Introdução à Análise de Algoritmos

Márcio Moretto Ribeiro

9 de setembro de 2021

Conteúdo

Apresentação

Esta apostila contém notas de aula do curso ministrado por mim no segundo semestre de 2021 para as duas turmas noturnas de Sistemas de Informação. Naquele ano o curso de Introdução à Análise de Algoritmos foi ministrado à distância por conta da pandemia de COVID19. Seu conteúdo está baseado nos livros que servem de bibliografia para o curso:

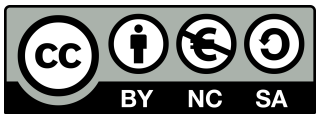
[1] T. H. Cormen. *Algoritmos: teoria e prática*. Campus, 2012. ISBN: 9788535236996.

[4] R. Sedgewick. *Algorithms in C*. Algorithms in C. Addison Wesley Professional, 2001. ISBN: 9780201756081.

[5] R. Sedgewick e K. Wayne. *Algorithms: Algorithms 4*. Pearson Education, 2011. ISBN: 9780132762564.

Além dos livros, serviu de material para elaboração destas notas outros materiais citados ao longo do texto bem como as gravações dos cursos do professor Robert Sedgeweick para o Coursera e do professor Ronald Rivest para o MIT.

A última versão desta apostila está disponível em um repositório no github (https://github.com/marciomr/apostila-iaa.git). Agradeço de antemão eventuais sugestões de correção que forem submetidas pela plataforma.



Capítulo 1

Introdução

1.1 Algoritmos

Um *problema computacional* é a especificação de uma relação desejada entre um certo *valor de entrada* escolhido em um conjunto de valores válidos e o *valor de saída* esperado.

Exemplo 1.1.1:

Problema da busca

Entrada: Uma sequência de n valores $a_1, \ldots, a_n$ em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou $\perp$ caso contrário.

A sequência $3, 5, 16, 17, -1$ junto do valor 5 é uma entrada válida para este problema. Qualquer entrada válida é chamada de *instância do problema*. A saída esperada para essa instância é 2, pois o valor 5 ocorre na segunda posição da sequência.

Outra instância do problema é dada pela mesma sequência junto do valor 42. Neste caso a saída esperada é $\perp$, uma vez que o valor 42 não ocorre na sequência.

Exemplo 1.1.2:

Problema da 3-soma

Entrada: Três sequência de $n \in \mathbb{N}$ valores cada $a_1, \ldots, a_n$, $b_1, \ldots, b_n$ e $c_1, \ldots, c_n$ em que $a_i, b_i, c_i \in \mathbb{Z}$ para $1 \leq i \leq n$.

Saída: A quantidade de is, js e ks tais que $a_i + b_j + c_j = 0$.

Uma instância do problema é da pelas sequências:

$1, 2, 3$

$2, 4, 6$

$-4, -8, -10$

A saída esperada neste caso é 2 porque:

$2 + 2 - 4 = 0$

$2 + 6 - 8 = 0$

Exemplo 1.1.3:

Problema da ordenação

Entrada: Uma sequência de n valores $a_1, \ldots, a_n$ em que $a_i \in \mathbb{Z}$ para $1 \leq i \leq n$.

Saída: Uma permutação da sequência de entrada $a'_1, \ldots, a'_n$ tal que $a_i \leq a_j$ para todo $i \leq j$.

Para a instância $3, 42, 17, 2, -1$ deste problema, a saída esperada é $-1, 2, 3, 17, 42$.

A disciplina de Introdução à Teoria da Computação (ITC) tem como objeto de estudo os problemas computacionais. Como eles se classificam entre os que tem solução ou não e entre os que tem solução eficiente ou não. A solução de um problema computacional é um algoritmo.

Os objetos de estudo desta disciplina são os algoritmos. Mas afinal, o que são algoritmos?

Um *algoritmo* parte de uma entrada escolhida em um conjunto potencialmente infinito de possibilidades (*princípio da massividade*) para produzir um valor de saída. O algoritmo processa a entrada por meio de uma sequência de passos (*princípio da discretude*) que produzem valores intermediários. Cada passo segue uma instrução simples (*princípio da elementaridade*) que só depende dos valores anteriores, não admitindo ambiguidades (*princípio da exatidão*) [3].

Um *prorgrama* é a realização de um algoritmo em certa *linguagem de programação*. Assim, um algoritmo é, de um lado, a solução de um problema de computação e, de outro, uma abstração de um conjunto de programas, ele é a idéia por trás desses programas.

