Introdução à Análise de Algoritmos

Márcio Moretto Ribeiro

9 de outubro 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:

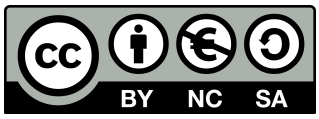
[1] T. H. Cormen. *Algoritmos: teoria e prática*. Campus, 2012. ISBN: 9788535236996.

[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

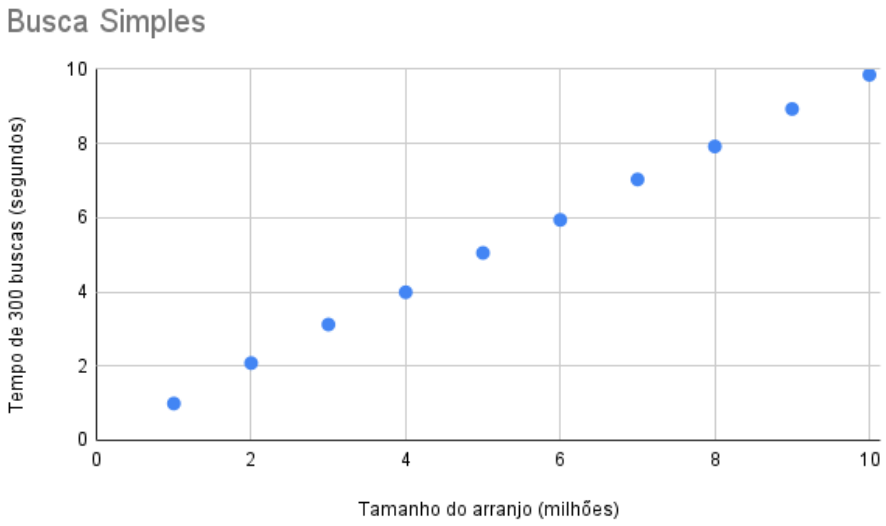


Figura 2.1: Tempo de processamento da busca sequencial.

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.

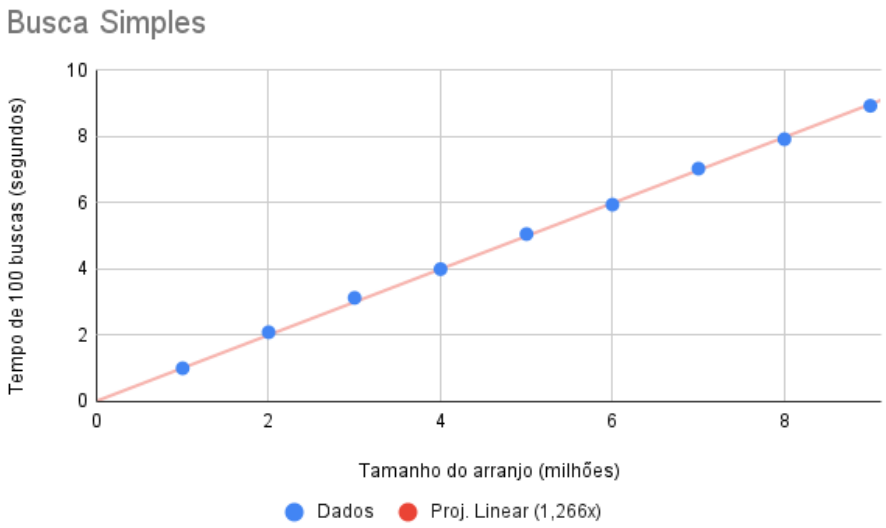


Figura 2.2: Gráfico ilustrando a hipótese de que o tempo de processamento da busca sequencial segue a função linear $t(x) = 0,997x$.

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

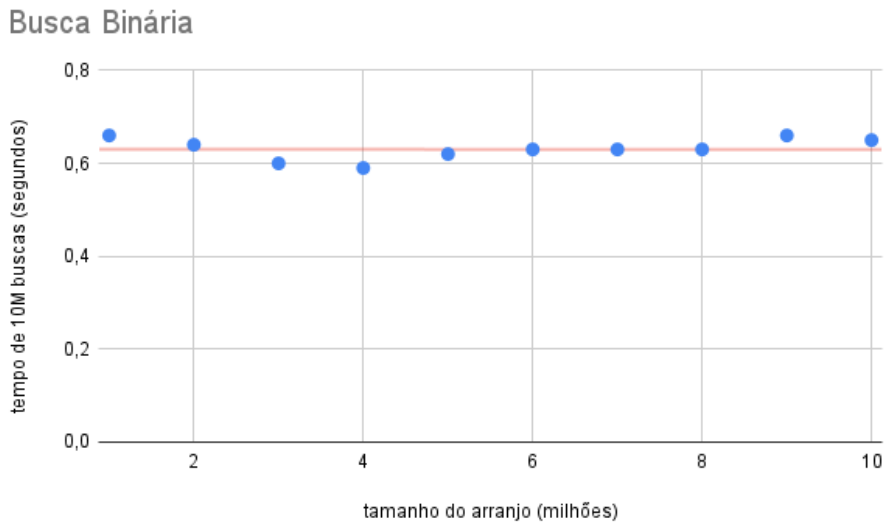


Figura 2.3: Gráfico ilustrando a hipótese de que o tempo de processamento da busca binária segue a função constante $t(x) = 0,63$.

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 $a = 0,043$

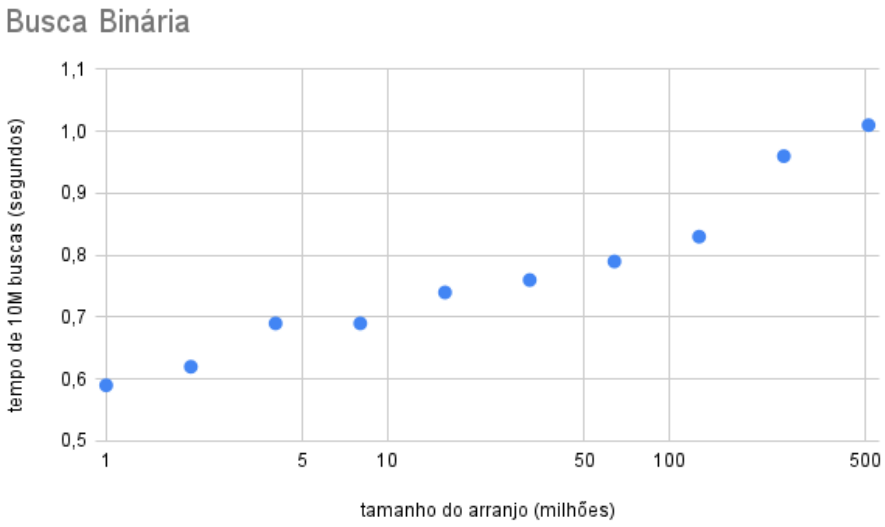


Figura 2.4: Gráfico ilustrando a hipótese de que o tempo de processamento da busca binária segue a função constante $t(2^y) = a.y + b$.

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 encontrar 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:

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

Para provar que essa propriedade é de fato invariante, usaremos um técnica similar às provas de indução. Primeiro precisamos mostrar que a propriedade vale na primeira vez que entramos no laço. Essa etapa é chamada de *inicialização* e corresponde à base da indução.

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

Em seguida precisamos fazer o passo de indução supondo a hipótese da indução. Para tanto, faremos o seguinte. Iremos supor que a propriedade valia na iteração anterior do laço. Em seguinda provaremos que a propriedade continua valendo depois de uma nova iteração. Essa etapa é chamada de *manutenção*.

