Programação Procedimental

O Problema da Busca

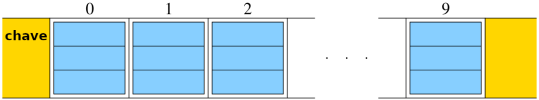
Aula 12

1 de Outubro de 2021

Roteiro

O Problema da Busca

- Temos uma coleção de elementos, onde cada elemento possui um identificador (chave única), e recebemos uma chave para busca.
- Devemos encontrar o elemento da coleção que possui a mesma chave ou identificar que ele não existe.



O Problema da Busca

Observações:

- Nos nossos exemplos usaremos um vetor de inteiros como a coleção.
  - O valor da chave será o próprio valor de cada número.

0	1	2	3	4	5	6	7	8	9
20	5	15	24	67	45	1	76		

- Os algoritmos servem para qualquer coleção de elementos que possuam chaves que possam ser comparadas.

O Problema da Busca

Vamos criar a função:

```
int busca_sequencial(int *dados, int n, int x); //x é a chave de busca
```

- A função deve retornar o índice do vetor que contém a chave ou -1 caso a chave não esteja no vetor.

O Problema da Busca

chave = 45 tam = 8

vet	20	5	15	24	67	45	1	76		
	0	1	2	3	4	5	6	7	8	9

chave = 100 tam = 8

vet	20	5	15	24	67	45	1	76		
	0	1	2	3	4	5	6	7	8	9

No primeiro exemplo, a função deve retornar 5, no segundo exemplo a função deve retornar -1.

O Problema da Busca

A busca sequencial é o algoritmo mais simples de busca:

- Percorre todo o vetor comparando a chave com o valor de cada posição.
- Se for igual para alguma posição, devolve esta posição.
- Se **todo o vetor** foi percorrido então devolva -1.

Busca Sequencial

```c
int busca_sequencial(int *dados, int n, int x) {
  int i;
  for (i = 0; i < n; i++)
    if (v[i] == x)
      return i;
  return -1;
}
```

Busca Sequencial

```c
#include <stdio.h>

int busca_sequencial(int *dados, int n, int x);

int main(){
  int pos, vet[] = {20, 5, 15, 24, 67, 45, 1, 76, -1, -1}; //-1 indica
                                                            //posição não usada
  pos = buscaSequencial(vet, 8, 45);
  if(pos != -1) printf("A posicao da chave 45 no vetor é: %d\n", pos);
  else printf("A chave 45 não está no vetor! \n");

  pos = buscaSequencial(vet, 8, 100);
  if(pos != -1) printf("A posicao da chave 100 no vetor é: %d\n", pos);
  else printf("A chave 100 não está no vetor! \n");
}

int busca_sequencial(int *v, int n, int x) {
  int i;
  for (i = 0; i < n; i++)
    if (v[i] == x)
      return i;
  return -1;
}
```

Roteiro

O Problema da Busca

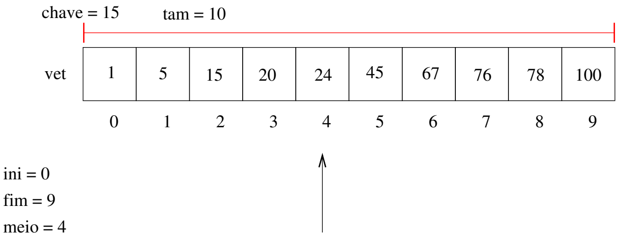
A busca binária é um algoritmo um pouco mais sofisticado.

- Vamos assumir que o vetor está ordenado, `l = 0` e `r = n-1`:

0	1	2	3	4	5	6	7	8	9
1	5	15	20	24	45	67	76		

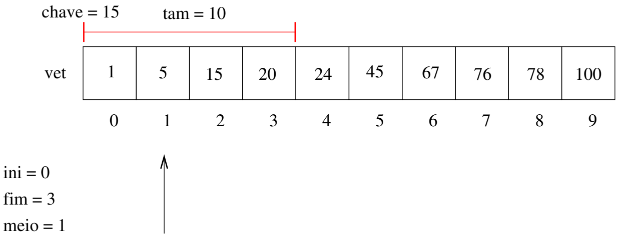
- Compare a chave de busca com o valor da posição do meio do vetor, isto é, `dados[m]` com `m=(l+r)/2`.
  1. Caso `dados[m] == x`, devolva esta posição.
  2. Caso `dados[m] > x`, então repita o processo mas considere a primeira metade do vetor.
  3. Caso `dados[m] < x`, conside a segunda metade para a próxima etapa da busca.

