Busca Binária

Encontros Linearítmicos ($n \lg n$)

Fernando Kiotheka

UFPR

08/05/2020

O que é?

Algoritmo que permite que você reduza um espaço de busca finito exponencialmente em um ponto, em ordem de lg ($\log_2$).

Pré-requisitos

Somente é necessário que seja possível dividir o espaço de busca em duas partições usando um predicado $P(i)$ onde i é um índice do espaço de busca. Definiremos que se $P(i)$ é `T`, estamos na esquerda e o objetivo está para a direita, se `F`, estamos na direita.

- Espaço de busca ordenado. Exemplo:

$$E_n := \mathbb{N}_{[1,8)} = \left[{}_0 1 {}_1 2 {}_2 3 {}_3 4 {}_4 5 {}_5 6 {}_6 7 {}_7\right]$$
$$P(i) := E_b[i] < 5$$

- Espaço de busca unimodal. Exemplo:

$$E_n := \left[n^2 : x \in \mathbb{N}_{[-3,3)}\right] = \left[{}_0 9 {}_1 4 {}_2 1 {}_3 0 {}_4 1 {}_5 4 {}_6\right]$$
$$P(i) := (i = 0 \vee E_b[i-1] > E_b[i])$$

Como?

A ideia é repetidamente pegar o meio do espaço de busca, verificar o predicado e reduzir o espaço de busca para um dos lados.
Por exemplo: $E_n := \mathbb{N}_{[1,8)}$ com $P(i) := E_n[i] \leq 2$:

$$E_n = \left[{}_0 1 {}_1 2 {}_2 3 {}_3 4 {}_4 5 {}_5 6 {}_6 7 {}_7\right]$$
$$E_n = \left[{}_0 1 {}_1 2 {}_2 3 {}_3 4 {}_4 5 {}_5 6 {}_6 7 {}_7\right]$$
$$E_n = \left[{}_0 1 {}_1 2 {}_2 3 {}_3 4 {}_4 5 {}_5 6 {}_6 7 {}_7\right]$$

Resultado: A posição 2 separa as duas metades.

Os detalhes

> *"Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky"*
>
> – Donald Knuth

É fácil errar por um (OB1) na hora de implementar o algoritmo.
O que vou explicar se baseia em intervalos semi-abertos $[0, n)$ que tem algumas propriedades interessantes, mas você pode optar por outra implementação que é mais intuitiva para você.
Uma variação comum por exemplo é usar intervalos fechados $[0, n-1]$, ou ainda algo maluco como $(-1, n)$. Depois da explicação, iremos estudar alguns outros exemplos.

Introdução do algoritmo

Temos um espaço de busca E_n:

$$E_n = \left[{}_0 1 \, {}_1 2 \, {}_2 3 \, {}_3 4 \, {}_4 5 \, {}_5 6 \, {}_6 7 \, {}_7 \right]$$

Aplicando a função binária $P(i) = E_n[i] \leq 2$ em E_n, obtemos E_b:

$$E_b = \left[{}_0 \mathtt{T} \, {}_1 \mathtt{T} \, {}_2 \mathtt{F} \, {}_3 \mathtt{F} \, {}_4 \mathtt{F} \, {}_5 \mathtt{F} \, {}_6 \mathtt{F} \, {}_7 \right]$$

Agora, trabalharemos neste novo espaço de busca.

Dividindo

Particionaremos o espaço de busca em subintervalos $[\texttt{lo}, \texttt{hi})$.

