

Apostila Introdutória de Algoritmos

Guilherme D. da Fonseca
Celina M. H. de Figueiredo

Versão rascunho para o curso de Análise de Algoritmos da Unirio
2009.

18 de Setembro de 2009

CAPÍTULO 2

Busca Binária

A técnica de busca binária consiste em examinar um número pequeno de elementos da entrada (normalmente apenas um) e, com isso, descartar imediatamente uma fração constante dos elementos da entrada (normalmente metade). Procede-se desta maneira até que o conjunto de elementos candidatos a serem a solução do problema seja suficientemente pequeno.

2.1. Busca em vetor

Um vetor $v = (v_1, \dots, v_n)$ contém n números reais e desejamos saber se um número x está ou não no vetor. Um algoritmo trivial é percorrer este vetor do primeiro ao último elemento, comparando-os com x . Ao encontrarmos um elemento com valor x , podemos parar. Mas, se nenhum elemento tiver este valor, somos obrigados a ler o vetor inteiro. Claramente não podemos fazer melhor que isso no pior caso, pois qualquer posição é candidata a ter o valor x e não inspecionar esta posição nos levaria a uma resposta errada.

Vamos mudar um pouco o problema. Agora sabemos que o vetor $v = (v_1, \dots, v_n)$ está ordenado, mais especificamente, para i de 1 até $n - 1$ temos $v_i \leq v_{i+1}$.

PROBLEMA 2. *Dados um vetor $v = (v_1, \dots, v_n)$ ordenado, contendo elementos reais e um número real x , determinar se existe uma posição i tal que $v_i = x$.*

O algoritmo anterior também funciona para este problema, mas sua complexidade de tempo de pior caso é $O(n)$. Será que podemos fazer melhor? A resposta é sim. Usando uma técnica chamada busca binária, podemos melhorar a complexidade para $O(\lg n)$. De fato, em um vetor com 1000 elementos, o número de comparações no pior caso reduz de 1000 para 10.

A técnica se torna mais intuitiva se apresentada como um jogo. Um jogador João pensa em um número de 1 a 1000 e uma jogadora Maria deve adivinhar este número. Quando Maria chuta um número, João responde se ela acertou ou, caso contrário, se o número em que ele pensou é maior ou menor do que o que ela chutou. A melhor estratégia para Maria é sempre dividir o intervalo em duas partes iguais. Começa chutando 500 (poderia ser 501 também). Em seguida chuta 250 ou 750, de acordo com a resposta de João.

Retornando ao problema de encontrar um elemento de valor x em um vetor ordenado, primeiro examinamos o elemento $v_{\lfloor (n+1)/2 \rfloor}$. Se $x > v_{\lfloor (n+1)/2 \rfloor}$, então sabemos que só as posições de $\lfloor (n+1)/2 \rfloor + 1$ a n são candidatas a ter valor x . Analogamente, se $x < v_{\lfloor (n+1)/2 \rfloor}$, então sabemos que só as posições de 1 a $\lfloor (n+1)/2 \rfloor - 1$ são candidatas a ter valor x . Claro que, se $x = v_{\lfloor (n+1)/2 \rfloor}$, o problema já está resolvido. Repetimos este processo até encontrarmos um elemento de valor x ou o intervalo ter apenas um elemento ou nenhum elemento. O pseudo-código deste algoritmo pode ser encontrado na figura 2.1.

É trivial provar que este algoritmo funciona, isto é, resolve o problema 2. Ainda assim vamos fazer a prova formalmente.

TEOREMA 2.1. *O algoritmo que acabamos de descrever resolve corretamente o problema 2.*

DEMONSTRAÇÃO. Ao examinarmos um elemento v_i do vetor $v = (v_1, \dots, v_n)$, procurando um elemento x temos três opções: $x < v_i$, $x = v_i$ e $x > v_i$. Caso $x = v_i$ o algoritmo retorna v_i , funcionando corretamente. Caso $x < v_i$, como o vetor está ordenado, somente os elementos de v_1 a v_{i-1} são candidatos a ter valor x e o algoritmo resolve este problema recursivamente. O caso $x > v_i$ é análogo.

Entrada:

v : Vetor de reais em ordem crescente.

$inicio$: Primeiro elemento da partição do vetor. Inicialmente 1.

fim : Último elemento da partição do vetor. Inicialmente o tamanho do vetor.

x : Valor que está sendo procurado.

