIANTE. Considere a função `buscaBinaria`. Mostre que `v[e-1] < x < v[d+1]` no início de cada iteração (ou seja, imediatamente antes da comparação de `e` com `d` na linha `3`).
2. CONVERGÊNCIA. Considere a função `buscaBinaria`. Mostre que a diferença `d - e` diminui a cada iteração.

3. Escreva uma versão da função `buscaBinaria` que tenha o seguinte invariante: no início de cada iteração, `v[e-1] < x < v[d]`.
4. Escreva uma versão da função `buscaBinaria` que tenha o seguinte invariante: no início de cada iteração, `v[e-1] < x ≤ v[d]`.
5. Escreva uma versão da busca binária que procure `x` em `v[0..n]`. Escreva outra versão para `v[1..n]`.
6. A seguinte implementação da busca binária funciona corretamente? Qual o invariante dessa versão? Que acontece se trocarmos "`while (e < d-1)`" por "`while (e < d)`"?

```
e = -1; d = n;
while (e < d-1) {
   m = (e + d)/2;
   if (v[m] == x) return m;
   if (v[m] < x) e = m;
   else d = m;
}
return -1;
```

7. Escreva e analise uma versão da função `buscaBinaria` para a formulação sofisticada do problema de busca. [Solução.]

Desempenho da busca binária

Quanto tempo a função `buscaBinaria` consome? No início da primeira iteração, `d - e` vale aproximadamente `n`. No início da segunda, vale aproximadamente `n/2`. No início da terceira, aproximadamente `n/4`. No início da `(k+1)`-ésima, aproximadamente $n/2^k$. Quando `k` passar de log n, o valor da expressão $n/2^k$ fica menor que 1 e a execução do algoritmo termina. Logo, o número de iterações é aproximadamente

`lg (n)`

no pior caso, sendo `lg (n)` o piso de log `n`. Isso é muito menos que o número de iterações da busca sequencial, pois log transforma multiplicações em somas. Por exemplo, se uma busca em um vetor de tamanho N exige T iterações, então uma busca em um vetor de tamanho $2N$ fará apenas $1 + T$ iterações, uma busca em um vetor de tamanho $8N$ fará apenas $3 + T$ iterações, e uma busca em um vetor de tamanho $16N$ fará apenas $4 + T$ iterações.

O consumo de tempo da função `buscaBinaria` é proporcional ao número de iterações e portanto proporcional a log `n` no pior caso.

Exercícios 5

1. Suponha que o vetor `v` tem 511 elementos e que `x` não está no vetor. Quantas vezes, exatamente, a função `buscaBinaria` comparará `x` com um elemento do vetor? Suponha agora que o vetor `v` tem 50000 elementos e que `x` não está no vetor. Quantas vezes, aproximadamente, a função `buscaBinaria` comparará `x` com um elemento do vetor?
2. Se preciso de t segundos para fazer uma busca binária em um vetor com n elementos, de quando tempo preciso para fazer uma busca em n^2 elementos?

Versão recursiva da busca binária

Para formular uma versão recursiva da busca binária é preciso generalizar ligeiramente o problema, trocando `v[0..n-1]` por `v[e..d]`. Para estabelecer uma ponte entre a formulação original e a generalizada, vamos usar uma "função-embalagem" (= *wrapper-function*) `buscaBinaria2`, que repassa o serviço para a função recursiva `bb`.

```
// Esta função recebe um inteiro x e um vetor
// crescente v[0..n-1] e devolve um índice m
// em 0..n-1 tal que v[m] == x. Se tal m
// não existe, devolve -1.

int
buscaBinaria2 (int x, int n, int v[]) {
   return bb (x, 0, n-1, v);
}
```

```
// Esta função recebe um inteiro x e um vetor
// crescente v[e..d] e devolve um índice m
// em e..d tal que v[m] == x. Se tal m
// não existe, devolve -1.

static int
bb (int x, int e, int d, int v[]) {
   if (e > d) return -1;
   else {
      int m = (e + d)/2;
      if (v[m] == x) return m;
```

```
      if (v[m] < x)
         return bb (x, m+1, d, v);
      else
         return bb (x, e, m-1, v);
   }
}
```

Qual a profundidade da recursão na função `bb`? Ou seja, quantas vezes `bb` chama a si mesma? Resposta: cerca de log `n` vezes.

Exercícios 6

1. Critique a seguinte versão da função `bb`. A função recebe um inteiro `x` e um vetor crescente `v[e..d]` tal que `e` ≤ `d` e promete devolver um índice `m` em `e..d` tal que `v[m] == x` (ou `-1`, se tal `m` não existe).

```
int bb (int x, int e, int d, int v[])  {
   if (e == d) {
      if (v[d] == x) return d;
      else return -1;
   }
   int m = (e + d) / 2;
   if (v[m] == x) return m;
   if (v[m] < x) return bb (x, m+1, d, v);
   else return bb (x, e, m-1, v);
}
```

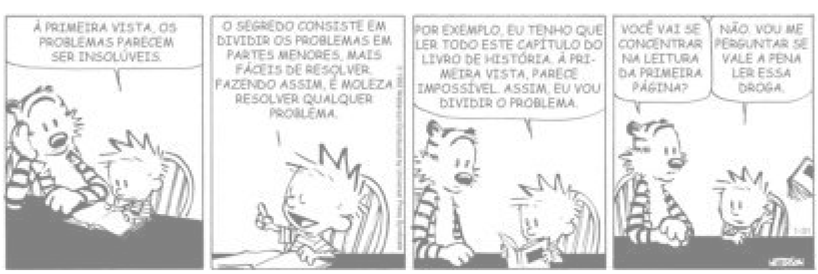
2. Escreva e analise versões das funções `buscaBinaria2` e `bb` para a formulação sofisticada do problema de busca. [Solução.]

Exercícios 7

1. Vetor de strings. Suponha que `v[0..n-1]` é um vetor de strings em ordem lexicográfica. Escreva uma função que receba uma string `x` e devolva um índice `m` tal que a string `x` é igual à string `v[m]`.
2. Vetor de structs. Suponha que cada elemento do vetor `v[0..n-1]` é uma struct com dois campos: o nome de um aluno e o número do aluno. Suponha que o vetor está em ordem crescente de números. Escreva uma função de busca binária que receba o número de um aluno e devolva o seu nome. Se o número não está no vetor, a função deve devolver a string vazia.
3. Familiarize-se com a função `bsearch` da biblioteca `stdlib`.

Exercícios 8: divisão e conquista

O paradigma da busca binária serve de base para o conhecido "método da divisão e conquista", que resolve eficientemente muitos problemas computacionais.



1. Escreva uma função que receba um vetor inteiro *estritamente* crescente `v[0..n-1]` e devolva um índice `i` entre `0` e `n-1` tal que `v[i] = i;` se tal `i` não existe, a função deve devolver `-1`. O seu algoritmo não deve fazer mais que log `n` comparações envolvendo elementos de `v`.
2. Escreva uma função eficiente que receba inteiros estritamente positivos k e n e calcule k^n. Quantas multiplicações sua função executa?

Veja a versão mais sofisticada deste capítulo

Veja o verbete *Binary search algorithm* na Wikipedia

Atualizado em 2016-02-09
https://www.ime.usp.br/~pf/algoritmos/
Paulo Feofiloff
DCC-IME-USP