1. Começamos com $\texttt{lo} := 0$ e $\texttt{hi} := |E_b| = 7$.

$$E_{b[0,7)} = \left[{}_0\,\texttt{T}\ {}_1\,\texttt{T}\ {}_2\,\texttt{F}\ {}_3\,\texttt{F}\ {}_4\,\texttt{F}\ {}_5\,\texttt{F}\ {}_6\,\texttt{F}\ {}_7 \right]$$

2. Pegamos o ponto médio $\texttt{mi} := \lfloor (\texttt{hi} + \texttt{lo})/2 \rfloor = 3$.
3. Como $E_b[3] = \texttt{F}$, o objetivo está para a esquerda.
4. Então fazemos $\texttt{hi} := \texttt{mi}$, reduzindo a busca pela metade.

$$E_{b[0,3)} = \left[{}_0\,\texttt{T}\ {}_1\,\texttt{T}\ {}_2\,\texttt{F}\ {}_3 \right]$$

Mas ainda não acabamos!

Dividindo de novo

1. Temos $\texttt{lo} := 0$ e $\texttt{hi} := 3$ da iteração anterior.

$$E_{b[0,3)} = \left[_0\,\texttt{T}\,_1\,\texttt{T}\,_2\,\texttt{F}\,_3\right]$$

2. Pegamos o ponto médio $\texttt{mi} := \lfloor(\texttt{hi} + \texttt{lo})/2\rfloor = 1$.
3. Como $E_b[1] = \texttt{T}$, o objetivo está para a direita.
4. Então fazemos $\texttt{lo} := \texttt{mi} + 1$, reduzindo a busca pela metade.

$$E_{b[2,3)} = \left[_2\,\texttt{F}\,_3\right]$$

Mas ainda não chegamos a um único ponto!

Dividindo pela última vez

1. Temos $\texttt{lo} := 2$ e $\texttt{hi} := 3$ da iteração anterior.

$$E_{b[2,3)} = \left[_{2}\ \texttt{F}\ _{3}\right]$$

2. Pegamos o ponto médio $\texttt{mi} := \lfloor(\texttt{hi} + \texttt{lo})/2\rfloor = 2$.
3. Como $E_b[2] = \texttt{F}$, o objetivo está para a esquerda.
4. Então fazemos $\texttt{hi} := \texttt{mi}$, reduzindo a busca pela metade.

$$E_{b[2,2)} = \left[_{2}\right]$$

Chegamos então a um ponto que divide o espaço de busca, 2.

$$E_b = \left[_{0}\ \texttt{T}\ _{1}\ \texttt{T}\ _{2}\ \texttt{F}\ _{3}\ \texttt{F}\ _{4}\ \texttt{F}\ _{5}\ \texttt{F}\ _{6}\ \texttt{F}\ _{7}\right]$$

Uma das implementações possíveis

Código 1 `intervalo-0-n.cpp`

```
int lo = 0;
int hi = n;
while (lo != hi) {
    // reescrito para evitar overflow
    int mi = lo + (hi - lo) / 2;
    if (P(mi)) {
        // direita
        lo = mi + 1;
    } else {
        // esquerda
        hi = mi;
    }
}
return lo; // ou hi, para o intervalo [0, n) tanto faz
```

Variações de implementação

Eu não uso o código anterior. Isso porque são possíveis algumas alterações para melhorar sua intuição:

- Alterar de `while (lo != hi)` para `while (lo < hi)` (não muda nada, mas talvez fique mais bonito)
- Alterar de `P(mi)` para `!P(mi)` e reordenar o `if` (acho mais intuitivo pensar primeiro esquerda, depois direita)
- Alterar o intervalo de $[0, n)$ para $[0, n-1]$ ou $(-1, n)$ como veremos a seguir
- Ou ainda iterar sobre passos binários, como também veremos

Implementação $[0, n-1]$

Variação frequentemente encontrada. Ao final, `lo` $=$ `hi` $+ 1$.

Código 2 `intervalo-0-n-1.cpp`

```
int lo = 0;
int hi = n - 1;
while (lo <= hi) {
    int mi = lo + (hi - lo) / 2;
    if (P(mi)) {
        lo = mi + 1;
    } else {
        hi = mi - 1;
    }
}
return lo;
```

Implementação $[0, n-1]$ com variável auxiliar

O Errichto recomenda usar uma variável auxiliar para não errar.