Saída:

Índice i tal que $v[i] = x$, se existir.

BuscaBinária($v, inicio, fim, x$)

Se $inicio < fim$

 Retorne " $x \notin v$ "

Se $inicio = fim$

 Se $v[inicio] = x$

 Retorne $inicio$

 Senão

 Retorne " $x \notin v$ "

$meio \leftarrow \lfloor (inicio + fim)/2 \rfloor$

Se $v[meio] > x$

 Retorne **BuscaBinária**($v, inicio, meio - 1, x$)

Se $v[meio] < x$

 Retorne **BuscaBinária**($v, meio + 1, fim, x$)

Retorne $meio$

FIGURA 2.1. Solução do Problema 2

O caso base é quando o vetor tem apenas 1 elemento ou nenhum elemento. Caso o vetor não tenha nenhum elemento, claramente não tem elemento com valor x . Caso tenha apenas 1 elemento o algoritmo resolve o problema comparando este elemento com x . \square

Resta agora analisarmos a complexidade de tempo do algoritmo. Faremos uma prova geral que servirá de base para todos os algoritmos baseados em busca binária. A idéia é que, como a cada passo descartamos uma fração constante dos elementos, a complexidade de tempo é logarítmica. Vamos chamar de $T(n)$ o tempo gasto pelo algoritmo para um vetor de tamanho n . Em um tempo constante, o algoritmo descarta uma fração $\alpha < 1$ constante (normalmente $\alpha = 1/2$) dos elementos. Temos então

$$T(n) = T(\alpha n) + 1.$$

Podemos assumir que o tempo constante de cada passo seja 1, pois a notação O ignora constantes multiplicativas.

Vamos provar que $T(n) = \Theta(\lg n)$, supondo que $T(\alpha n) = \Theta(\lg n)$. Usando indução temos

$$T(n) = T(\alpha n) + 1 = c \lg(\alpha n) + 1 = c \lg n + c \lg \alpha + 1.$$

Se fizermos $c = -1/\lg \alpha$ temos $T(n) = c \lg n$ e finalizamos a indução.

Com isto temos:

TEOREMA 2.2. *O algoritmo que descrevemos tem complexidade de tempo $\Theta(\lg n)$, onde n é o número de elementos do vetor.*

2.2. Busca em vetor ciclicamente ordenado

Muitas vezes, falaremos de índices de vetores módulo n . Com isto queremos dizer que, se $v = (v_1, \dots, v_n)$ e nos referimos a um elemento v_i fora do intervalo, ou seja, $i < 1$ ou $i > n$, então estamos nos referindo ao elemento do intervalo obtido somando ou subtraindo n a i quantas

vezes for necessário. Por exemplo, em um vetor $v = (v_1, \dots, v_5)$, quando dizemos v_{-5} , v_0 ou v_{10} estamos nos referindo ao elemento v_5 .

Seja $v = (v_1, \dots, v_n)$ um vetor de reais com índices módulo n . Dizemos que v está ciclicamente ordenado se o número de elementos v_i tais que $v_i \leq v_{i+1}$ para i de 1 a n é igual a $n - 1$. Por exemplo, o vetor $(5, 8, 9, 10, 1, 3)$ está ciclicamente ordenado.

PROBLEMA 3. *Dados um vetor v ciclicamente ordenado, contendo elementos reais e um número real x , determinar a posição i tal que $v[i] = x$, se existir.*

Para resolvermos este problema devemos examinar duas posições ao invés de uma. É útil pensarmos no vetor como um círculo. Examinamos os elementos v_i e v_j com $i < j$ de modo que o número de elementos entre v_i e v_j pelos dois lados do círculo seja aproximadamente igual. Caso $v_i \leq v_j$, sabemos que se $v_i \leq x < v_j$ então x só pode estar nas posições de i até $j - 1$ e se $x < v_i$ ou $x \geq v_j$ então x só pode estar nas posições menores que i ou maiores ou iguais a j . Caso $v_i > v_j$, sabemos que se $x \geq v_i$ ou $x < v_j$ então x está nas posições de i até $j - 1$ e se $v_j \leq x < v_i$ então x está nas posições menores ou iguais a j ou maiores que i .