Um algoritmo é *correto* se para toda instância do problema ele produz a saída esperada depois de uma sequência finita de passos. Nesse caso dizemos que o algoritmo *resolve* o problema.

Há uma controversa se devemos ou não considerar uma sequência infinita de instruções como um algoritmo. Essa questão, complicada, está no coração do nascimento da ciência da computação e será tratada em ITC. Neste curso focaremos nos algoritmos corretos e, assim, escaparemos dela.

Para enfatizar o fato de que algoritmos abstrações de programas, eles serão apresentados neste curso em uma linguagem informal conhecida como *pseudo-código*.

Exemplo 1.1.4:

Considere a seguinte solução para o problema da busca.

```
BuscaSequencial(A, b)
1  ▷ Recebe a_1, ..., a_n com a_i ∈ ℤ e b ∈ ℤ
2  ▷ Devolve i tal que a_i = b se existir e ⊥ caso contrário
3  for i ← 1 até n
4        do if a_i = b
5              then return i
6  return ⊥
```

As duas primeiras linhas são apenas comentários que explicitam a especificação do problema que o algoritmo resolve. A linha 3 indica que um certo valor i deve variar de 1 até n. As duas linhas seguintes estabelecem que se a_i for igual a b então o valor de i

deve ser devolvido como resposta do problema. Por fim, a última linha indica que se o algoritmo chegou naquele ponto, então o valor $\perp$ deve ser devolvido como solução do problema.

Esta disciplina estuda algoritmos. Como podemos garantir que certo algoritmo resolve um problema, ou seja, que ele é correto? O algoritmo do exemplo acima está correto? Por que? Como podemos comparar duas soluções distintas para um mesmo problema? Ou seja, se conhecemos dois um mais algoritmos corretos para um mesmo problema, como avaliamos qual é melhor? O algoritmo do exemplo acima é o melhor algoritmo possível para o problema da busca? Como podemos garantir isso?

Avaliaremos os algoritmos corretos a partir da quantidade de recursos que eles consomem. Estudaremos particularmente dois recursos: espaço de memória e, principalmente, o tempo de execução.

Nos capítulos seguintes veremos uma série de algoritmos para resolver alguns problemas centrais da computação como o problema da busca e da ordenação. Em cada caso avaliaremos os algoritmos apresentados quanto sua corretude e sua eficiência em consumo de tempo e espaço de memória.

No Capítulo X apresentaremos o estudo dos algoritmos a partir do método empírico. Relembraremos o método e veremos um exemplo comparando o tempo de execução de duas soluções para o problema da busca em sequências ordenadas. Então exploraremos técnicas para arrsicar modelos matemáticos adequados para avaliar o consumo de tempo dos algoritmos. E finalmente veremos ferramentas matemáticas uteis para comparar funções quanto ao seu crescimento, a chamada notação assintótica. No Capitulo X estudaremos algoritmos de ordenação como estudo de caso da teoria apresentada anteriormente. Veremos uma série de algoritmos que resolvem o mesmo problema e utilizaremos as técnicas apresentadas para construir e testar modelos do consumo de tempo deles. Estudaremos também um limite teórico da eficiência do problema da ordenação e veremos dois algoritmos que superam esse limite utilizando mais informações do que as assumidas no enunciado do teorema. Concluiremos a apostila no Capítulo X apresentando algoritmos e técnicas um pouco mais avançãdos como programação dinâmica e análise amortizada.

Capítulo 2

Método Empírico

Esquematicamente, o método empírico pode ser descrito por cinco fases:

1. *Observação:* medições sobre algum aspecto do mundo
2. *Hipótese:* concepção de um modelo consistente com as observações
3. *Predição:* eventos são previstos de acordo com o modelo
4. *Verificação:* as predições são testadas fazendo-se novas observações
5. *Validação:* o processo se repete ajustando o modelo até que ele concorde com as observações

Dois pontos centrais sobre o método empírico é que as verificações devem ser *reprodutíveis* e as hipóteses *falseáveis*. Uma hipóstese que não pode ser falseada por observações (empíricamente) não é científica e a o processo de verificação deve poder ser feito por outros cientístas independentes.