Busca Binária



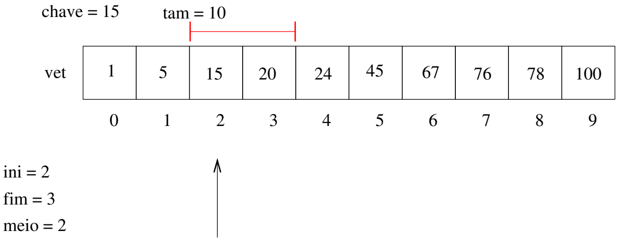
Como o valor da posição do meio é maior que a chave, atualizamos **fim** do vetor considerado.

Busca Binária



Como o valor da posição do meio é menor que a chave, atualizamos **ini** do vetor considerado.

Busca Binária



Finalmente encontramos a chave e podemos devolver sua posição 2.

Busca Binária

Código recursivo:

```c
int busca_binaria(int *dados, int l, int r, int x) {
  int m = (l + r) / 2;
  if (l > r) return -1; /*caso base*/
  if (dados[m] == x) return m; /*caso base*/
  else if (dados[m] < x)
    return busca_binaria(dados, m + 1, r, x);
  else
    return busca_binaria(dados, l, m - 1, x);
}
```

0	1	2	3	4	5	6	7	8	9
1	5	15	20	24	45	67	76		

Busca Binária

Código não recursivo:

```c
int busca_binaria_iterativa(int *dados, int n, int x){
  int ini=0, fim=n-1, meio;
  while(ini <= fim){ //enquanto o vetor tiver pelo menos 1 elemento
    meio = (ini+fim)/2;
    if(dados[meio] == x) return meio;
    else if(dados[meio] > x) fim = meio - 1;
    else ini = meio + 1;
  }
return -1;
}
```

0	1	2	3	4	5	6	7	8	9
1	5	15	20	24	45	67	76		

Busca Binária

```c
#include <stdio.h>

int busca_binaria(int *dados, int l, int r, int x);

int main(){
  int vet[] = {1, 5, 15, 20, 24, 45, 67, 76, 78, 100};
  int pos = busca_binaria(vet, 0, 9, 15);
  if(pos != -1)
    printf("A posicao da chave 15 no vetor é: %d\n", pos);
  else
    printf("A chave 15 não está no vetor! \n");
}

int busca_binaria(int *dados, int l, int r, int x) {
  int m = (l + r) / 2;
  if (l > r) return -1; /*caso base*/
  if (dados[m] == x) return m; /*caso base*/
  else if (dados[m] < x)
    return busca_binaria(dados, m + 1, r, x);
  else
    return busca_binaria(dados, l, m - 1, x);
}
```

O Problema da Busca

Busca Binária **vs.** busca sequencial:

- Qual solução é a mais eficiente?

Roteiro

Busca sequencial e consumo de tempo

Quantos segundos demora para executar a seguinte função?

```
1 int busca_sequencial(int *v, int n, int x) {
2   int i;
3   for (i = 0; i < n; i++)
4     if (v[i] == x)
5       return i;
6 return -1;
7 }
```

Depende...

- do computador onde ele for rodado
- da posição de x no vetor
  - no melhor caso, a linha 4 é executada 1 vez
  - no pior caso, a linha 4 é executada n vezes
- do valor de n
  - $n = 10$ vs $n = 10.000$

Busca sequencial e consumo de tempo

Queremos analisar algoritmos:

- Independentemente do computador onde ele for rodado
- Em função do valor de n (a quantidade de dados)

Em geral, queremos analisar o pior caso do algoritmo

- A análise do melhor caso, em geral, não é interessante
- A análise do caso médio é mais difícil
  - Normalmente é uma análise probabilística, precisamos fazer suposições sobre os dados de entrada

Busca sequencial e consumo de tempo

```
1 int busca_sequencial(int *v, int n, int x) {
2   int i;
3   for (i = 0; i < n; i++)
4     if (v[i] == x)
5       return i;
6   return -1;
7 }
```

Consumo de tempo por linha no pior caso:

- Linha 2: tempo c_2 (declaração de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
  - No pior caso, essa linha é executada $n + 1$ vezes
- Linha 4: tempo c_4 (acessos, comparação e `if`)
  - No pior caso, essa linha é executada n vezes
- Linha 5: tempo c_5 (acesso e `return`)
- Linha 6: tempo c_6 (`return`)

