Estrutura de Dados

Universidade Federal de Minas Gerais

RASCUNHO, março de 2024

Prefácio

Este texto tem como propósito primordial (1) atender à necessidade de um material que apresente os tópicos abordados na disciplina de *Estrutura de Dados* de forma organizada e (2) oferecer suporte a estudantes que não tenham cursado a disciplina de *Matemática Discreta*, promovendo um encontro mais suave com os tópicos em matemática do conteúdo.

O texto está sendo aprimorado, e ficarei grato por qualquer sugestão ou comentário que você (leitor) possa enviar para o e-mail: augustoguerradelima@proton.me.

Belo Horizonte, fevereiro de 2024
Augusto Guerra de Lima
☺

Sumário

1 Complexidade de algoritmos

Antes de tudo, uma pergunta simples: Quantas comparações o laço de iteração abaixo deve fazer até parar?

```
for ( i :=0;  i<n;  i++)
```

O laço realiza $n+1$ comparações e repete n vezes. Esse tipo de contagem é importante para determinar a complexidade de um algoritmo, isso será feito junto de uma notação matemática chamada **notação assintótica**.

Tempo de execução, armazenamento e robustez são algumas características que determinam a qualidade de um algoritmo.

A **complexidade do tempo de execução** de um algoritmo é avaliada para estimar sua eficiência antes da implementação. A eficiência é representada por uma função que recebe o tamanho da entrada do algoritmo como parâmetro.

Ao longo das sessões será utilizado o modelo *random-access machine* (RAM), assim consideramos que um único processador realiza as tarefas de forma sequencial, com operações básicas como as aritméticas ou comparações, com custo constante e com os dados armazenados na memória.

1.1 Crescimento assintótico de funções

A **análise assintótica** avalia o comportamento de uma função para valores de parâmetros *significativamente grandes*. Por exemplo, ao ver a expressão n^2+10 é comum pensar em valores pequenos de n, a análise assintótica se preocupa com valores enormes de n onde a constante 10 por exemplo, não faria diferença e o termo n^2 dominaria assintoticamente o crescimento da função.

As notações utilizadas para descrever complexidade são definidas em termos de funções cujos domínios são o conjunto dos números inteiros não negativos. Definiremos que $\mathbb{N}_0 = \{0,1,2...\}$ e $\mathbb{R}^+$ como o conjunto dos números reais não negativos.

Sejam f e g funções tais que a taxa de crescimento de f é maior que a de g, podemos denotar $g \prec f$.

Antes de introduzir a notação assintótica, vamos avaliar um exemplo preliminar.

Exemplo 1 (Comparando funções)

Sejam f e g funções definidas no conjunto dos números inteiros não negativos, para comparar o comportamento assintótico de g e f, é preciso encontrar uma constante k tal que $g(n) \leq kf(n)$ para todo n suficientemente grande (escreveremos $n \geq n_0$). Isto é, encontrar constantes k e n_0.

Sabendo $g(n) = \lceil \frac{n}{2} \rceil + 10$ e $f(n) = n$, encontrar k e n_0 tais que $g(n) \leq kf(n), \forall n \geq n_0$.

$$\left\lceil \frac{n}{2} \right\rceil \leq kn; \quad \left\lceil \frac{n}{2} \right\rceil \leq \frac{n}{2} + 11 \leq \frac{n}{2} + \frac{n}{2} = n.$$

Se $k = 1$ e $n_0 = 21$, por exemplo, então $f(n_0) = \lceil \frac{21}{2} \rceil + 10 = 21$ e $g(n_0) = 21$ e uma solução foi encontrada. Podem haver mais soluções.

•

Exemplo 2 (Caso sem solução)

Se $g(n) = n^3$ e $f(n) = n^2$, encontrar k e n_0 tais que $g(n) \leq kf(n),\ \forall n \geq n_0$.

$$n^3 \leq kn^2 \Rightarrow n \leq k,\ \forall n \geq n_0.$$

É impossível que uma função linear seja sempre menor ou igual a uma constante, portanto não há solução.

•

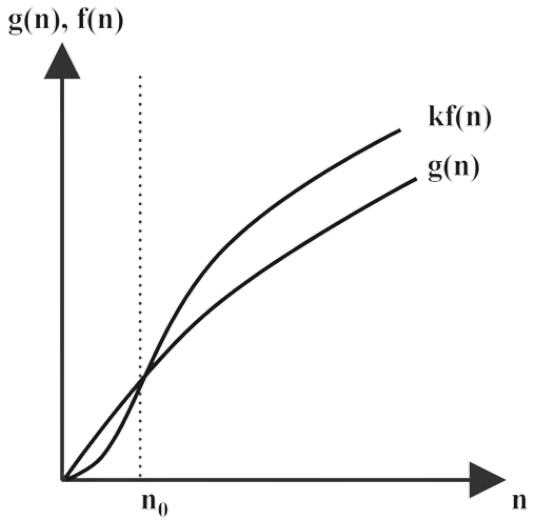

Figura 1: $g(n) \in O(f(n))$.