Pela Hipótese da Indução, ee b estivesse na sequência $a_1, \ldots, a_i$ então, necessariamente temos que $b = a_i$. Neste caso, não chegaríamos na linha 2, porque o algoritmo teria encerrado na linha 3.

Assim, sempre que chegamos na linha 2, b não ocorre em $a_1, \ldots, a_{i-1}$. Ou seja, a propriedade de fato é um invariante.

Para concluir a prova vamos usar a propriedade invariante para provar que a saída esperada é aquela produzida pelo algoritmo. Para isso, precisamos analisar o que ocorre quando saímos do laço. Essa etapa é chamara de *término*.

No nosso exemplo, o invariante nos mostra que quando chegamos na linha 3, é a primeira vez em que $a_i = b$. E que se chegarmos na linha 4, sabemos que b não ocorre em $a_1, \ldots, a_n$.

Resumindo o que vimos até aqui. Para demonstrar a correção de um algoritmo, o primeiro passo é encontrar uma propriedade *invariante*. Seguimos então os seguintes passos:

1. *Inicialização*: provamos que a propriedade vale na primeira iteração (isso equivale a prova da base da indução).

2. *Manutenção*: provamos que se a propriedade valia na iteração anterior (hipótese da indução), ela continua valendo na iteração atual (isso equivale ao passo de indução).

3. *Término*: usamos o invariante para avaliar o que acontece quando saímos do laço.

Os dois primeiros passos são inspirados na prova por indução e servem para provar que a propriedade escolhida é, de fato, invariante. O terceiro passo completa a prova da correção.

Vamos agora para um exemplo um pouco mais complexo:

```
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:

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

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

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

A etapa de manutenção segue da seguinte forma. 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 $a_{j+1}, \ldots, a_n$.

Para completar a prova, falta o término. 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  m ← 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 m ← m + 1
7  return m
```

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

Capítulo 4

Modelos

Vimos no Capítulo **??** que podemos avaliar o tempo de processamento de um algoritmo usando o método empírico. Para isso, em algum momento precisamos de um *modelo* que iremos testar. Nos exemplos que vimos o modelo foi tirado intuitivamente a partir das observações. Há métodos mais adequados para conceber um modelo para o consumo de tempo de um algoritmo.

Vamos mais uma vez considerar os algortimos de busca que temos estudado.

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

Quando esse algoritmo for implementado e executado em uma máquina, as instruções seguidas tomarão um tempo que depende de uma série de fatores. Vamos seguir a suposição razoável que instruções iguais tomarão o mesmo tempo em uma mesmo contexto. Assim, podemos supor, por exemplo, que:

- incrementar uma variável (linha 1) toma tempo c_1
- comparar duas variáveis (lina 2) toma tempo c_2
- devolver um valor (linhas 3 e 4) toma tempo c_3

Os valores de c_1, c_2 e c_3 deve ser diferente a depender do contexto, mas vamos supor que sejam *constantes* em um mesmo contexto. Fazendo essa suposição, vamos então contar quantas vezes cada uma dessas operações é feita.

O número de repetições depende de duas coisas: o tamanho da entrada (n) e os valores da entrada. Gostaríamos de um modelo que dependesse apenas do tamanho da entrada. Podemos fazer dois tipos de análises para abstrair os valores da entrada:

- *análise de caso médio*: nos dá um valor mais preciso, mas depende da distribuição de probabilidade da entrada ou
- *análise de pior caso*: nos dá um valor menos preciso, mas ainda assim bastante útil e não depende de nenhuma suposição sobre os valores da entrada.

Nos focaremos em análises de pior caso porque na maioria das vezes não sabemos de antemão a distribuição de probabilidade da entrada.

Vamos contar o número de repetições de cada operação assumindo o pior caso:

- a atualização da variável i na linha 1 se repete n vezes,
- a comparação na linha 2 se repete n vezes no pior caso e
- as linhas 3 e 4 ocorrem uma única vez ao todo durante toda a execução.

Podemos então construir o seguinte modelo para o tempo de processamento do algoritmo no pior caso em função do tamanho n da entrada:

$$t(n) = c_1 n + c_2 n + c_3 = (c_1 + c_2)n + c_3$$

O mesmo modelo que extraímos das observações no Capítulo **??**.

Vamos agora replicar esse mesmo tipo de análise para nosso outro algoritmo.

```
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 ⊥
```

O primeiro passo é atribuir constantes para o tempo de processamento de cada operação atômica:

- c_1 para as atribuições nas linhas 1 e 2,
- c_2 para as comparações nas linhas 3, 5 e 7,
- c_3 para a divisão na linha 4,
- c_4 para o incremento e decremento nas linhas 6 e 8 e
- c_5 para devolver um valor nas linhas 9 e 10.

Agora vamos contar o número de vezes que cada linha é executada no pior caso. As linhas 1 e 2 são executadas uma vez cada. As linhas 9 e 10 são executadas uma vez ao todo. Já as linhas 3 a 6 são executadas enquanto $i \leq j$.

Para avaliar quantas vezes as linhas 3-6 são executadas, vamos primeiro supor que o tamanho da sequência é uma potência de 2 e, portanto, existem x tal que:

$$n = 2^x \Rightarrow x = log_2(n)$$

O pedaço da sequência que nos interessa é $a_i, \ldots, a_j$ – na prova da correção vimos b nunca está no outro pedaço da sequência. Note agora que a cada passo o tamanho dessa sequência diminui pela metade. Ele começa sendo $n = 2^x$, cai para $\frac{n}{2} = 2^{x-1}$ na segunda iteração e assim consecutivamente até chegar em $1 = 2^0$. Precisamo de x passos para chegar no último passo.

Assim, temos o seguinte modelo para o tempo e processamento do algoritmo:

$$t(2^x) = 2c_1 + c_5 + x(3c_2 + c_3 + c_4)$$

Aplicando a mudança de variável, temos que:

$$t(n) = (3c_2 + c_3 + c_4)log_2(n) + 2c_1 + c_5$$

O mesmo modelo que obtivemos no Capítulo **??** a partir das observações.

Exercício 2: Considere o seguinte algoritmo $3Soma$ apresnetado no final do capítulo anterior. Calcule o tempo de processamento em função do tamanho n da entrada assumindo que:

- Cada iteração de variável toma tempo constante c_1
- Cada atribuição toma tempo constante c_2
- Cada soma toma tempo constante c_3
- A saída toma tempo constante c_4

Capítulo 5

Crescimento de Funções

Os modelos que aprendemos a construir para avaliar o tempo de processamento de um algoritmo contém uma série de constantes que dependem do contexto em que os experimentos serão feitos – processador, memória, sistema operacional etc. Uma teoria para comparar algoritmos deveria abstrair esses valores e focar apenas naquilo que se mantém válido em todos os contextos.

Assim, quando comparamos a eficiência de dois algoritmos vamos comparar o tempo processamento de cada um deles como uma função do tamanho da entrada. Além disso, vamos focar especificamente no que acontece para valores grandes da entrada. Em outras palavras, vamoas analisar as funções de maneira *assintótica*.

Considere os dois modelos que vimos no capítulo anterior:

$$\begin{aligned} g(n) &= a \cdot n + b \\ f(n) &= c \cdot log_2(n) + d \end{aligned}$$

Dependendo das constantes a, b, c e d, os valores de $f(n)$ podem ser maiores do que de $g(n)$ para certos valores de n. Mas o g cresce tão mais rápido que f que conforme aumentamos o valor de n, eventualmente g supera f e se mantém maior para sempre independente dos valores das constantes. Dizemos que, assintoticamente g cresce mais rápido do que f.