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6$$

Busca sequencial e consumo de tempo

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n+1) + c_4 \cdot n + c_5 + c_6$$

Cada c_i não depende de n, depende apenas do computador

- Leva um tempo constante

Sejam $a := c_2 + c_3 + c_5 + c_6$, $b := c_3 + c_4$ e $d := a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

$$\begin{aligned} c_2 + c_3 \cdot (n+1) + c_4 \cdot n + c_5 + c_6 &= c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n \\ &= a + b \cdot n \leq a \cdot n + b \cdot n = d \cdot n \end{aligned}$$

Isto é, o crescimento do tempo é linear em `n`

- Se n dobra, o tempo de execução praticamente dobra

Notação Assintótica

Como vimos, existe uma constante d tal que, para $n \geq 1$,

$$c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 \leq d \cdot n$$

d não interessa tanto, depende apenas do computador...

- Estamos preocupados em uma estimativa

O tempo do algoritmo é da ordem de n

- A ordem de crescimento do tempo é igual a de $f(n) = n$

Dizemos que

$$T(n) = c_2 + c_3 \cdot (n + 1) + c_4 \cdot n + c_5 + c_6 = \mathrm{O}(n)$$

Veremos uma definição formal de $\mathrm{O}(\cdot)$ em breve...

Busca Binária

```c
int busca_binaria(int *dados, int l, int r, int x) {
  int m = (l + r) / 2;
  if (l > r) return -1;
  if (dados[m] == x) return m;
  else if (dados[m] < x)
    return busca_binaria(dados, m + 1, r, x);
  else
    return busca_binaria(dados, l, m - 1, x);
}
```

Quantas chamadas realizamos no pior caso?

- primeiro chamamos para um vetor de tamanho n
- depois para $n/2$, $n/4$, $n/8$, ..., ??
- no pior caso, só paramos quando o (sub)vetor tem 1 elemento
  - Ou seja, após $\log_2 n = \lg n$ chamadas
- gastamos um tempo constante c em cada chamada

Para $n \geq 1$, o consumo de tempo é no máximo:

- $c + c \lg n \leq 2c \lg n = \mathrm{O}(\lg n)$

Objetivos

Temos dois objetivos ao analisar algoritmo

- Entender o tempo de execução de um algoritmo
  - Exemplo: busca linear é $\mathrm{O}(n)$
  - Vamos dizer que o algoritmo é $\mathrm{O}(f(n))$
- Comparar dois algoritmos
  - Busca linear é $\mathrm{O}(n)$ **vs.** busca binária é $\mathrm{O}(\lg n)$
  - Prova formal que um algoritmo é melhor que o outro

Comparando funções

Queremos comparar duas funções f e g

- Queremos entender a velocidade de crescimento de f
- Queremos dizer que f cresce **mais lentamente ou igual** a g

f pode ser o tempo de execução do algoritmo e g uma função mais simples de entender

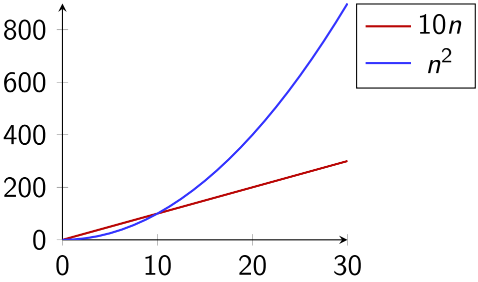
- $f(n) = c_2 + c_3 \cdot (n+1) + c_4 \cdot n + c_5 + c_6$ e $g(n) = n$
- $f(n) = 3n^2 + 10 \lg n$ e $g(n) = n^2$

f e g podem ser os tempos de execução de dois algoritmos

- $f(n) = dn$ e $g(n) = c + c \lg n$

Primeira Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para todo n



Problema: $10n > n^2$ para $n < 10$

Solução: Comparar apenas n suficientemente grande

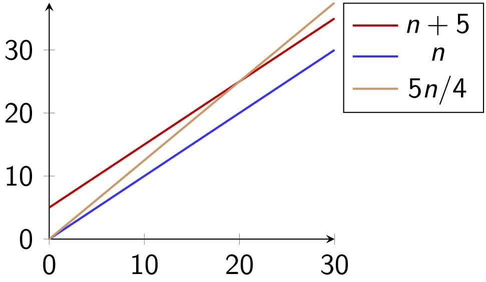
- Para todo $n \geq n_0$ para algum n_0

Segunda Ideia

Comparar funções verificando se $f(n) \leq g(n)$ para $n \geq n_0$

Problema: $n + 5 > n$ para todo n

- Mas a velocidade de crescimento das funções é o mesmo
- Vamos ignorar constantes e termos menos importantes