TEOREMA 2.3. *O algoritmo que acabamos de descrever resolve corretamente o problema 3.*

DEMONSTRAÇÃO. Buscando um elemento com valor x , examinamos dois elementos v_i e v_j do vetor $v = (v_1, \dots, v_n)$, com $i < j$. Caso $v_i \leq v_j$ o vetor formado pelos elementos de v_i à v_j está ordenado e x é candidato a estar nas posições de índice i até $j - 1$ se e só se $v_i \leq x < v_j$. O procedimento é chamado recursivamente para a partição do vetor candidata a conter elemento de valor x . Caso $v_i > v_j$ o vetor formado pelos elementos após v_j e anteriores a v_i está ordenado e o argumento é análogo.

O caso base é quando o vetor tem apenas 1 elemento ou nenhum elemento. Caso o vetor não tenha nenhum elemento, claramente não tem elemento com valor x . Caso tenha apenas 1 elemento o algoritmo resolve o problema comparando este elemento com x . \square

Para facilitar a implementação podemos sempre pegar como p_i o ponto com o menor índice i dentro do intervalo, como está ilustrado na figura 2.2. Assim evitamos que a partição do vetor seja descontínua na memória.

A complexidade de tempo deste algoritmo é $\Theta(\lg n)$, pelo mesmo princípio do algoritmo da sessão 2.1.

2.3. Ponto extremo de polígono convexo

A técnica de busca binária tem várias aplicações em geometria computacional, especialmente quando a entrada é um polígono convexo.

Um ponto no plano é representado por um par de coordenadas reais. Representamos um polígono de n vértices como um vetor $v = (v_1, \dots, v_n)$ contendo n pontos no plano. A posição v_1 contém um dos vértices (qualquer um), v_2 o próximo vértice no sentido anti-horário e assim por diante. Denotamos por $\odot(p_1, p_2, p_3)$ o ângulo positivo $\widehat{p_1 p_2 p_3}$ medido no sentido anti-horário. Devido a natureza cíclica dos polígonos, trabalharemos com índices módulo n , ou seja, se o índice do vetor for maior do que n ou menor do que 1, devemos somar ou subtrair n até que o índice esteja neste intervalo. Um polígono é convexo se, para i de 1 à n , o ângulo $\odot(v_{i-1}, v_i, v_{i+1})$ for maior que 180° (figura 2.3(a)). Note que quando $i = 1$, ao dizermos $i - 1$ estamos nos referindo a posição n . Quando $i = n$, ao dizermos $i + 1$ estamos nos referindo a posição 1.

Existem várias definições equivalentes para polígono convexo. A maioria caracteriza a interseção do polígono com uma reta. Uma definição deste tipo é: um polígono é convexo se sua interseção com uma reta ou é nula ou é um ponto ou um segmento de reta. Esta definição considera o polígono cheio, ou seja, o interior do polígono também é considerado parte do polígono. Esta última definição não nos fornece diretamente nenhum algoritmo para verificar se, dado um polígono, ele é convexo. Já a definição do parágrafo anterior nos fornece um algoritmo linear para verificar convexidade. Basta examinarmos todos os ângulos.

Dizemos que um vértice v_i de um polígono $P = (v_1, \dots, v_n)$ é extremo na direção de um vetor d se $d \cdot v_i \geq d \cdot v_j$ para todo $j \neq i$. Denotamos por $u \cdot v$ o produto escalar $u_x v_x + u_y v_y$.

Entrada:

v : Vetor de reais ciclicamente ordenado.
 $inicio$: Primeiro elemento da partição do vetor. Inicialmente 1.
 fim : Último elemento da partição do vetor. Inicialmente o tamanho do vetor.
 x : Valor que está sendo procurado.

Saída:

Índice i tal que $v[i] = x$, se existir.

BuscaBináriaCíclica(v , $inicio$, fim , x)

Se $inicio < fim$

Retorne " $x \notin v$ "

Se $inicio = fim$

Se $v[inicio] = x$

Retorne $inicio$

Senão

Retorne " $x \notin v$ "

$meio \leftarrow \lfloor (inicio + fim + 1)/2 \rfloor$

Se $v[inicio] \leq v[meio]$

Se $x \geq v[inicio]$ e $x < v[meio]$

Retorne BuscaBináriaCíclica(v , $inicio$, $meio - 1$, x)

Senão

Retorne BuscaBináriaCíclica(v , $meio$, fim , x)

Senão

Se $x \geq v[meio]$ e $x < v[inicio]$

Retorne BuscaBináriaCíclica(v , $meio$, fim , x)

Senão

Retorne BuscaBináriaCíclica(v , $inicio$, $meio - 1$, x)

FIGURA 2.2. Solução do Problema 3

Uma outra definição mais geométrica é que v_i é extremo na direção d se a reta perpendicular a d que passa por v_i divide o plano em dois semiplanos tais que todos os pontos do polígono que não estão sobre a reta estão em um mesmo semiplano e o ponto $v_i + d$ está no outro semiplano (figura 2.3(b)).