De fato, nos experimentos que fizemos, é possível notar que a busca binária é mais eficiente do que a sequencial apenas quando começamos a testar com sequências grandes.

Como nos interessa analisar o comportamento assintótico das funções,

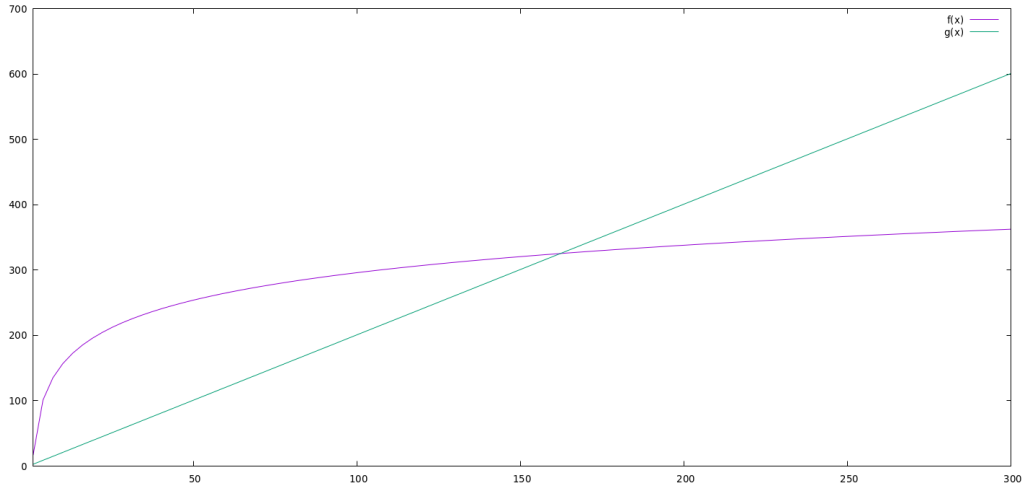


Figura 5.1: Crescimento das funções $g(n) = 2n+1$ e $f(x) = 42 \cdot log_2(n) + 17$. Há um ponto em que as curvas se cruzam e então $g(n)$ passa a ser sempre maior do que $f(n)$.

trataremos como equivalentes funções que crescem de maneira similar uma vez que abstraimos as constantes.

A notação Θ formaliza matematicamente essa ideia:

$$\Theta(g(n)) := \{f(n) : \exists c_1, c_2, n_0 \text{ tal que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$$

Ou seja, $\Theta(g(n))$ é o conjunto de todas as funções que crescem de maneira parecida com g, que são *assintoticamente equivalentes* a g.

Exemplo 5.0.1:

$$3n^2 + 2n \in \Theta(n^2)$$

Para mostrar isso, temos que encontrar constantes c_1, c_2 e n_0 tais que $0 \leq c_1 n^2 \leq 3n^2 + 2n \leq c_2 n^2$ para todo $n \geq n_0$.

Vamos considerar apenas valores de $n \geq 1$. Neste caso, n^2 é sempre positivo e podemos dividir tudo por n^2 sem alterar a direção das ineqações. Obtemos então:

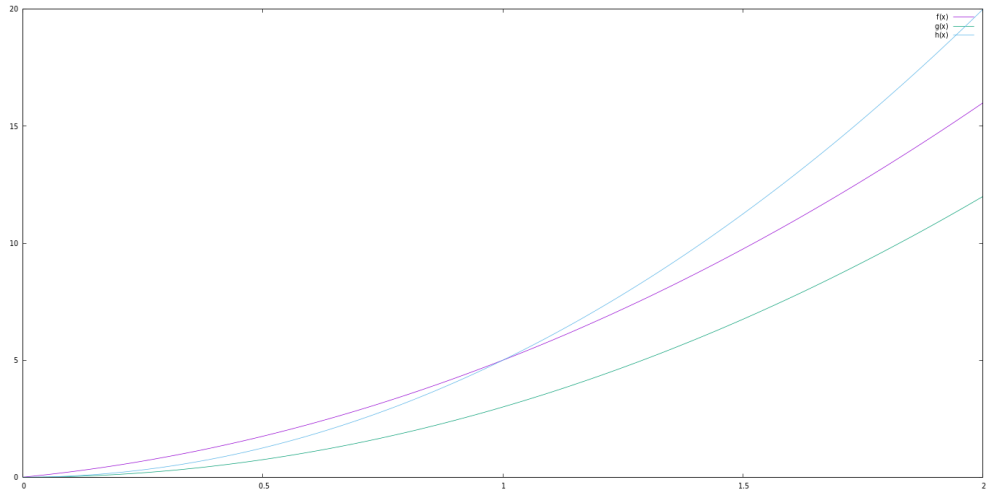


Figura 5.2: A função $f(x) = 3n^2 + 2n$ cresce de maneira assintóticamente equivalente a função n^2 porque $g(x) = 5n^2$ cresce mais rápido que f e $h(x) = 3n^2$ cresce mais devagar que f.

$$c_1 \leq 3 + \frac{2}{n} \leq c_2$$

Agora fica fácil ver que para $c_1 = 3$ e $c_2 = 5$ temos que as inequações valem para qualquer $n \geq 1$:

$$3n^2 \leq 3n^2 + 2n \leq 5n^2$$

Exemplo 5.0.2:

$$4n^4 - 2n^2 + 2 \in \Theta(n^4)$$

Para mostrar isso, temos que encontrar constantes c_1, c_2 e n_0 tais que $0 \leq c_1 n^4 \leq 4n^4 + 2n^2 + 2 \leq c_2 n^4$ para todo $n \geq n_0$.

Novamente, para $n \geq 1$, n^4 é sempre positivo e podemos dividir tudo por n^2 para obter:

$$c_1 \leq 4 - \frac{2}{n^2} + \frac{2}{n^4} \leq c_2$$

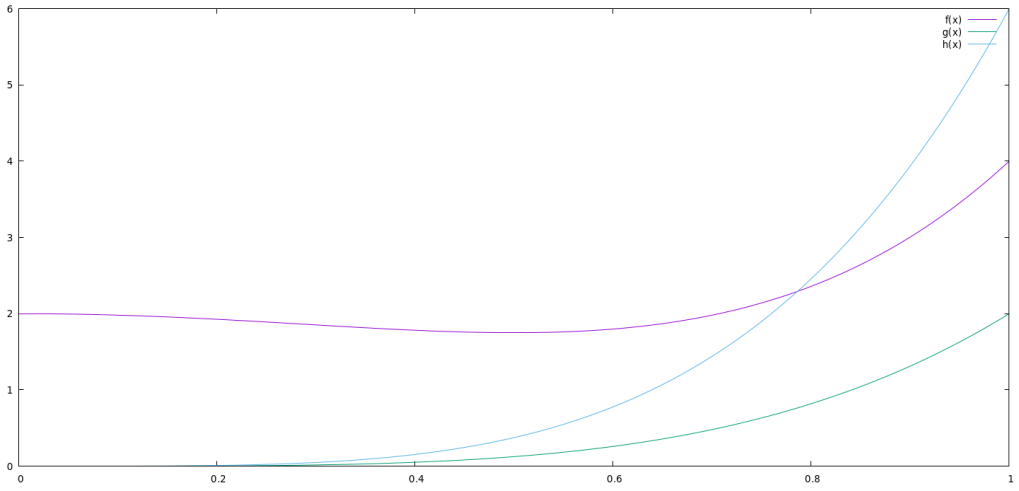


Figura 5.3: A função $f(x) = 2n^4 - 2n + 2$ cresce de maneira assintóticamente equivalente a função n^4 porque $g(x) = 6n^4$ cresce mais rápido que f e $h(x) = 2n^4$ cresce mais devagar que f.

