Entendendo Algoritmos

Nicolas Ramos Carreira

Sumário

Capítulo 0

Sobre o livro

O livro, em seu inicio, já destaca qual a sua ideia central, que é: Ser um livro de fácil leitura, que irá pegar conteúdos complexos e simplificar através das explicações,exemplos, ilustrações e prática ao longo do livro.O livro deixa claro que não aborda todos os algoritmos existentes, mas sim os mais importantes e considerado úteis pelo autor.

0.1 Panorama geral dos capitulos

Os primeiros 3 capítulos do livro se constituirão no seguinte:

- Capitulo 1: Aprenderemos o nosso primeiro algoritmo básico, a busca binária. Além disso, veremos como analisar a velocidade de um algoritmo utilizando a notação Big-O (que será utilizada ao longo do livro inteiro)
- Capitulo 2: Aprenderemos duas estruturas de dados fundamentais, que são os arrays e listas encadeadas. Elas também são usadas na criação de estruturas de dados mais avançadas, como a tabela hash
- Capitulo 3: Aprenderemos o uso da recursão, uma técnica muito útil utilizada em muitos algoritmos

Os capitulos acima são os mais importantes (principalmente por conta da notação Big-O e da recursão), portanto, são os que o autor vai em ritmo mais lento.

O restante do livro apresenta alguns algoritmos de aplicação mais ampla. Veja:

- Tecnicas para resolução de problemas: Abordadas nos capítulos 4, 8, 9. São abordadas técnicas, como divisão e conquista (cap 4), programação dinamica (cap 9) e algoritmo guloso (cap 8) para problemas que não sabemos bem como resolvê-lo de forma eficiente.

- Tabela Hash: É uma estrutura de dados muito útil que é abordada no capitulo 5
- Algoritmos de grafos: Abordados nos capitulos 6 e 7, grafos são uma maneira de modelar uma rede. Veremos sobre a pesquisa em largura (cap 6) e algoritmo de Dijkstra (cap 7)
- K-vizinhos mais próximos: Abordado no capitulo 7, essa é uma técnica simples de aprendizado de maquina. Podemos utiliza-la para criar recomendações de sistema, mecanismo OCR e até um sistema para prever valores (ou seja, tudo que envolve prever um valor).
- Proximos passos: O capitulo 11 é discorrido sobre dez algoritmos que valem a pena uma leitura posterior (quando você ja estiver craque em algoritmos)

0.2 Como usar o livro

O autor se preocupou bastante com a ordem com que os assuntos seriam abordados, sendo assim, o ideal é que se leia os capitulos em ordem (eles se baseiam uns nos outros).

Execute o código dos exemplos. Isso te fará reter melhor os conteúdos abordados. Você pode baixa-los no github através DESTE LINK (nesse repositório, o autor disponibilizou os códigos em várias linguagens, como C#, Python, Ruby..). Uma observação é que os exemplos abordados utilizam Python como linguagem.

Obviamente que é primordial que os exercicios passados sejam feitos. Eles nos ajudarão a conferir nosso pensamento (se estamos seguindo a linha de raciocinio correta ou não)

OBS: Um detalhe é que EU, Nicolas, gostaria de deixar registrado a forma de estudo que eu usei para estudar o livro. Além de fazer o que foi falado acima, para os conteúdos teóricos, irei ler, entender e depois passar por escrito aqui para o LATEX

0.3 Quem deve ler o livro

É destinado a qualquer um que queira aprender sobre programação e imergir no mundo dos algoritmos

Capítulo 1

Introdução a algoritmos

Neste capítulo, aprenderemos:

- Como fazer uma busca binária
- Entender o uso da notação Big-O para analisar a velocidade de algoritmos

1.1 Introdução

Um algoritmo é o conjunto de instruções para realizar determinada tarefa. Cada trecho de um código poderia ser um algoritmo, mas este livro se concentra nos mais importantes. Os algoritmos apresentados no livro foram escolhidos porque são rápidos ou porque resolver problemas interessantes.

Em cada um dos casos, o autor irá descrever o algoritmo e apresentar um exemplo. Em seguida, será falado sobre o tempo de execução do algoritmo em notação Big-O. Por fim, serão explorados outros casos de uso (problemas) onde há aplicabilidade daquele algoritmo

1.1.1 O que aprenderemos sobre desempenho

Aprenderemos, principalmente, a comparar o desempenho de diferentes algoritmos. Exemplo: "Para este caso, devemos usar quicksort ou mergesort. Devemos usar uma lista encadeada ou um array?"

1.1.2 O que aprenderemos sobre a solução de problemas

1.2 Pesquisa binária

Antes de tudo, vamos a um exemplo: Suponhamos que tenhamos uma lista telefonica e queremos procurar um contato com a letra K, poderíamos começar foleando a lista até encontrar ou podemos começar do meio (ja que sabemos que

não estará no começo) e partir dali.

Outro exemplo interessante é o Facebook. Suponhamos que o Facebook quer verificar se determinada conta existe. O nome da conta é Nicolas. Ele irá procurar no banco de dados. Poderia até começar do A, mas faz mais sentido começar a busca pelo meio.