Agora podemos definir o problema:

PROBLEMA 4. *Dados um polígono convexo P e um vetor d determinar o vértice de P extremo na direção d .*

Vamos começar pegando dois vértices quaisquer v_i e v_j do polígono $P = (v_1, \dots, v_n)$, com $i < j$. Podemos usar este par de vértices para decompor P em dois polígonos convexos $P_1 = (v_i, v_{i+1}, \dots, v_j)$ e $P_2 = (v_1, v_2, \dots, v_i, v_j, v_{j+1}, \dots, v_n)$. Para usarmos o princípio de busca binária precisamos descobrir qual desses dois polígonos contém o ponto extremo. Primeiro comparamos $d \cdot v_i$ com $d \cdot v_j$. Vamos considerar inicialmente que $d \cdot v_i > d \cdot v_j$ e depois trataremos do outro caso. Comparamos então $d \cdot v_i$ com $d \cdot v_{i+1}$. Caso $d \cdot v_i > d \cdot v_{i+1}$ o polígono que contém o ponto extremo é $P_1 = (v_i, v_{i+1}, \dots, v_j)$. Para provarmos este fato vamos considerar a reta r perpendicular a d que passa por v_i e os dois semiplanos S e \bar{S} definidos por ela. Chamamos de S o semiplano que contém v_{i+1} . Os pontos que estão em S não são candidatos a serem extremos, pois o produto escalar de qualquer um desses pontos com d é menor que $d \cdot v_i$. Todos os pontos de P_2 estão em S , pois caso contrário r interceptaria o interior de P_2 e também tangenciaria P_2 no vértice v_i . Caso $d \cdot v_i < d \cdot v_{i+1}$ o polígono que contém o ponto extremo é P_2 , usando o mesmo argumento. Caso $d \cdot v_i < d \cdot v_j$, devemos comparar $d \cdot v_j$ com $d \cdot v_{j+1}$. Se $d \cdot v_j > d \cdot v_{j+1}$,

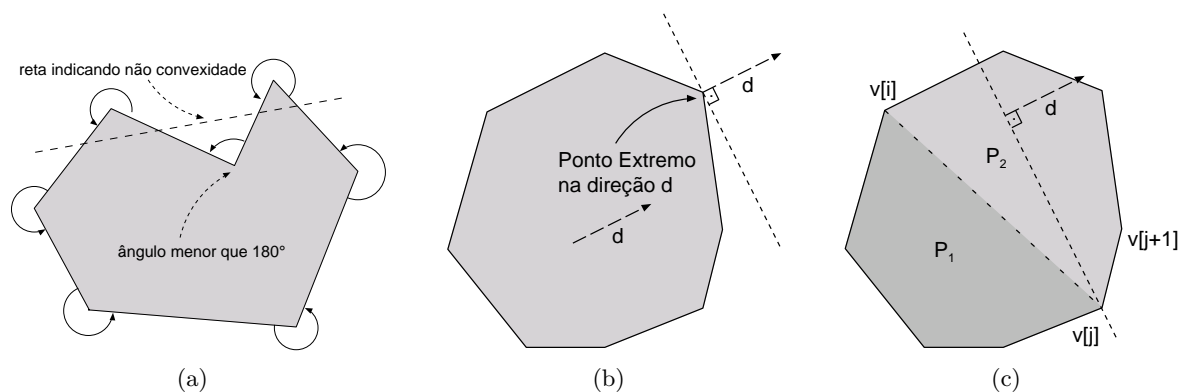


FIGURA 2.3. (a) Polígono não convexo. (b) Ponto extremo na direção d . (c) Algoritmo examinando v_i , v_j e v_{j+1} com $d \cdot v_i < d \cdot v_j$ e $d \cdot v_j < d \cdot v_{j+1}$

então P_1 contém o ponto extremo. Caso contrário P_2 contém o ponto extremo. Um exemplo está ilustrado na figura 2.3(c).