O valor da função $4\frac{2}{n^2} + \frac{2}{n^4}$ é maior do que 2 para e menor do que 6 qualquer valor $n \geq 1$.

Assim, temos que para todo $n \geq 1$:

$$2n^4 \leq 4n^4 - 2n^2 + 2 \leq 6n^4$$

Exemplo 5.0.3:

$$2n^3 \notin \Theta(n^2)$$

Suponha por absurdo que existam c, e n_0 tais que $2n^3 \leq c$ para todo $n \geq n_0$.

Considerando $n > 0$ e dividindo os dois lados por n^2, temos:

$$2n \leq c$$

Mas não é difícil ver que $2n > c$ assim que $n \geq \frac{c}{2}$, contrariando a suposição.

Os últimos três exemplos sugerem uma regra geral: todo polinômio é assintoticamente equivalente ao seu fator dominante.

De fato, esse resultado pode ser enunciado formalmente:

Teorema 5.0.4. *Seja* $p(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_0$ *com* $a_d > 0$ *então* $p(n) \in \Theta(n^d)$

Corolario 5.0.5.

$$a \cdot n + b \in \Theta(n)$$

Dizemos que o consumode de tempo do algortimos Busca Sequencial é $\Theta(n)$, ou simplesmente que o algoritmo é *linear*.

Já o consumo de tempo do algoritmo Busca Binária, como mostra o próximo exemplo, é $\Theta(log(n))$, ou *logarítmico*.

Exemplo 5.0.6:

$$a \cdot log_2(n) + b \in \Theta(log(n))$$

Para mostrar isso, temos que encontrar constantes c_1, c_2 e n_0 tais que $0 \leq c_1 log(n) \leq a \cdot log_2(n) + b \leq c_2 log(n)$ para todo $n \geq n_0$.

Dividindo tudo por $log(n)$, temos:

$$c_1 \leq \frac{a}{log(2)} + \frac{b}{log(n)} \leq c_2$$

Fica fácil ver que as inequações valem para $c_1 = \frac{a}{log(2)}$ e $c_2 = \frac{a}{log(2)} + b$ para qualquer $n \geq 1$.

Algumas classes de funções são bastante recorrentes ao analisar o tempo de processamento de algoritmos, tanto que nos referiremos a elas por um nome específico:

- Constante $\Theta(1)$
- Linear $\Theta(n)$
- Logarítmico $\Theta(log(n))$
- Linearítmico $\Theta(n.log(n))$

- Quadrático $\Theta(n^2)$
- Cúbica $\Theta(n^3)$
- Exponencial $\Theta(2^n)$

Embora a notação Θ represente um conjunto de funções, abusaremos um pouco dela para usá-la em equações. Assim, quando escrevermos por exemplo $n\Theta(n) = \Theta(n^2)$ o que queremos dizer é que para qualquer $f(n) \in \Theta(n)$ temos que $n.f(n) \in \Theta(n^2)$. De fato, podemos mostrar isso:

Exemplo 5.0.7:

$$n\Theta(n) = \Theta(n^2)$$

Se $f(n) \in \Theta(n)$ então, por definição, existem c_1, c_2 e n_0 tais que:

$$0 \leq c_1 \cdot n \leq f(n) \leq c_2 \cdot n \text{ para todo } n \geq n_0$$

Considerando $n_0 > 0$ e multiplicando tudo por n temos que:

$$0 \leq c_1 \cdot n^2 \leq n \cdot f(n) \leq c_2 \cdot n^2 \text{ para todo } n \geq n_0$$

Considere mais uma vez o algoritmo da Busca Sequencial. As duas primeiras linhas são executadas $\Theta(n)$ vezes no pior caso. Já as duas últimas $\Theta(1)$ vezes.

Como isso temos que o tempo de execução em função do tamanho da entrada n no pior caso segue a seguinte função:

$$T(n) = \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$$

Exemplo 5.0.8:

Primeiro note que $\Theta(1)$ é uma constante. Isso porque, por definição temos que:

$$0 \leq c_1 \leq f_3(n) \leq c_2 \text{ para todo } n \geq n_1$$

As outras duas funções precisam respeitar as seguintes inequações:

$$0 \leq c_3 \cdot n \leq f_1(n) \leq c_4 \cdot n \text{ para todo } n \geq n_2$$

$$0 \leq c_5 \cdot n \leq f_2(n) \leq c_6 \cdot n \text{ para todo } n \geq n_3$$

Somando f_1 e f_2 temos que:

$$0 \leq (c_3 + c_5)n \leq f_1(n) + f_2(n) \leq (c_4 + c_6)n \ \forall n \geq max(n_1, n_2)$$

Se somarmos f_3, certamente não alteramos as inequações do lado esquerdo e do lado direito temos:

$$f_1(n) + f_2(n) + f_3(n) \leq (c_2 + c_4 + c_6)n \ \forall n \geq max(n_1, n_2, n_3, 1)$$

A notação Θ indica limites inferior e superior para o crescimento de uma função. Quando avaliamos o tempo e processamento de um algoritmo, é comum desejarmos garantir que ele seja suficientemente eficiente. Ou seja, em geral queremos estabelecer apenas o *limite assintótico superior*. Para tanto, utilizamos a notação O.

$$O(g(n) := \{f(n) : \exists c, n_o \text{ tais que } 0 \leq f(n) \leq cg(n) \ \forall n \geq n_0\}$$

Sempre que $f(n) \in \Theta(g(n))$ temos que $f(n) \in O(g(n))$. Ou seja, $\Theta(g(n)) \subseteq O(g(n))$. Já a recíproca não é necessariamente verdadeira:

Exemplo 5.0.9:

$n \notin \Theta(n^2)$, mas $n \in O(n^2)$

A primeira parte é totalmente análoga ao Exemplo **??**. Para mostrar a segunda, note que:

$$0 \leq n \leq n^2 \text{ para todo } n \geq 1$$

Da mesma forma, existem situações em que nos interessa apenas o *limite assintótico inferior* de uma função. Por exemplo, quando avaliamos certos problemas computacionais, em alguns casos específicos sabemos que é impossível resolver o problema de maneira assintoticamente mais eficiente do que certa função $f(n)$. Nesses casos, dizemos que o problema é $\Omega(f(n))$.

$$\Omega(g(n) := \{f(n) : \exists c, n_o \text{ tais que } 0 \leq cg(n) \leq f(n)\ \forall n \geq n_0\}$$

Quando temos uma solução $O(g(n))$ para um problema $\Omega(g(n))$ dizemos que essa solução é *ótima*.

Exemplo 5.0.10:

O problema da busca em uma sequência arbitrária é $\Omega(n)$. Isso porque precisamos consultar todos n elementos da sequência para saber se o elemento pertence a ela ou não.

Assim, o algoritmo de BuscaSequencial que temos estudado é ótimo para o problema da busca em sequência arbitrária, mas ele não é ótimo para o problema da busca em sequência ordenada. Para este segundo problema conhecemos uma solução $O(log(n))$.

Teorema 5.0.11. *Para qualquer função $g(n)$ temos que $f(n) \in \Theta(g(n))$ se e somente se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$. Em outras palavras:* $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

Quando analisamos o tempo de processamento de um algoritmo como uma função do tamanho da entrada, é consensual a simplificação de desprezar os termos não dominantes e, com isso, simplificar as contas considerando apenas o comportamento assintótico. Há uma controvérsia, porém, sobre a constante multiplicadora do fator dominante. Alguns autores a desprezam por se tratar de um valor que depende do contexto. Outros autores, avaliam que esse valor não deve ser desprezado. No primeiro caso, utilizamos a notação Θ já apresentada. No segundo caso, utilizamos a notação til ($\sim$).