Código 3 `intervalo-0-n-1-aux.cpp`

```
int lo = 0;
int hi = n - 1;
int ans;
while (lo <= hi) {
    int mi = lo + (hi - lo) / 2;
    if (P(mi)) {
        lo = mi + 1;
    } else {
        ans = mi;
        hi = mi - 1;
    }
}
return ans;
```

Implementação $(-1, n)$

Variação maluca. Sem mais +1s e -1s, ao final $\texttt{lo} + 1 = \texttt{hi}$.

Código 4 `intervalo--1-n.cpp`

```
int lo = -1;
int hi = n;
while (hi - lo > 1) {
    int mi = lo + (hi - lo) / 2;
    if (P(mi)) {
        lo = mi;
    } else {
        hi = mi;
    }
}
return hi;
```

Implementação de passo binário

Não é incomum, a sutileza encanta. Permite fazer múltiplas buscas num laço só, diferente do normal que fica ilegível ao combinar laços.

Código 5 `passo-binario-reverso.cpp`

```
int i = n;
for (int p = n; p >= 1; p /= 2) {
    while (i-p >= 0 && !P(i-p)) { i -= p; }
}
return i;
```

Código 6 `passo-binario.cpp`

```
int i = -1;
for (int p = n; p >= 1; p /= 2) {
    while (i+p < n && P(i+p)) { i += p; }
}
return i + 1;
```

Vamos aos predicados!

Todas as implementações descritas anteriores entregam o mesmo resultado dado o predicado certo, mas qual o predicado certo?
Depende do problema que a gente quer resolver.
Exemplos comuns que podem ser resolvidos em $\mathcal{O}(\lg n)$ são:

- Posição à esquerda ou à direita de valor em vetor ordenado
- Contagem de ocorrências de valor em vetor ordenado
- Posição de valor em vetor ordenado deslocado sem duplicados
- Posição de máximo/mínimo em vetor unimodal
- Raíz quadrada de um número (método de bisecção)
- i-ésimo elemento de união de vetores ordenados
- Elemento em vetor ordenado que é igual ao seu índice
- Otimização de problemas com solução gananciosa conhecida

Posição à esquerda ou à direita de valor

Esses são os predicados mais importantes para buscar elementos. Eles são tão importantes que uma versão que procura em vetores está disponível na biblioteca padrão:

- `lower_bound`(início, fim, valor): $P(i) := v[i] < x$
- `upper_bound`(início, fim, valor): $P(i) := v[i] \leq x$

Exemplo

$$v = \left[{}_0 1\, {}_1 2\, {}_2 2\, {}_3 3\, {}_4 3\, {}_5 5\, {}_6 5\, {}_7\right]$$

Quando o valor existe, retornam o intervalo de elementos $(a, b]$.

- `lower_bound`$(v, v + n, 3) - v = 3$
- `upper_bound`$(v, v + n, 3) - v = 5$

Quando está ausente, retornam o lugar onde ele deveria estar.

- `lower_bound`$(v, v + n, 4) - v = 5$
- `upper_bound`$(v, v + n, 4) - v = 5$

Contagem de ocorrências de valor

Fica fácil contar quantas ocorrências de um valor existem:

$$v = \left[{}_0 1\, {}_1 2\, {}_2 3\, {}_3 3\, {}_4 3\, {}_5 5\, {}_6 5\, {}_7\right]$$

No caso do elemento 3:

- $\texttt{lower\_bound}(v, v+n, 3) - v = 2$
- $\texttt{upper\_bound}(v, v+n, 3) - v = 5$
- $n_3 = 5 - 2 = 3$

E no caso de um elemento inexistente:

- $\texttt{lower\_bound}(v, v+n, 4) - v = 5$
- $\texttt{upper\_bound}(v, v+n, 4) - v = 5$
- $n_4 = 5 - 5 = 0$

Isso é simples devido ao uso de intervalos $[0, n)$:

- A diferença dos limites é o tamanho do intervalo.