Notação Assintótica

Dizemos que uma função $f(n) = \mathrm{O}(g(n))$ se

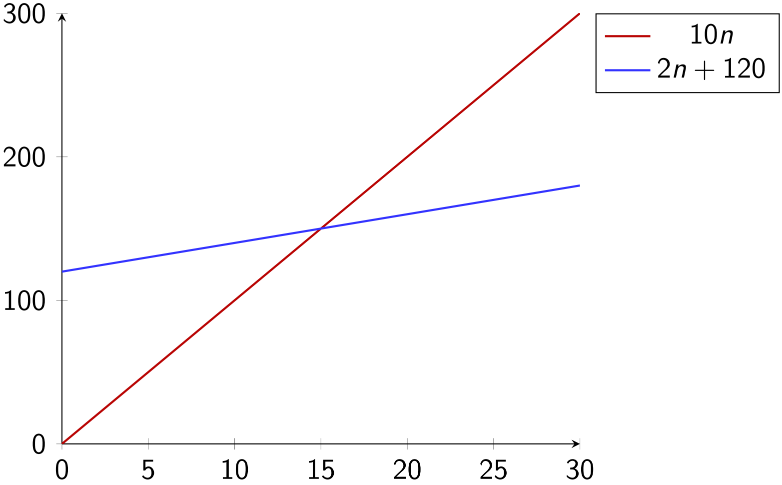
- existe uma constante c
- existe uma constante n_0

tal que

$$f(n) \leq c \cdot g(n), \qquad \text{para todo } n \geq n_0$$

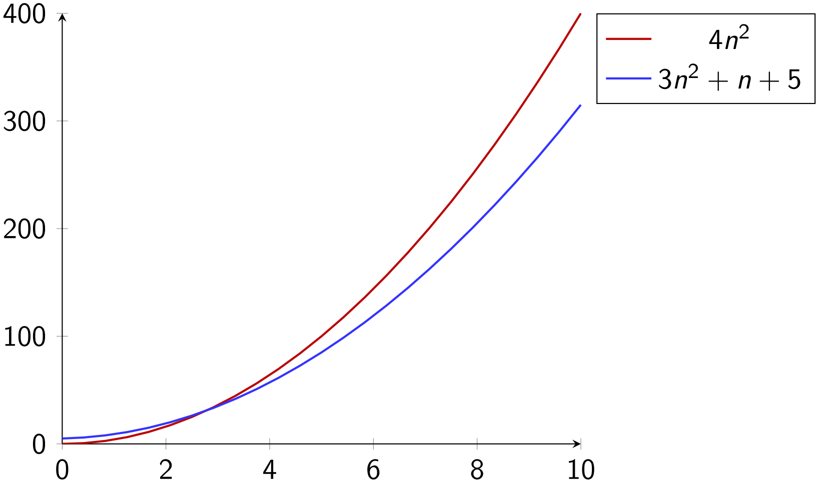
$f(n) = \mathrm{O}(g(n))$ se, para todo n suficientemente grande, $f(n)$ é menor ou igual a um **múltiplo de $g(n)$**

Exemplo: $2n + 120 = \mathrm{O}(n)$



Basta escolher, por exemplo, $c = 10$ e $n_0 = 15$

Exemplo: $3n^2 + n + 5 = \mathrm{O}(n^2)$



Basta escolher, por exemplo, $c = 4$ e $n_0 = 4$

Outros exemplos

$$
\begin{aligned}
1 &= \mathrm{O}(1) \\
1.000.000 &= \mathrm{O}(1) \\
5n + 2 &= \mathrm{O}(n) \\
5n^2 + 5n + 2 &= \mathrm{O}(n^2) \\
\log_2 n &= \mathrm{O}(\log_{10} n) \\
\log_{10} n &= \mathrm{O}(\log_2 n)
\end{aligned}
$$

Nomenclatura e consumo de tempo

- $O(1)$: tempo constante
  - não depende de n
  - Ex: atribuição e leitura de uma variável
  - Ex: operações aritméticas: `+`, `-`, `*`, `/`
  - Ex: comparações (`<`, `<=`, `==`, `>=`, `>`, `!=`)
  - Ex: operadores booleanos (`&&`, `&`, `||`, `|`, `!`)
  - Ex: acesso a uma posição de um vetor