Assim, por exemplo, temos que:

$$4n^3 + 2n + 8 \sim 4n^3 \in \Theta(n^3)$$

Já um exemplo negativo seria:

$$4n^3 + 2n + 8 \nsim 2n^3$$

Formalmente, dizemos que $f(n) \sim g(n)$ se e somente se $lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

Exercício 3: Mostre que o tempo de processamento do algorítimo $3Soma$ é $\Theta(n^3)$ no pior caso.

Capítulo 6

Algoritmos de Ordenação

O problema da ordenação é central na análise de algoritmos por dois motivos. Em primeiro lugar, ele é um problema com muitas aplicações em diversas áreas da computação. Em segundo lugar, são conhecidas diversas soluções bem diferentes para esse problema, o que o torna um excelente exemplo para a teoria que estamos apresentando.

Vamos relembrar o problema:

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 $i < j$.

6.1 Selection Sort

Comecemos com uma solução bem simples para o problema, um algoritmo chamado Selection Sort. Esse algortimo consiste em identificar o maior elemento da sequência e trocá-lo com o último e repetir o processo até que todos estejam em ordem.

Assim, podemos começar com a seguinte sub-rotina:

MAXIMO(A)
1 ▷ Recebe uma sequência $a_1, \ldots, a_n$
2 ▷ Devolve i tal que a_i é o maior elemento da sequência
3 $imax \leftarrow 0$
4 **for** $i \leftarrow 2$ até n
5 **do if** $a_i > a_{imax}$
6 **then** $imax \leftarrow i$
7 **return** $imax$

Para provar que este algoritmo é correto, note que a seguinte propriedade é invariante na linha 2:

$$a_{imax} \text{ é o maior elemento em } a_i, \ldots, a_{i-1}$$

Inicialização: na primeira vez que passamos pela linha 2 temos que $i = 2$. Portanto, a sequência $a_1, \ldots, a_{i-1}$ tem apenas um elemento (a_1) que só pode ser o maior.

Manutenção: se na iteração anterior a_{imax} era o maior elemento em $a_1, \ldots, a_{i-1}$ temos duas possibilidades: se a_i fosse maior que a_{imax} então teríamos atualizado $imax$ e ele continuaria sendo o maior da sequência, caso contrário $imax$ não teria sido atualizado, mas a_{imax} continuaria sendo o maior elemento.

Término: quando chegamos na linha 5 temos que a_{imax} é o maior elemento de $a_1, \ldots, a_n$.

Analisar o tempo de processamento deste algoritmo também é simples. Para facilitar as contas, faremos mais uma simplificação. Ao invés de somar o tempo de processamento de cada linha, calcularemos o tempo apenas de uma das linhas que mais se repete, neste caso escolhemos a 3. Essa linha é processada n vezes e, portanto, o algortimo é $\Theta(n)$.

Apresentemos agora o algoritmo Selection Sort.

SELECTIONSORT(A)
1 **for** $j \leftarrow n$ até 2
2 **do** $imax \leftarrow Maximo(a[1 : j])$
3 **then** $a_{imax} \leftrightarrow a_j$

A propriedade invariante depois da linha 3, neste caso, é:

$a_j, \ldots, a_n$ está em ordenada e a_j é o maior elemento de $a_1, \ldots, a_j$

Inicialização: Na primeira passagem temos que $j = n$ e, portanto a squência $a_j, \ldots, a_n$ tem um único elemento a_n, além disso, pela correção do algoritmo $Maximo$, a_n é o maior elemento da sequência $a_1, \ldots, a_n$.

Manutenção: Pela hipótese de indução $a_j, \ldots, a_n$ estava ordenada na iteração anterior e a_j é o maior elemento de $a_1, \ldots, a_j$, assim $a_{j-1} \leq a_j$ e temos que $a_{j-1}, \ldots, a_n$ está em ordem crescente. Além disso, pela correção do algortimo $Maximo$ temos que a_{j-1} é o maior elemento em $a_1, \ldots, a_{j-1}$.

Término: Quando saimos do laço, $j = 2$ e o invariante garante, portanto, que $a_2, \ldots, a_n$ está ordenado. Além disso, ele garante que a_2 é o maior elemento da sequência a_1, a_2. Concluímos que ao término $a_1, \ldots, a_n$ está ordenado.

Para analisar o tempo de processamento deste algoritmo, consideraremos a linha 2 que se repete n vezes. A operação executada nessa linha, porém, não é atômica. Como vimos, ela toma tempo proporcional ao tamanho da entrada que varia em cada iteração. Na primeira iteração toma tempo proporcional a n, na segunda $n-1$, então $n-2$ e assim por diante.

O tempo de processamento em função do tamanho n da entrada, então, é:

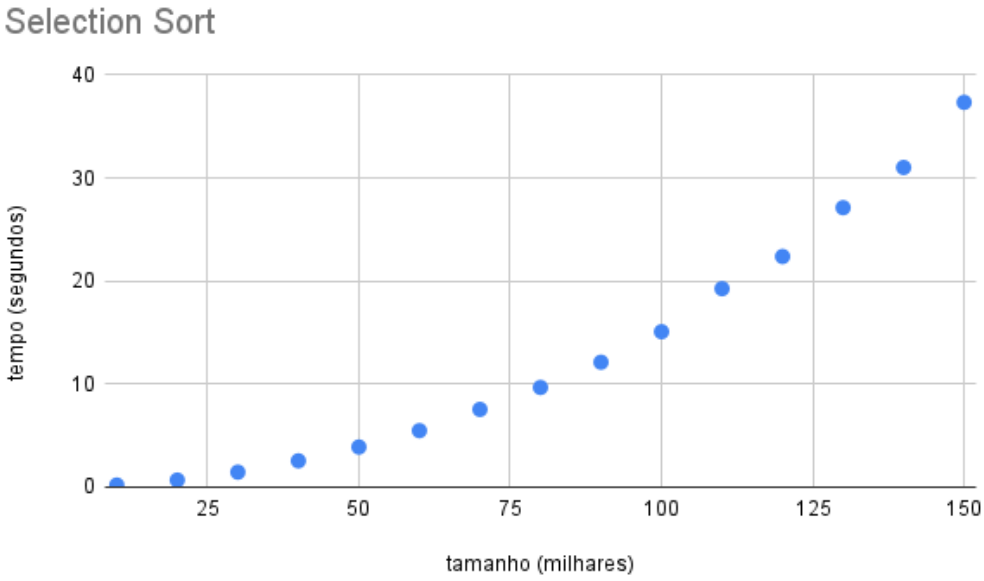
$$T(n) = n + (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