Contagem de ocorrências de valor (continuado)

A implementação do passo binário, por exemplo, nos permite juntar as duas buscas em um código só:

Código 7 `contagem-passo-binario.cpp`

```
int l = n;
int r = n;

for (int p = n; p >= 1; p /= 2) {
    while (l-p >= 0 && !(v[l-p] <  x)) { l -= p; }
    while (r-p >= 0 && !(v[r-p] <= x)) { r -= p; }
}

return r - l;
```

Posição de valor

Uma outra coisa simples para se fazer com esses predicados é devolver uma posição onde um valor aparece.
Dado que $n \neq 0$, podemos verificar a posição dada pelo predicado $P(i) := v[i] < x$ (`lower_bound`). Ela será a mais à esquerda.

Exemplo

$$v = \left[{}_0 1 \, {}_1 2 \, {}_2 2 \, {}_3 3 \, {}_4 3 \, {}_5 5 \, {}_6 5 \, {}_7\right]$$

- `lower_bound`$(v, v + n, 3) - v = 3$. $v[3] = 3$.
- `lower_bound`$(v, v + n, 4) - v = 5$. $v[5] \neq 4$.
- `lower_bound`$(v, v + n, 5) - v = 5$. $v[5] = 5$.

Posição de valor (continuado)

Como isso é frequente, existe uma função da biblioteca padrão para tal, que caso não encontre o elemento, retorna o final do vetor:

- `binary_search`(início, fim, valor)

Isso também pode ser codificado em uma variação da busca binária:

Código 8 `posicao.cpp`

```
int lo = 0;
int hi = n;
while (lo != hi) {
    int mi = lo + (hi - lo) / 2;
    if (a[mi] < x) { lo = mi + 1; }
    else if (a[mi] == x) { return mi; }
    else { hi = mi; }
}
return NAO_ACHEI;
```

O problema dos elementos duplicados

Imagine que queremos achar o mínimo em um vetor ordenado que foi deslocado (ou alternativamente, um vetor que é decrescente e depois é crescente como uma parábola).

$$v_1 = \left[{}_0 2\ {}_1 2\ {}_2 2\ {}_3 2\ {}_4 2\ {}_5 0\ {}_6 2\ {}_7\right]$$
$$v_2 = \left[{}_0 2\ {}_1 0\ {}_2 2\ {}_3 2\ {}_4 2\ {}_5 2\ {}_6 2\ {}_7\right]$$

O algoritmo começa verificando o meio do vetor, mas para qual lado devemos prosseguir?

- Não podemos usar as bordas como apoio pois elas são iguais ao elemento escolhido
- Pensando num caso geral, é impossível saber em qual lado o mínimo se encontra sem verificar todos os elementos

Então aqui temos um problema no mínimo $\mathcal{O}(n)$ por natureza (a menos que existam mais restrições que possam nos ajudar).

Posição de valor em vetor deslocado sem duplicados

Vamos pegar dois exemplos:

$$v_1 = \left[{}_0 4\, {}_1 5\, {}_2 6\, {}_3 7\, {}_4 9\, {}_5 1\, {}_6 2\, {}_7\right]$$
$$v_2 = \left[{}_0 8\, {}_1 2\, {}_2 3\, {}_3 4\, {}_4 5\, {}_5 6\, {}_6 7\, {}_7\right]$$

Primeiro, resolvemos o problema do vetor estar deslocado.
Queremos um predicado que divida o vetor desta forma:

$$v_{1b} = \left[{}_0 \mathrm{T}\, {}_1 \mathrm{T}\, {}_2 \mathrm{T}\, {}_3 \mathrm{T}\, {}_4 \mathrm{T}\, {}_5 \mathrm{F}\, {}_6 \mathrm{F}\, {}_7\right]$$
$$v_{2b} = \left[{}_0 \mathrm{T}\, {}_1 \mathrm{F}\, {}_2 \mathrm{F}\, {}_3 \mathrm{F}\, {}_4 \mathrm{F}\, {}_5 \mathrm{F}\, {}_6 \mathrm{F}\, {}_7\right]$$

Podemos se ancorar nas bordas para fazer predicados:

$$P(i) := v[0] \leq v[i]$$
$$P(i) := v[n-1] < v[i]$$

Assim obtemos dois vetores ordenados para procurar o valor.