O aspecto do mundo que pretendemos investigar nesta disciplina é o tmepo de processamento de um algoritmo. Lembre-se, porém, que um algoritmo é uma ideia que precede o advento dos computadores. O algoritmo de Euclídes, por exemplo, data de 300 a.C., ou seja, séculos antes dos primeiros computadores começarem a ser construídos nos anos 40. Embora o algoritmo seja um conceito matemático, uma série de pesquisadores tiveram a ideia de investigá-los de maneira empírica no final dos anos 60. A série de livros The Art of Programing, de Donald Knuth, é um marco dessa abordagem dos estudos de algoritmos [2].

Em nosso recorte, observaremos o tempo de processamento da execução de um programa para diferentes entradas. Considere, por exemplo, o algoritmo para o problema da Busca apresentado no capítulo anterior:

BuscaSequencial(A, b)
1 **for** $i \leftarrow 1$ até n
2 **do if** $a_i = b$
3 **then return** i
4 **return** $\perp$

Vamos implementar esse algoritmo na linguagem C de maneira direita:

```
// devolve a posicao de n no arranjo ou -1 se nao encontrar
int buscasequencial(int* array, int n, int size){
  int i;
  for(i = 0; i < size; i++)
    if(array[i] == n)
      return i;
  return -1;
}
```

Realizamos observações usando uma máquina específica (um notebook Dell com processador intel core i7 de 8^a geração de 1,9GHz) em um sistema operacional específico (Linux 5.11). Medimos o tempo total de 300 buscas mal sucedidas em arranjos de tamanhos diferentes com valores inteiros positivos calculados aleatoriamente[1]. Os programas que calcula o tempo da busca e que gera as entradas estão disponíveis em `https://github.com/marciomr/IAA`. Variamos o tamanho do arranjo entre um milhão e dez milhões. Obtivemos o seguinte resultado para dez observações:

As observações sugerem que para cada um milhão de valores no arranjo, o tempo de processamento aumenta mais ou menos um segundo. Essa poderia ser nossa hipótese, mas podemos fazer algo um pouco mais sofisticado. Vamos plotar os valores da tabela em um gráfico (Figura 2).

Como os pontos estão mais ou menos alinhados e como é razoável supor que um arranjo sem nenhum elemento retornaria instantaneamente o resultado, faremos a hipótese de que o tempo de processamento segue uma

[1]Mais precisamente os valores seguem uma sequência pseudo-aleatória partindo de uma semente incial.

tamanho do arranjo em milhões	tempo de 300 buscas em segundos
1	0,99
2	2,08
3	3,12
4	3,99
5	5,05
6	5,94
7	7,03
8	7,92
9	8,93
10	9,85

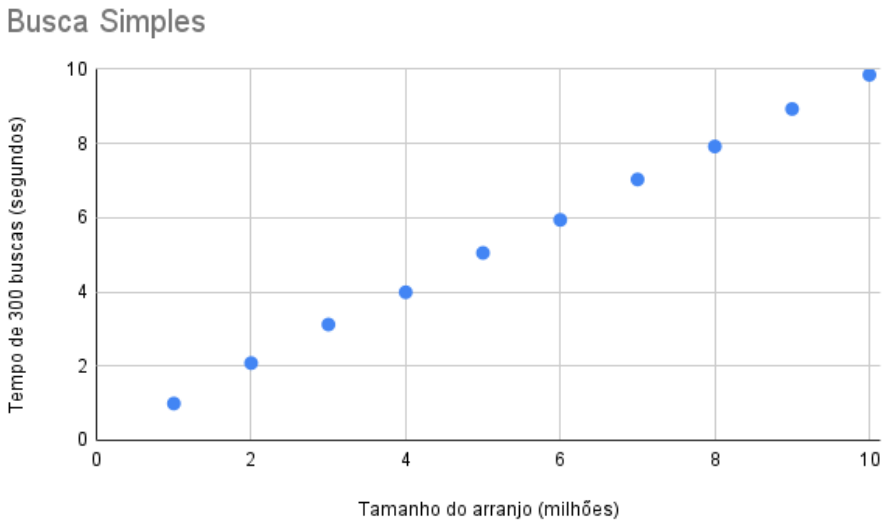


Figura 2.1: Tempo de processamento da busca sequencial.

função linear partindo do zero. Ou seja, a seguinte função descreve o tempo de processamento da nossa implementação em nosso ambiente:

$$t(x) = a.x$$

Podemos então usar uma regressão linear para estimar o valor de a que minimize a distância desse reta para cada um dos pontos. Obtemos então o valor $a = 0,997$. Na Figura 2 plotamos a função $t(x) = 0,997x$ no gráfico anterior.