- $O(\lg n)$: logarítmico
  - quando n dobra, o tempo aumenta em uma constante
  - Ex: Busca binária

Nomenclatura e consumo de tempo

- $\mathrm{O}(n)$: linear
  - quando n dobra, o tempo dobra
  - Ex: Busca linear
  - Ex: Encontrar o máximo/mínimo de um vetor
- $\mathrm{O}(n \lg n)$:
  - quando n dobra, o tempo um pouco mais que dobra
  - Ex: algoritmos de ordenação que veremos
- $\mathrm{O}(n^2)$: quadrático
  - quando n dobra, o tempo quadriplica
  - Ex: BubbleSort, SelectionSort e InsertionSort
- $\mathrm{O}(n^3)$: cúbico
  - quando n dobra, o tempo octuplica
  - Ex: multiplicação de matrizes $n \times n$

Um cuidado

O que significa dizer que o tempo de um algoritmo é $\mathrm{O}(n^3)$?

- Para instâncias grandes ($n \geq n_0$)
- O tempo é menor ou igual a um múltiplo de n^3

Pode ser que o tempo do algoritmo seja $2n^2$...

- $2n^2 = \mathrm{O}(n^3)$, mas...
- $2n^2 = \mathrm{O}(n^2)$

Ou seja, podemos ter feito uma análise "folgada"

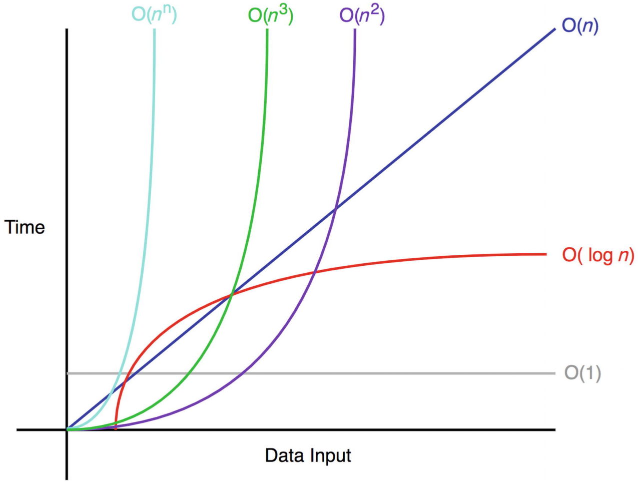
- achamos que o algoritmo é muito pior do que é realmente

No curso, não faremos análises "folgadas"

- existe uma maneira formal de lidar com isso (notações Ω e Θ)
- mas não precisamos desse formalismo em PP

Eficiência dos Algoritmos

Por que isso importa?



Eficiência dos Algoritmos

Por que isso importa?

ops \ n	10	20	50	100	500	1000
$10000n$	0.0001	0.0002	0.0005	0.001	0.005	0.01
$1000n \log n$	0.00003	0.00009	0.0003	0.0007	0.004	0.01
$100n^2$	0.00001	0.00004	0.0003	0.001	0.03	0.1
$10n^3$	0.00001	0.00008	0.001	0.01	1.3	10
$2n^4$	0.00002	0.0003	0.01	0.2	125	0.5 horas
$n^{\log n}$	0.000002	0.0004	4	5.4 horas	10^5 séc.	
2^n	0.000001	0.001	13 dias	10^{11} séc.		
3^n	0.00006	3	10^5 séc.			
$n!$	0.004	77 anos				

[1] Supondo 10^9 operações de algoritmo por segundo.

Roteiro

Eficiência dos Algoritmos

Para se ter uma idéia da diferença de eficiência dos dois algoritmos, considere que temos 10^6 (um milhão) de itens.

- Com a busca sequencial, a procura de um item qualquer gastará na média

$$(10^6 + 1)/2 \approx 500000 \text{ acessos.}$$

- Com a busca binária teremos

$$(\log_2 10^6) - 1 \approx 20 \text{ acessos.}$$

Mas uma ressalva deve ser feita: para utilizar a busca binária, o vetor **precisa estar ordenado!**

- Se você tiver um cadastro onde vários itens são atualizados com **frequência**, a busca binária pode não ser a melhor opção, já que você precisará manter o vetor ordenado.

Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides dos Profs. Rafael Schouery e Lehilton L. C. Pedrosa, da UNICAMP.