Frequentemente, ao projetarmos um algoritmo para resolver um determinado problema, acabamos descobrindo que nosso algoritmo é ainda mais poderoso do que o planejado, podendo resolver problemas aparentemente mais difíceis. Para provarmos que nosso último algoritmo funciona, usamos o fato do polígono ser convexo para garantir que se r intercepta um polígono em um vértice, então r não pode interceptar o polígono em uma região que não tem extremo neste vértice. Mas r não é uma reta qualquer, e sim uma reta perpendicular a d . Podemos definir um polígono d -monótono como: um polígono é d -monótono se sua interseção com uma reta perpendicular a d ou é nula ou é um ponto ou um segmento de reta. Nosso algoritmo não funciona apenas com polígonos convexos, mas também com polígonos d -monótonos quaisquer!

TEOREMA 2.4. *O algoritmo que acabamos de descrever resolve corretamente o problema 4.*

Para obtermos complexidade $O(\lg n)$, devemos escolher v_i e v_j de modo que o número de vértices entre v_i e v_j nos dois sentidos seja aproximadamente igual. O pseudo-código da figura 2.4 nos leva a uma implementação bastante simples e eficiente deste algoritmo, embora alguns detalhes sejam diferentes de nossa descrição.

Uma das técnicas usadas neste pseudo-código é usar uma função `PontoExtremoLin` que examina os vértices um a um e encontra o vértice extremo (esta função simples não está descrita no pseudo-código). Existem duas vantagens em chamar esta função quando o polígono é muito pequeno. Uma é que para polígonos pequenos é mais rápido examinar todos os vértices do que fazer uma busca binária. A outra vantagem é que assim não precisamos nos preocupar com casos pequenos, que teriam de ser tratados de forma especial.

A complexidade de tempo deste algoritmo é $\Theta(\lg n)$, pelo mesmo princípio do algoritmo da sessão 2.1.

2.4. Função de vetor

Vamos finalizar o capítulo com um exemplo de algoritmo que, embora seja baseado em busca binária, não tem complexidade logarítmica. O nosso problema é o seguinte:

PROBLEMA 5. *Dados um vetor $v = (v_1, \dots, v_n)$ ordenado, contendo elementos reais e uma função $f : \mathbb{R} \rightarrow \mathbb{R}$, construir um vetor $v' = (v'_1, \dots, v'_n)$ tal que $v'_i = j$ se existir j com $f(v_i) = v_j$ e $v'_i = \varepsilon$ caso contrário.*

Claramente não podemos esperar resolver este problema em tempo $O(\lg n)$ porque temos que construir um vetor com n elementos. Ainda assim a técnica de busca binária resolve diretamente o problema, como está ilustrado no pseudo-código da figura 2.5.

A prova do teorema abaixo é trivial e fica como exercício.

Entrada:

v : Vetor com os vértices do polígono.

$inicio$: Primeiro elemento da partição do vetor. Inicialmente 1.

fim : Último elemento da partição do vetor. Inicialmente o tamanho do vetor.

d : Direção em que se deseja maximizar.

Saída:

Vértice extremo na direção d .

Observações:

PontoExtremoLin é uma função idêntica em funcionalidade à PontoExtremo, que examina os vértices um a um, escolhendo o extremo.

PontoExtremo($v, inicio, fim, d$)

Se $fim - inicio \leq 8$

Retorne PontoExtremoLin($v, inicio, fim, d$) //Função que examina os pontos um a

um

$meio \leftarrow \lfloor (inicio + fim + 1)/2 \rfloor$

Se $d.x * v[inicio].x + d.y * v[inicio].y < d.x * v[meio].x + d.y * v[meio].y$

Se $d.x * v[meio].x + d.y * v[meio].y < d.x * v[meio + 1].x + d.y * v[meio + 1].y$

Retorne PontoExtremo($v, meio + 1, fim, d$)

Senão

Retorne PontoExtremo($v, inicio + 1, meio, d$)

Senão

Se $d.x * v[inicio].x + d.y * v[inicio].y < d.x * v[fim].x + d.y * v[fim].y$

Retorne PontoExtremo($v, meio + 1, fim, d$)

Senão

Retorne PontoExtremo($v, inicio, meio - 1, d$)

FIGURA 2.4. Solução do Problema 4

Entrada:

v : Vetor de números reais em ordem crescente.

f : Função $f : \mathbb{R} \rightarrow \mathbb{R}$

Saída:

Vetor v' tal que $v'[i] = j$ se existir j com $f(v[i]) = v[j]$ e $v'[i] = \varepsilon$ caso contrário.