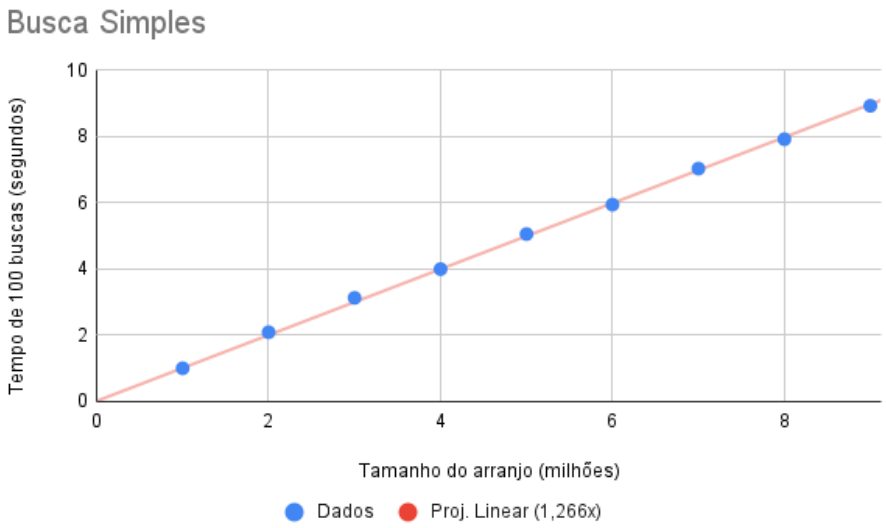


Figura 2.2: Gráfico ilustrando a hipótese de que o tempo de processamento da busca sequencial segue a função linear $t(x) = 0,997x$.

Podemos agora testar nossa hipótese de que o tempo de processamento da nossa implementação da busca sequencial para outros valores ainda não observados. A segunda coluna da Tabela 2 mostra os valores previstos pela hipótese para o tempo de processamento para entradas de tamanho 11 a 15 milhões. Finalmente, podemos testar a hipótese. Na última coluna da mesma tabela indicamos os valores observados para essas entradas:

Os valores observados são notavelmente próximos aos previstos. Ou seja, nossa hipótese foi verificada. Poderíamos neste momento utilizar um teste estatístico para verificar nossa hipótese, mas isso foge do escopo desta disciplina. Por hora diremos apenas que os dados parecem verificar a hipótese.

tamanho do arranjo em milhões	tempo previsto	tempo observado
11	10,97	10,87
12	11,97	11,81
13	12,97	12,78
14	13,96	14,00
15	14,96	14,74

Tabela 2.1: Tempo de processamento previsto e observado para tamanhos maiores de arranjos.

Consideremos agora a seguinte versão modificada do Problema da busca:

Problema da busca em uma sequência ordenada

Entrada: Uma sequência de n valores $a_1, \ldots, a_n$ em ordem crescente, isto é, $a_i \leq a_j$ para todo $i \leq j$ e $b \in \mathbb{Z}$.

Saída: $i \in \mathbb{N}$ tal que $a_i = b$ se existir ou $\perp$ caso contrário.

Note que este problema é uma versão restrita do problema anterior. Assim, toda solução do Problema da busca é também uma solução para o Problema da busca em uma sequência ordenada – o inverso não é necessariamente verdadeiro. Em particular, o algoritmo de Busca Sequêncial resolve ambos os problemas. O seguinte algoritmo, por sua vez, resolve apenas o segundo:

BuscaBinaria(A, b)

```
1  ▷ Recebe uma sequência ordenada a_1, ... a_n e um valor b todos inteiros
2  ▷ Devolve i tal que a_i = b ou ⊥ caso b não ocorra na sequência
3  i ← 1
4  j ← |A|
5  while i ≤ j
6        do m ← ⌊(j+i)/2⌋
7           if b < a_m
8              then j ← m − 1
9              else  if b > a_m
10                       then i ← m + 1
11                       else  return m
12 return ⊥
```

tamanho do arranjo em milhões	tempo de 300 buscas em segundos
1	0,00
2	0,00
3	0,00
4	0,00
5	0,00
6	0,00
7	0,00
8	0,00
9	0,00
10	0,00

Este algoritmo é um pouco mais sofisticado do que o anterior. Começamos avaliando o elemento no centro da sequência (a_m). Como a sequência está em ordem crescent, se o valor procurado (b) for maior do que a_m então ele deve estar depois do valor central e podemos ignorar todos os valores anteriores a m. Analogamente, se b for menor que a_m ele deve estar antes de m e podemos ignorar todos os valores posteriores.