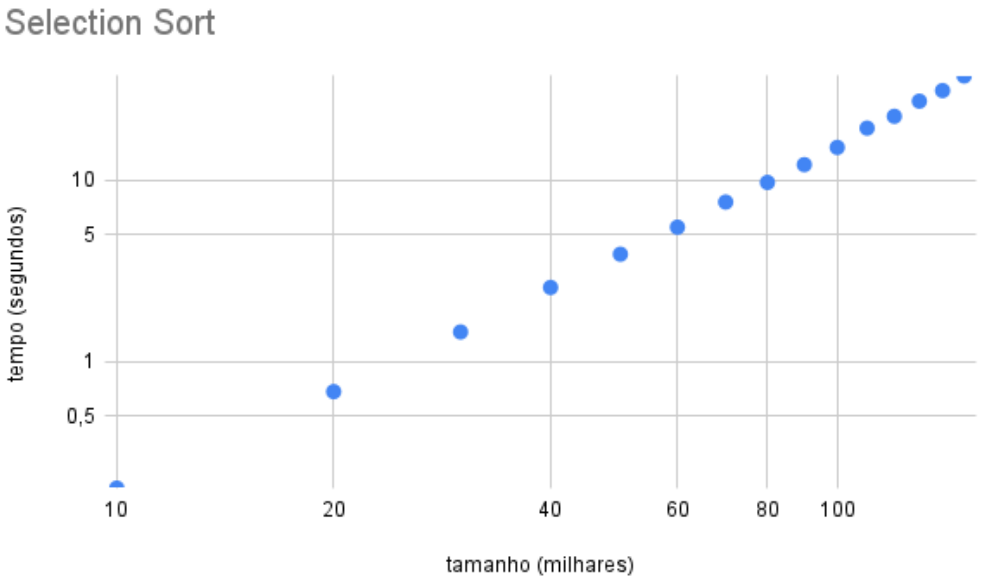
Assim se calcularmos o tempo de processamento de uma implementação desse algoritmo para entradas de diferentes tamanhos, devemos observar uma parábola (Figura **??**):

Podemos testar que se trata de fato de uma parábola poltando os mesmo valores em um gráfico em que ambos os eixos estão em escala logarítmica. Se a função tiver o formato $T(n) = an^2$, como nossa análise sugere, ao aplicar o *log* nos dois lados, temos que:

$$log(T(n)) = log(an^2) = 2(log(n) + log(a)) = 2log(n) + 2log(a)$$

Assim, o que obtemos é a equação de uma reta com inclinação 2. Na Figura **??** observamos que de fato em uma gráfico log-log o que obtemos é uma reta com inclinação $1,94$, valor bem próximo do esperado.


Selection Sort
40
30
20
10
0
tempo (segundos)
25
50
75
100
125
150
tamanho (milhares)


Selection Sort
10
5
1
0,5
tempo (segundos)
10
20
40
60
80
100
tamanho (milhares)

6.2 Insertion Sort

Nosso segundo algoritmo segue uma ideia também simples. Ele ordena uma sequência de maneira análoga a forma como muitas pessoas ordenam cartas de baralhos nas mãos.

A princípio as cartas estão todas viradas para baixo. Elas são inseridas uma por uma na mão do jogador cada qual em sua posição relativa correta.

```
InsertionSort(A)
1  for j ← 2 até n
2  chave ← a_j
3  for i ← j − 1 até 1
4        do a_{i+1} ← a_i
5           if a_i > chave
6              then sai do laço
7  a_{i+1} ← chave
```

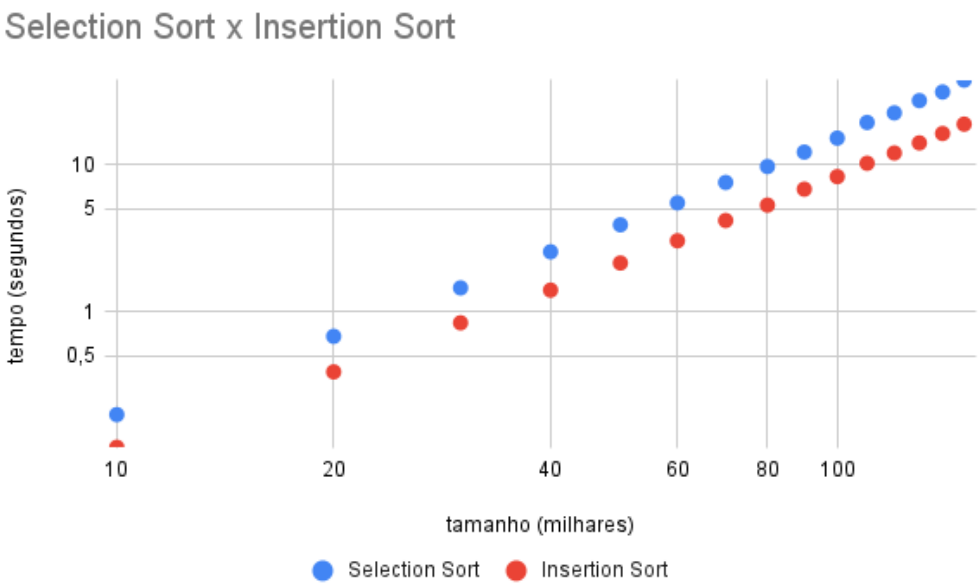
Para provar a correção deste algoritmo, note que a seguinte propriedade é invariante na linha 2:

$$\text{a sequência } a_1, \ldots, a_{j-1} \text{ está em ordem crescente}$$

A análise do consumo de tempo é similar à do SelectionSort. Uma diferença está no fato de que independente da entrada, o SelectionSort sempre processa todos elementos já o InsertionSort pode sair do laço interno precocemente. Note então que, quando o pior caso ocorre quando os elementos originalmente estão em ordem descrescente. Neste caso, a análise do consumo de tempo é idêntica à do SelectionSort:

$$T(n) = n + (n-1) + (n-2) + \cdots + 2 = \sum_{i=2}^{n} = \frac{n(n+1)}{2} - 1 \in \Theta(n^2)$$

Assim, no pior caso os algoritmos são identícos. Veremos como analisar o caso médio na seção sobre o QuickSort, por ora basta dizer que ela também é $\Theta(n^2)$ neste caso, porém a constante multiplicadora é menor. Isso fica evidente quando verificamos o modelo. Plotando o tempo e processamento dos dois algoritmos que vimos até aqui em um mesmo gráfico com escala



log nos dois eixos, vemos que ambos apresentamos uma reta praticamente com a mesma inclinação – o primeiro tinha inclinação 1,94 e este 1,88. A diferença é no fato de deslocamento vertical que indica uma diferença no fator multiplicador do termo dominante. Essa diferença é significativa porque testamos o algoritmo com uma entrada aleatória e, nesse caso, o InsertionSort é um pouco mais eficiente:

6.3 Merge Sort

Antes de apresentar o próximo algoritmo, estudaremos um paradigma de projeto de algoritmos chamado *divisão-e-conquista* que consiste nos seguintes passos:

- *Divisão:* a instância original do problema é dividida em um número de instâncias menores.
- *Conquista:* instâncias suficientemente pequenas são resolvidas diretamente.
- *Combinação:* se necessário, as soluções das instâncias menores são combinadas em para resolver a instância original.

Para exemplificar o paradigma, revisitaremos o algoritmo da busca binária em uma versão recursiva.

BuscaBinaria(A, b)

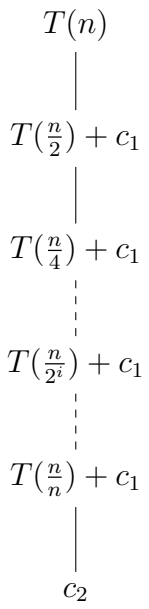
```
1  ▷ recebe uma sequência ordenada a_1, ..., a_n e um valor b
2  ▷ devolve V se b está na sequência e F c.c.
3  if n = 1
4     then if a_1 = b
5             then return V
6     else
7          return F
8  m ← ⌊n/2⌋
9  if b ≤ a_m
10    then return BuscaBinaria(A[1 : m])
11    else  return BuscaBinaria(A[m + 1 : n])
```

As linhas 1 a 5 resolvem o caso base (*conquista*) enquanto as linhas 6 a 10 dividem o problema em instâncias menores. Neste caso, não há necessidade de combinar as soluções pois a cada passo, metade da instância original pode ser ignorada.