FunçãoDeVetor(v, f)

Alocar vetor v' com n posições inteiras

Para i de 1 até n

$j \leftarrow \text{BuscaBinária}(v, 1, n, f(v[i]))$

Se $j = "x \notin v"$

$v[i] \leftarrow \varepsilon$

Senão

$v[i] \leftarrow j$

Retorne v'

FIGURA 2.5. Solução do Problema 5

TEOREMA 2.5. O algoritmo da figura 2.5 resolve corretamente o problema 5.

Vamos analisar a complexidade de tempo deste algoritmo. Temos um *loop* com n repetições. Cada repetição chama BuscaBinária com um vetor com n elementos. Como a complexidade

de tempo de pior caso da função BuscaBinária é $\Theta(\lg n)$, a complexidade total é $O(n \lg n)$. Podemos dizer que a complexidade de tempo é $\Theta(n \lg n)$ pois a busca binária pode levar o tempo de pior caso em todas as chamadas. Em toda esta análise consideramos que a função f pode ser computada em tempo $O(1)$. Com isto temos:

TEOREMA 2.6. *O algoritmo da figura 2.5 tem complexidade de tempo de pior caso $\Theta(n \lg n)$.*

Note que, caso o vetor de entrada v não estivesse ordenado, poderíamos ordená-lo em tempo $\Theta(n \lg n)$ antes de iniciarmos as buscas binárias. Esta ordenação não alteraria a complexidade assintótica do procedimento completo, que permaneceria $\Theta(n \lg n)$. Em muitos problemas em que a entrada não está ordenada da maneira que desejamos, vale a pena iniciarmos ordenando a entrada convenientemente.

2.5. Resumo e Observações Finais

A técnica de busca binária fornece algoritmos extremamente eficientes para diversos problemas. A idéia central é, a cada passo, descartarmos metade (ou alguma outra fração constante) dos elementos da entrada, examinando apenas um número constante de elementos. Isto é possível em casos onde a entrada é fornecida segundo alguma ordenação conveniente. Para buscarmos um elemento em um vetor ordenado, examinamos o elemento central do vetor, e assim podemos determinar qual a metade candidata a conter o elemento procurado. Em vetores ciclicamente ordenados, podemos proceder de forma semelhante, dividindo o vetor. No caso de polígonos convexos, usamos sempre o fato de que, ao unirmos dois vértices quaisquer de um polígono convexo, os dois novos polígonos obtidos também são convexos. Graças a isso, podemos usar a técnica de busca binária para resolver problemas como o ponto extremo de um polígono convexo em uma dada direção.

Em muitos problemas, a entrada não é fornecida ordenada. Pode valer a pena ordená-la, para que se possa usar a técnica de busca binária. Devido a alta performance prática dos algoritmos de ordenação e sua complexidade de $\Theta(n \lg n)$, pode-se pensar em um vetor ordenado como uma estrutura de dados extremamente simples e eficiente.

Um caso que não foi estudado aqui, é quando o número de elementos que podem conter a solução é desconhecido ou muito maior que a posição onde se espera encontrar a solução. Uma alternativa eficiente nesses casos é examinar os elementos segundo uma progressão geométrica. Chamamos este procedimento de busca ilimitada. Por exemplo, examinamos inicialmente o elemento v_4 , em seguida v_8 , v_{16} e assim por diante, até descobrirmos que o elemento que procuramos tem índice menor do que o examinado. Procedemos então com a busca binária tradicional. Nesse caso, o algoritmo é sensível a saída, tendo complexidade de tempo em função do índice do elemento procurado no vetor. Pode-se usar esta técnica, por exemplo, para encontrar o máximo de uma função.

Uma alternativa a busca binária é usarmos interpolação. Imagine que desejamos encontrar a palavra “bola” em um dicionário de 1000 páginas. Certamente não vamos começar examinando a página 500, mas sim uma página próxima do início, como 100. A busca por interpolação pode ser extremamente eficiente quando o vetor em que a busca é realizada tem estrutura previsível, porém, há casos em que a busca por interpolação tem complexidade de tempo linear, e não logarítmica como a busca binária, sendo muito ineficiente.

Uma aplicação geral de busca binária é quando desejamos encontrar o maior valor para o qual uma determinada propriedade é válida. Muitas vezes, é mais simples escrever um algoritmo que teste se a propriedade vale para um valor dado. Podemos então fazer uma busca ilimitada para encontrar o menor valor para o qual a propriedade falha.