Como fizemos no exemplo anterior, vamos avaliar o tempo de processamento deste algoritmo utilizando o método empírico. O primeiro passo é fazer algumas observações. Vamos repetir as observações feitas no exemplo anterior.

As observações sugerem que o novo algoritmo é muito mais eficiente do que o primeiro. Porém, elas não nos ajudam a conceber um modelo.

Façamos então observações com mais repetições. Depois de alguns testes aprendemos que repetindo dez milhões de buscas o tempo de processamento passa a ser mensurável.

Desta vez conseguimos fazer as medições. As observações sugerem que o tempo de processamento é independente do tamanho do arranjo. Para arranjos de qualquer tamanho o tempo de procesamento parece ser maior ou menos o mesmo. Podemos então levantar a hipótese de que o tempo de processamento é constante:

$$t(x) = a$$

Para calcular o valor de a peguemos a média das observações $a = 0,63$.

Assim, nosso modelo preveria que para entradas de qualquer tamanho, o tempo de processamento seria próximo a $0,63$ segundos. Vamos testar essa

tamanho do arranjo em milhões	tempo de 10M buscas em segundos
1	0,66
2	0,64
3	0,60
4	0,59
5	0,62
6	0,63
7	0,63
8	0,63
9	0,66
10	0,65

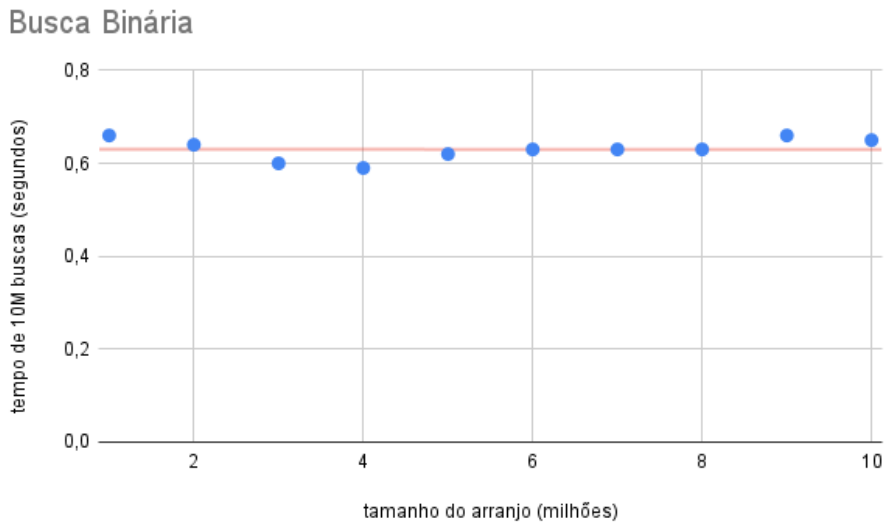


Figura 2.3: Gráfico ilustrando a hipótese de que o tempo de processamento da busca binária segue a função constante $t(x) = 0,63$.

tamanho do arranjo em milhões	tempo previsto	tempo observado
20	0,63	0,68
100	0,63	0,75
500	0,63	0,81
1000	0,63	0,89

Tabela 2.2: Tempo de processamento previsto e observado para tamanhos maiores de arranjos.

tamanho do arranjo em milhões	tempo de 10M buscas em segundos
1	0,59
2	0,62
4	0,69
8	0,69
16	0,74
32	0,76
64	0,79
128	0,83
256	0,96
512	1,01

hipótese com entradas de tamanhos bem maior para ver se a hipótese se verifica.

O processamento continua muito rápido mesmo para arranjos bem grandes, mas parece que a hipótese não foi verificada. Seguindo o método empírico, devemos fazer novas observações para tentar formular uma nova hipótese. Tentemos repetir nossas observações com uma maior amplitude de valores. Ao invés de aumentar o tamanho de nossa sequência em um tamanho fixo a cada observação, vamos tentar desta vez dobrar o tamanho da sequência a cada observação.