Para analisar o tempo de processamento deste algoritmo primeiro estabelecemos que as linhas 1 a 5 tomas tempo constante (c_2), assim comas as linhas 6, 7 e 9 (c_1). Jás as linhas 8 e 10 são chamadas recursivas. Assim, o tempo de processamento do pior caso em função do tamanho da entrada n é descrito pela seguinte recorrência:

$$\begin{aligned} T(n) &= c_1 + T(\left\lceil \frac{n}{2} \right\rceil) \\ T(1) &= c_2 \end{aligned}$$

Resolveremos essa recorrência utilizando o método da árvore de recorrência:



A árvore chega no caso base $T(1)$ quando $2^i = n$, ou seja, quando $i = lg(n)$. Portanto, a altura da árvore é $lg(n)$. Em cada passo, é somado c_1 e no último é somado c_2. Concluímos que:

$$T(n) = c_1.lg(n) + c_2$$

Talvez um exemplo melhor do paradigma da divisão-e-conquista seja um versão do algoritmo de busca para sequências arbitrárias. Dividiremos a sequência ao meio, mas neste caso, é preciso buscar nas duas metades dela. Por fim, precisamos combinar os resultados. O elemento b está na sequência se ele está na metade direita ou na metade esquerda.

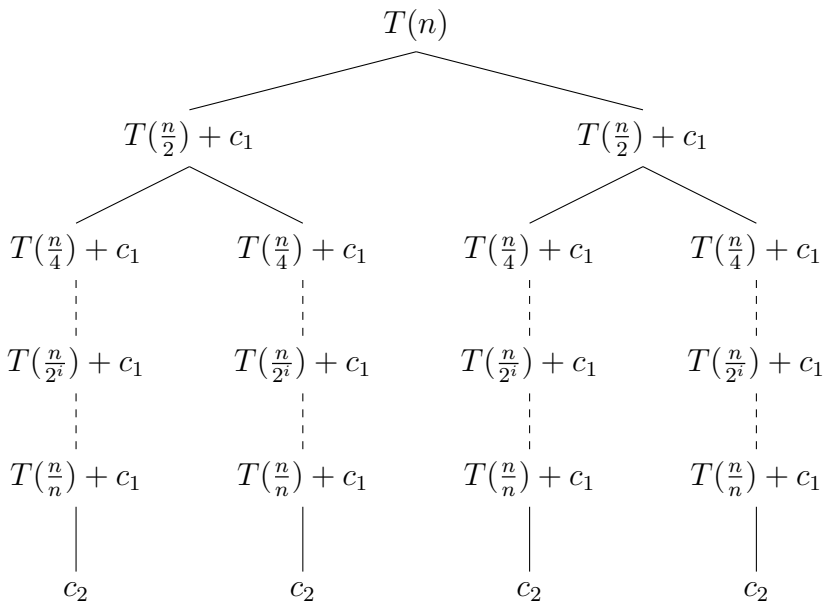
BuscaRecursiva(A, b)

```
1  ▷ recebe uma sequência a_1, ..., a_n e um valor b
2  ▷ devolve V se b está na sequência e F c.c.
3  if n = 1
4      then if a_1 = b
5               then return V
6      else
7           return F
8  m ← ⌊n/2⌋
9  return BuscaRecursiva(A[1 : m]) ou BuscaRecursiva(A[m + 1 : n])
```

Como no exemplo anterior, o caso base (linhas 3 a 7) toma tempo constante c_2 e as operações de divisão, atribuição e o ou lógico (linhas 8 e 9) também tomam tempo constante c_2. Desta vez, porém, a recursão é aplicada nas duas metades da sequência. Assim, a recorrência que devemos resolver é:

$$\begin{aligned} T(n) &= c_1 + T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) \\ T(1) &= c_2 \end{aligned}$$

Para calcular o resultado dessa recorrência, vamos novamente analisar sua árvore:



Como no caso anterior, essa árvore chega às folhas quando $2^i = n$, ou seja, quando $i = lg(n)$. Portanto, a altura dela é $lg(n)$. Em cada nível dessa árvore dobramos a quantidade de c_1 somadas. Assim, no i-ésimo nível são somados 2^i constantes e no último são somados $2^{lg(n)} = n$ constantes. Portanto, o resultado de nossa recorrência é:

$$\begin{aligned} T(n) &= c_1.\sum_{i=1}^{lg(n)} 2^i + c_2.n \\ &= c_1.\frac{1-2^{lg(n)}}{1-2} + c_2.n \\ &= c_1.(n-1) + c_2.n \\ &= (c_1+c_2).n - c_1 \\ &\in \Theta(n) \end{aligned}$$

O merge sort, ou algoritmo de ordenação por intercalação, também segue o paradigma da divisão-e-conquista. A ideia é quebrar a instância original ao meio até que ela tenha apenas um elemento. Neste caso, o caso base, não é preciso fazer nada pois qualquer sequência com um único elemento está trivialmente ordenada. O segredo está na combinação das instâncias menores para resolver a instância original. Para este passo, usaremos um algoritmo de intercalação.

Podemos enunciar o problema da intercalação da seguinte forma:

Problema da intercalação

Entrada: Duas sequências $a_1, \ldots, a_n$ e $b_1, \ldots, b_m$ ambas em ordem crescente.

Saída: Uma sequência $c_1, \ldots, c_{m+n}$ em ordem crescente cujos elementos são os mesmos das sequências da entrada.

Para resolver esse problema utilizamos o seguinte algoritmo:

```
MERGE(A, B)
 1  i ← 1
 2  j ← 1
 3  for k ← 1 até n + m
 4        do if (a_i ≤ b_j e i ≤ n) ou j > m
 5              then c_k ← a_i
 6                   i ← i + 1
 7              else
 8                   c_k ← b_j
 9                   j ← j + 1
10  return C
```

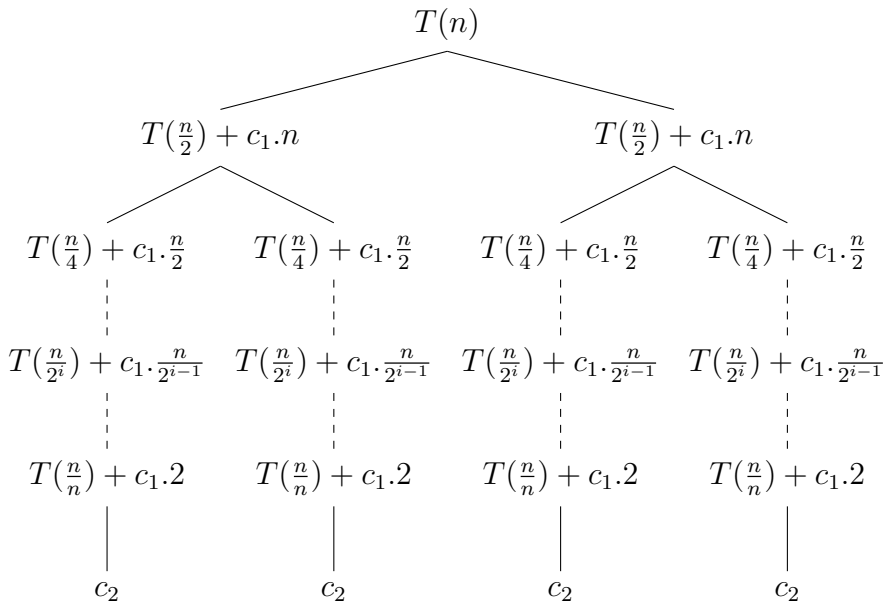
Com isso, podemos escrever o algoritmo de ordenação da seguinte forma:

MergeSort(A, B)
1 **if** $n > 1$
2 **then** $m \leftarrow \lfloor \frac{n}{2} \rfloor$
3 $MergeSort(A[1 : m])$
4 $MergeSort(A[m + 1 : n])$
5 $A \leftarrow Merge(A[1 : m], A[m + 1 : n])$

O caso base do algoritmo ocorre quando $n = 1$. Neste caso, nada precisa ser feito e o tempo tomado é constante c. Já a linha 4 toma tempo linear no tamanho da entrada n. Com isso, temos que o tempo de execução do algoritmo em função de n é:

$$\begin{aligned} T(n) &= c_1.n + T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) \\ T(1) &= c_2 \end{aligned}$$

Analisaremos, então a seguinte árvore de recorrência:



Em cada nível da árvore, é somado $2.c_1.n$. Como a altura dessa árvore também é $lg(n)$ temos que:

$$T(n) = 2.c_1.n.lg(n) + 2.n.c_2 \in \Theta(n.lg(n))$$

Ambos os algoritmos de ordenação que vimos nas seções anteriores, selection sort e insertion sort, tomam tempo $\Theta(n^2)$ no pior caso. Comparado a esses, o merge sort é muito mais eficiente. Isso porque a função $lg(n)$ cresce muito mais devagar do que a função n. Na Figura 6.3 plotamos em um grafíco log-log o resultado de um experimento em que foi computado o tempo de processamento de sequências aleatórias para cada um três algoritmos. Para sequências de 100 mil elementos, o tempo de processamento dos algoritmos básicos tomou tempo próximo a 10 segundo e esse tempo aumentou para cerca de 100 segundos quando aumentamos o número de elementos para 300 mil. Já o tempo de processamento do merge sort variou próximo de 0,1 segundos. Além disso, note que as inclinações das retas nos dois primeiros casos é próxima de 2. Como discutido anteriormente esse era o resultado esperado visto que os algoritmos tomam tempo $\Theta(n^2)$. A inclinação da reta que corresponde ao tempo de processamente do merge sort é menor, mais próxima de 1 do que de 2.

Agora que vimos como resolver três recorrências diferentes usando o método da árvore de recorrência, apresentaremos um resultado geral para resolver recorrências:

Teorema 6.3.1 (Mestre)**.** *Sejam $a \geq 1$, $b > 1$ e $T(n) = aT(\frac{n}{b}) + f(n)$, então:*

1. *se $f(n) \in O(n^{log_b(a-\epsilon)})$ para algum $\epsilon > 0$ então $T(n) \in \Theta(n^{log_b a})$*
2. *se $f(n) \in \Theta(n^{log_b a})$ então $T(n) \in \Theta(n^{log_b a} lg(n))$*
3. *se $f(n) \in O(n^{log_b(a+\epsilon)})$ para algum $\epsilon > 0$ e se $af(\frac{n}{b}) \leq cf(n)$ para algum $c < 1$ e todo n suficientemente grande então $T(n) \in \Theta(f(n))$*

Exemplo 6.3.2:

Comecemos pela recorrência que acabamos de analisar

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

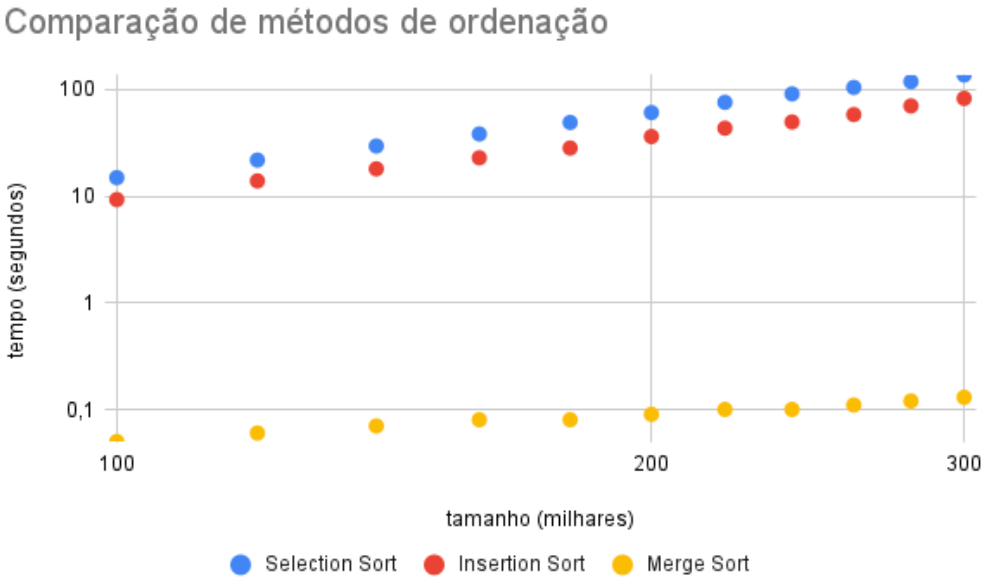


Figura 6.1: Resultado de experimento para calcular o processamento dos algoritmos de ordenação vistos até agora em um gráfico log-log.

O primeiro passo é avaliar $n^{log_b a}$ que neste caso é $n^{log_2 2} = n$. Então temos que verificar se $f(n) = cn \in \Theta(n)$. Neste caso, isso vale. Portante se aplica o caso 2 do teorema e temos que:

$$T(n) \in \Theta(n^{log_2 2}.lg(n)) = \Theta(n.lg(n))$$

Exemplo 6.3.3:

Considere a recorrência que obtivemos ao analisar a busca binária:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

O primeiro passo é avaliar $n^{log_b a}$ que neste caso é $n^{log_2 1} = n^0 = 1$. Então temos que verificar se $f(n) = c \in \Theta(1)$. Neste caso, isso vale. Portante se aplica o caso 2 do teorema e temos que:

$$T(n) \in \Theta(n^{log_2 1}.lg(n)) = \Theta(1.lg(n)) = \Theta(lg(n))$$

Exemplo 6.3.4:

Considere agora a recorrência que obtivemos ao analisar a busca recursiva:

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$

Primeiro avaliamos $n^{log_2 2} = n$. Então temos que verificar se $f(n) = c \in \Theta(n)$. Neste caso, isso não é verdade c cresce mais lentamente do que n.

Se $\epsilon = 1$ temos que $n^{log_2(2-1)} = n^0 = 1$ e $c \in O(1)$. Portanto podemos aplicar o caso 1 do teorema e temos que:

$$T(n) \in \Theta(n^{log_2 2} = \Theta(n^1) = \Theta(n)$$

Exemplo 6.3.5:

Para terminar, considera a seguinte recorrência:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Primeiro avaliamos $n^{log_2 2} = n$. Então temos que verificar se $f(n) = n^2 \in \Theta(n)$. Neste caso, isso não é verdade n cresce mais rapidamente do que n.

Se $\epsilon = 2$ temos que $n^{log_2(2+2)} = n^2$ e $n^2 \in \Omega(n^2)$ e $2(\frac{n}{2})^2 = \frac{n^2}{2} \leq \frac{1}{2}.n^2$ para todo $n > 0$. Portanto podemos aplicar o caso 3 do teorema e temos que:

$$T(n) \in \Theta(f(n)) = \Theta(n^2)$$

6.4 Quick Sort

6.5 Heap Sort

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.