Todos os exemplos acima representam a aplicabilidade da busca binária. A busca binária é um algoritmo, onde passamos para ele uma lista de elementos (que deverá estar ordenada) e se o elemento buscado está na lista, será retornada a posição do elemento e caso contrário, será retornado None

Um exemplo interessante para visualizarmos é: suponhamos que tivéssemos uma lista onde temos números de 1 a 100 e eu peço a alguem que tente acertar o numero que estou pensando (nesse caso 99), onde digo, a cada tentativa, se o número chutado está muito baixo ou muito alto. Uma maneira que a pessoa possa pensar para encontrar o número que estou pensando é chutar número a número começando do 1. Esse método se chama pesquisa simples (ou pesquisa estúpida como disse o autor). Essa é uma maneira pouco eficiente, sendo 99 o número que escolhi, a pessoa precisaria chutar 99 números para acertar. A cada chute, ela estaria eliminando apenas um número por vez dessa maneira.

1.2.1 Uma maneira melhor de buscar (a pesquisa binária)

Dessa forma, uma maneira mais eficiente de acertar o número que estou pensando seria começar chutando do 50, assim, eu diria "muito baixo". Com isso, a pessoa eliminaria METADE dos números, pois ela saberia que os números de 1 a 50 são muito baixos. Aí depois, suponhamos que ela chute 75 e (como meu número é 99) eu fale "muito baixo". Ainda sim, ela eliminou uma quantidade considerável de valores. Isso continuaria, até que enfim meu número fosse encontrado. Essa maneira de pesquisar se chama pesquisa binária.

Você percebe portanto que na pesquisa binária, a busca ocorre através dos valores intermediários,o que permite eliminar uma grande quantidade de valores a cada etapa

Suponha agora que você esteja procurando uma palavra em um dicionário com 240.000 palavras. Na pior das hipóteses (a última palavra desse dicionário), em cada uma das formas de busca:

- Pesquisa simples: 240.000 etapas
- Pesquisa binária: 18 etapas, uma vez que a cada etapa eliminamos o número de palavras pela metade, até que sobre apenas uma palavra.

Portanto, percebe-se uma grande diferença! Podemos dizer que a pesquisa binária, de maneira geral, para uma lista com n elementos, ela precisa de $\log_2 n$

para retornar o valor correto, enquanto isso, a pesquisa simples precisaria de n etapas. Exemplo: em uma lista com 8 elementos, na pesquisa simples, precisariamos checar, em seu máximo, 8 elementos. Enquanto isso, na pesquisa binária, precisariamos, em seu máximo, checar $\log_2 8$, que seriam 3 elementos

OBS: na notação Big O sempre quando falamos de log, na verdade estamos nos referindo a $\log_2$

Um detalhe que já chegamos a comentar é que para que a pesquisa binária ocorra, a nossa lista precisa estar ordenada.

Implementação da pesquisa binária

Agora, para conseguir visualizar melhor a pesquisa binária, realizaremos sua implementação. Faremos a implementação em um array. Teremos a função pesquisa_binaria, que receberá um array ordenado e um valor. Se o valor estiver no array, será retornada sua posição. Caso contrário, será retornado None.

No começo da nossa função pesquisa_binaria, teremos:

```
baixo = 0
alto = len(lista) - 1
```

Acima, acontece o mapeamento das posições do array, para assim conseguirmos pegar a posição intermediária. Após isso, teremos:

```
meio = (baixo + alto) // 2
chute = lista[meio]
```

O que ele faz acima é encontrar o meio do nosso array e depois pegar o valor que temos no meio dele. Perceba que utilizamos o operador //, ou seja, estamos fazendo uma divisão inteira. Isso ocorre para que o valor "meio"seja arredondado para baixo caso (baixo + alto) não seja um valor par. A partir disso, temos:

```
if chute < item:
 baixo = meio + 1
```

Se o valor do primeiro chute for MENOR que o item que estamos procurando, então o valor "baixo"é atualizado para meio + 1. Isso acontece porque ele passa a desconsiderar todos os valores do antigo meio para baixo. O mesmo ocorre para caso o valor do primeiro chute for MAIOR que o item que estamos procurando:

```
if chute > item:
 alto = meio - 1
```

O que acontece acima é que se o valor do chute for maior que o item que estamos procurando, o valor "alto"é atualizado, pois ele passa a desconsiderar todos os valores do antigo meio para cima

Agora veja o código completo abaixo:

```
def pesquisa_binaria(lista, item):
baixo = 0
alto = len(lista) - 1

while baixo <= alto:
 meio = (baixo + alto)//2
 chute = lista[meio]

 if chute == item:
  return meio

 if chute > item:
  alto = meio - 1

 else:
  baixo = meio + 1

return None
```

Veja abaixo visualmente o algoritmos para ter um melhor entendimento. Usaremos como exemplo a lista: [1, 3, 5, 7, 9], onde tentaremos buscar o valor 3.

1.2.2 Tempo de execução

1.3 Notação Big O

1.3.1 Tempo de execução dos algoritmos cresce a taxas diferentes

1.3.2 Vendo diferentes tempos de execução Big O

1.3.3 A notação Big O estabelece o tempo de execução para a pior hipótese

1.3.4 Alguns exemplos comuns de execução Big O

1.3.5 O caixeiro-viajante

1.4 Resumo do capítulo

Capítulo 2

Ordenação por seleção