As observações sugerem que o tempo de processamento cresce linearmente conforme dobramos o tamanho da nossa entrada. Podemos arriscar então que o tempo de processamento segue o seguinte modelo:

$$t(2^y) = a.y + b$$

Como fizemos no exemplo anterior, podemos computar os valores de a e b utilizando uma técnica de regressão linear. Ficamos assim com $a = 0,043$

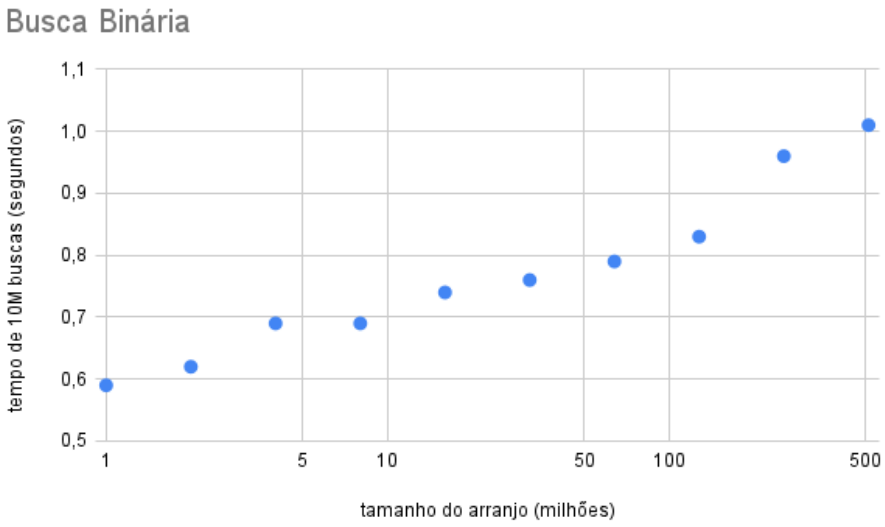


Figura 2.4: Gráfico ilustrando a hipótese de que o tempo de processamento da busca binária segue a função constante $t(2^y) = a.y + b$.

e $b = 0,529$.

Por fim, mudamos a variável $2^y = x$ e obtemos a seguinte equação:

$$t(x) = a.log_2(x) + b = 0,043.log_2(x) + 0,529$$

Capítulo 3

Correção

No capítulo anterior vimos que é possível utilizar o método empírico para avaliar tempo de processamento de diferentes soluções para um mesmo problema. Neste capítulo daremos um passo atrás. Como podemos garantir, para começo de conversa, que um algoritmo de fato resolve um problema? Em outras palavras, como podemos provar a correção de um algoritmo?

Para provar a correção de um algoritmo iremos usar uma forma de prova por indução. Assim, antes de partir para os exemplos de prova por indução em um algoritmo, vale a pena relembrar como funciona uma prova por indução em um contexto mais típico.

Normalmente, uma prova por indução é usada para provar alguma propriedade sobre números naturais. A ideia de uma prova por indução é relativamente simples. Ela se divide em três etapas. Primeiro precisamos provar que a propriedade vale para o 0 ou para o primeiro número que nos interessa. Isso é chamado de *Base da Indução*. Então supomos que a propriedade vale para um número n, *Hipótese da Indução*. Por fim, provamos que se vale para n então vale para $n+1$, *Passo de Indução*. Assim, mostramos que vale para 0 e se vale para 0, deve valer para 1, e se vale para 1, deve valer para 2 e assim por diante. Com isso, provamos que a propriedade vale para todos os números naturais.

Vejamos um exemplo. Considere a seguinte somatória:

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Vamos provar por indução que esse resultado vale para qualquer número

natural n.

O primeiro passo é provar a base da indução. Vamos provar que o resultado vale para $n = 1$

$$1 = \frac{1(1+1)}{2}$$

Agora vamos explicitar a Hipótese de Induação.

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Por fim, fazemos o Passo de Indução.

$$\begin{aligned} 1 + 2 + 3 + \cdots + n + n + 1 &= \frac{n(n+1)}{2} + n + 1 \\ &= \frac{n^2 + n + 2n + 2}{2} \\ &= \frac{n^2 + 3n + 2}{2} \\ &= \frac{(n+1)(n+2)}{2} \end{aligned}$$

Note que a primeira equação vale por conta da Hipótese de Indução.