Posição de máximo/mínimo em vetor unimodal

Uma função unimodal é aquela que tem um único ponto de máximo ou mínimo. Desde que seja sempre possível saber se um ponto está no lado crescente ou decrescente, podemos achá-lo.

$$v_{\text{min}} = \left[{}_0 4\ {}_1 3\ {}_2 2\ {}_3 1\ {}_4 2\ {}_5 4\ {}_6 5\ {}_7\right]$$
$$v_{\text{max}} = \left[{}_0 1\ {}_1 2\ {}_2 4\ {}_3 5\ {}_4 4\ {}_5 3\ {}_6 2\ {}_7\right]$$

Como nesse vetor não há elementos duplicados lado a lado, é possível sempre saber se estamos na parte decrescente ou crescente:

$$P_{\text{min}}(i) := (i + 1 < n \wedge v[i] > v[i + 1])$$
$$P_{\text{max}}(i) := (i + 1 < n \wedge v[i] < v[i + 1])$$

Assim obtemos essa divisão:

$$v_{\text{min}b} = v_{\text{max}b} = \left[{}_0 \mathtt{T}\ {}_1 \mathtt{T}\ {}_2 \mathtt{T}\ {}_3 \mathtt{F}\ {}_4 \mathtt{F}\ {}_5 \mathtt{F}\ {}_6 \mathtt{F}\ {}_7\right]$$

Que retorna o índice indexado em 0 do mínimo ou máximo.

Trabalhando com números reais...

Vimos como achar posição de máximo/mínimo em um vetor, mas nos problemas é bem mais comum fazermos uma otimização em números reais. O método é exatamente o mesmo, porém o predicado verifica um valor real. Sugestões:

- Sempre que possível evite usar `double`, `float` e variados. Geralmente o problema pede uma precisão específica, então utilize inteiros que representam precisões maiores (1000 para representar 10.00)
- Se você optar pelo uso de números reais, cuidado com a função de comparação. Para comparar números reais é preciso sempre pensar em uma constante mínima (ε)
- Alternativamente, é frequente rodar a busca binária uma quantidade fixa de vezes a fim de que o valor convirja para um valor específico.

Exemplo: Achar a raíz quadrada com ε

Código 9 `raiz-quadrada-epsilon.cpp`

```
double lo = 0;
double hi = x;
while (hi - lo > EPS) {
    double mi = lo + (hi - lo) / 2;
    if (mi*mi < x) {
        lo = mi;
    } else {
        hi = mi;
    }
}
return lo;
```

ε	1e-1	1e-2, 1e-3	1e-4	1e-5, 1e-6, 1e-7
Valor	1.375	1.41406	1.41418	1.41421

Exemplo: Achar a raíz quadrada com iterações

Código 10 `raiz-quadrada-iteracoes.cpp`

```
double lo = 0;
double hi = x;
for (int i = 0; i < IT; i++) {
    double mi = lo + (hi - lo) / 2;
    if (mi*mi < x) {
        lo = mi;
    } else {
        hi = mi;
    }
}
return lo;
```

Iterações	1	3	4	8	14	18
Valor	1	1.25	1.375	1.41406	1.41418	1.41421

O resto desse papo

Outros exemplos ficarão como exercício para o leitor. Literalmente. Preparei um contest com uma boa quantidade de exercícios para praticar a busca binária em vários exemplos diferentes, boa sorte!

- Querem ler alguns dos problemas e tirar dúvidas?
- Todos os slides e códigos estão no site