Exercícios

- 2.1) Escreva o pseudo-código do algoritmo de busca binária em vetor sem usar recursão.

- 2.2) Ache o erro na demonstração abaixo, que prova que a complexidade da busca binária é $O(\lg \lg n)$:

Seja $T(n)$ o tempo do algoritmo de busca binária em um vetor com n elementos, no pior caso. Temos:

$$T(n) = T(\lceil (n-1)/2 \rceil) + O(1)$$

$$T(n) = O(1), n \leq 1$$

Vamos supor, para obter uma prova por indução, que $T(i) = O(\lg \lg n)$ para $i \leq n$. Vamos calcular $T(n+1)$. Temos: $T(n+1) = T(\lceil n/2 \rceil) + O(1) = O(\lg \lg(\lceil n/2 \rceil)) + O(1)$. Como $\lg \lg(\lceil n/2 \rceil) \leq \lg \lg(n+1)$ temos $T(n+1) = O(\lg \lg(n+1)) + O(1) = O(\lg \lg(n+1))$, finalizando a indução.

- 2.3) Escreva um algoritmo eficiente que receba como entrada um vetor $v = (v_1, \dots, v_n)$ de números inteiros e responda se existe $v_i = i$. Analise a complexidade de tempo do seu algoritmo e prove que ele funciona.
- 2.4) Projete um algoritmo que receba como entrada um polígono convexo P armazenado em um vetor e dois pontos u e v . O algoritmo deve retornar o vértice p_i de P que minimiza $\sphericalangle(u, v, p_i)$. A complexidade de tempo deve ser $O(\lg |P|)$. Prove que o seu algoritmo funciona.
- 2.5) Projete um algoritmo que receba como entrada um polígono convexo P armazenado em um vetor e um ponto u . O algoritmo deve responder se u está ou não no interior de P em tempo $O(\lg |P|)$.
- 2.6) Dados dois vetores de números reais em ordem crescente, escreva dois algoritmos, um deles baseado em busca binária e o outro não, para dizer se os dois vetores possuem algum elemento em comum. Analise a complexidade dos algoritmos em função de m e n , os tamanhos dos dois vetores. Quando é mais vantajoso usar cada um dos algoritmos?
- 2.7) Dados uma função real $f(x)$ e um valor α , o problema de achar uma raiz da função consiste em encontrar um valor de x tal que $|f(x)| < \alpha$. Escreva um algoritmo que resolve o problema usando busca binária caso $f(0) < 0$ e $f(1) > 0$. Escreva um algoritmo mais completo que resolva o problema para qualquer função que possua somente uma raiz, ou seja, existe apenas um valor de x tal que $f(x) = 0$. Nesse último caso, deve-se usar busca binária ilimitada em duas direções simultaneamente, e a complexidade de tempo deve depender do módulo do valor da raiz.
- 2.8) Neste exercício, o algoritmo que você deve projetar não é para ser usado por um computador. Embora a técnica de busca binária não apareça neste problema, o exercício trabalha análise de complexidade e conceitos de busca ilimitada. Imagine que você foi colocado em um corredor com infinitas portas para ambos os lados (pelo menos você não avista final). Você sabe que existe uma porta que leva a saída, mas não parece fácil encontrá-la, pois todas as portas que você abriu até então são fraudes, levando a uma parede de tijolos. Escreva um algoritmo que defina como você deve caminhar para examinar as portas, de modo a andar, no total, apenas $O(d)$ metros, onde d é a distância entre sua posição inicial e a porta que leva a saída.
- *2.9) Projete um algoritmo com complexidade de tempo sub-linear ($o(n)$) ou prove que isto é impossível. A entrada é um polígono convexo P com n vértices armazenado em um vetor e um ponto u .
- O algoritmo deve retornar o vértice de P mais próximo de u .
 - O algoritmo deve retornar o ponto do interior ou bordo de P mais próximo de u .

- *2.10) Modifique o procedimento BuscaBinária substituindo a linha “ $meio \leftarrow \lfloor (inicio + fim)/2 \rfloor$ ” por “ $meio \leftarrow$ variável aleatória inteira com distribuição uniforme de $inicio$ a fim ”. O algoritmo continua retornando sempre a resposta certa? Calcule a complexidade assintótica de $E[T(n)]$, o valor esperado do tempo do algoritmo para uma entrada de tamanho n .