Passemos agora para um problema computacional. Continuemos considerando o problema da busca em uma sequência ordenada e os dois algoritmos que conhecemos para ele. Primeiro o algoritmo da busca sequencial:

```
BuscaSequencial(A, b)
1  for i ← 1 até n
2        do if a_i = b
3              then return i
4  return ⊥
```

Para mostrar que o algoritmo é correto temos que incontrar um *invariante*. Uma propriedade que vale em todas as iterações. No nosso caso, vamos considerar a linha 2 do algoritmo e a propriedade será a seguinte:

$$b \text{ não ocorre em } a_1, \ldots, a_{i-1}$$

Vamos provar que essa propriedade é de fato invariante usando a técnica da indução.

Primeiro temos que provar que ela vale para $i = 1$. Essa é a base da indução.

Para isso basta notar que neste caso a sequência é vazia e, portanto, b não pertence a ela.

A Hipótese de Indução já foi explicitada. Para mostrar o passo de indução vamos supor a HI e mostrar que a propriedade continua valendo para $i + 1$. Se b estivesse na sequência $a_1 \dots a_i$ então, pela HI, $b = a_i$. Neste caso, não chegaríamos na linha 2, porque o algoritmo teria encerrado antes disso.

Assim, sempre que chegamos na linha 2, b não ocorre em $a_1, dots, a_{i-1}$. Portanto, quando chegamos na 3, é a primeira vez em que $a_i = b$. E se chegarmos na linha 4, sabemos que b não ocorre em $a_1, \dots, a_n$.

Vamos agora para nosso segundo exemplo:

BuscaBinaria(A, b)

```
1  i ← 1
2  j ← |A|
3  while i ≤ j
4        do m ← ⌊(j+i)/2⌋
5           if b < a_m
6              then j ← m − 1
7              else  if b > a_m
8                       then i ← m + 1
9                       else  return m
10 return ⊥
```

Vamos mostrar que as seguintes propriedades são invariantes na linha 4:

$$b \text{ não ocorre em } a_1, \dots, a_{i-1}$$

$$b \text{ não ocorre em } a_{j+1}, \dots, a_n$$

A base da indução é simples, no primeiro momento $i = 1$ e $j = n$. Portanto ambas propriedades valem porque as duas sequências são vazias nessas condições.

Vamos supor que a propriedade vale em um certo momento quando chegamos na linha 4. Agora imagine que chegamos mais uma vez nessa linha.

Neste caso, não saímos do laço. Portanto, uma de duas coisas teve que ocorrer: $b < a_m$ ou $b > a_m$.

No primeiro caso, temos que $j = m-1$. Como a sequência está ordenada, b não ocorre em $a_{j+1} = a_m, \ldots, a_n$ porque $b < a_m$. Além disso, pela hipótese de indução, temos que b não ocorre em $a_1, \ldots, a_{i-1}$.

No segundo caso, temos que $i = m+1$. Como a sequência está ordenada, b não ocorre em $a_1 = a_1, \ldots, a_n$ porque $b > a_m$. Além disso, pela hipótese de indução, temos que b não ocorre em $a_{j+1}, \ldots, a_n$.

O invariante vale sempre na linha 4. Se chegarmos na linha 9 é proque $a_m = b$ e se chegarmos na linha 10 é porque b não está nem em $a_1, \ldots, a_i$ nem em $a_j, \ldots, a_n$ e $i > j$. Portanto b não está em $a_1, \ldots, a_n$.

Exercício 1: Considere o seguinte algoritmo:

```
3Soma(A, B, C)
1  n ← 0
2  for i ← 1 até n
3        do for j ← 1 até n
4                 do for k ← 1 até n
5                    if a_i + b_j + c_k = 0
6                       then n ← n + 1
7  return n
```

Prove que este algoritmo resolve o problema da 3-soma apresentado no Capítulo 1.

Bibliografia

[1] T. H. Cormen. *Algoritmos: teoria e prática*. Campus, 2012. ISBN: 9788535236996.

[2] D.E. Knuth. *The Art of Computer Programming: Volume 1: Fundamental Algorithms*. Pearson Education, 1997. ISBN: 9780321635747.

[3] A.I. Mal'cev. *Algorithms and Recursive Functions*. Wolters-Noorhoff, 1970.

[4] R. Sedgewick. *Algorithms in C*. Algorithms in C. Addison Wesley Professional, 2001. ISBN: 9780201756081.

[5] R. Sedgewick e K. Wayne. *Algorithms: Algorithms 4*. Pearson Education, 2011. ISBN: 9780132762564.