1.1.1 Notação O

A notação $O(f(n))$ (O grande de f de n), é o **limite assintótico superior justo** é a mais utilizada para descrever a complexidade de um algoritmo. Para uma função $f(n)$, $O(f(n))$ é o conjunto de funções

$$O(f(n)) = \{g : \mathbb{N}_0 \to \mathbb{R}^+ : \ \exists k,\ n_0 > 0;\ 0 \leq g(n) \leq kf(n),\ \forall n \geq n_0\}.$$

$O(f(n)) = \{$"Todas as funções que têm taxa de crescimento menor ou igual a $f(n)$"$\}$.

Dizer $g(n) \in O(f(n))$ é dar um limite superior para $g(n)$, ou seja, intuitivamente para todos os valores $n \geq n_0$, encontrar uma família de funções com taxa de crescimento maior que $g(n)$. A figura 1 mostra um exemplo gráfico.

Utilizando a notação O, muitas vezes é possível determinar o tempo de execução de um algoritmo apenas inspecionando sua estrutura global. Um laço de iteração que fizer $n+1$ comparações por exemplo, tem tempo de execução $O(n)$.

Exemplo

Sejam $g(n) = 2n^2 + n$ e $f(n) = n^2$, é verdadeiro que $g(n) \in O(f(n))$ ou melhor, $2n^2 + n \in O(n^2)$.

Substituindo na desigualdade e simplificando os termos

$$2n^2 + n \leq kn^2 \Rightarrow 2 + \frac{1}{n} \leq k \forall n \geq n_0$$

$k = 3$ e $n_0 = 1$ satisfazem $g(n) \in O(f(n))$. Basta substituir os valores na desigualdade e verificar

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

A desigualdade é verdadeira para todo $n \geq 1$, em outras palavras $n_0 = 1$.

•

1.1.2 Notação Ω

A notação $\Omega(f(n))$ (Ômega de f de n), é o **limite assintótico inferior justo**. Nesse caso, dizer $g(n) = \Omega(f(n))$, significa que existe uma constante k tal que $0 \leq kf(n) \leq g(n)$ para todo $n \geq n_0$. Para uma função $f(n)$, $\Omega(f(n))$ é o conjunto de funções

$$\Omega(f(n)) = \{g : \mathbb{N}_0 \to \mathbb{R}^+ : \ \exists k,\ n_0 > 0;\ 0 \leq kf(n) \leq g(n),\ \forall n \geq n_0\}.$$

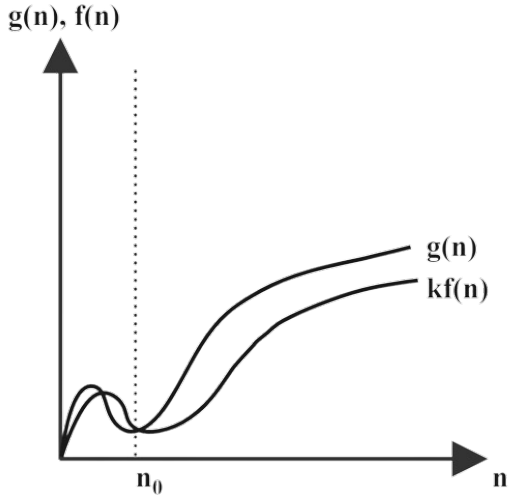

Figura 2: $g(n) \in \Omega(f(n))$.

$\Omega(f(n)) = \{$"Todas as funções que têm taxa de crescimento maior ou igual a $f(n)$"$\}$.

É importante notar que $g(n) \in O(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$. Essa propriedade é chamada de **simetria de transposição**.

Dizer que a complexidade de tempo de execução de um algoritmo é $\Omega(f(n))$ significa que seu tempo de execução será no mínimo $kf(n)$ para uma entrada de tamanho $n \geq n_0$.

Exemplo

Seja $f(n) = 2n^2$, verifique se $f(n) \in \Omega(n^2)$, $f(n) \in \Omega(n\log(n))$ e $f(n) \in \Omega(n)$, utilizando a definição de taxa de crescimento.

Como $n \prec n\log(n) \prec n^2 \approx 2n^2$, então é possível dizer que $2n^2 \in \Omega(n^2)$, $2n^2 \in \Omega(n\log(n))$ e $2n^2 \in \Omega(n)$.

•

1.1.3 Notação Θ

$g(n) \in \Theta(f(n))$(g de n está em theta de f de n) significa que $f(n)$ é o **limite assintótico restrito** de $g(n)$, existem constantes k_0 e k_1 de forma que $0 \leq k_0 f(n) \leq g(n) \leq k_1 f(n)$, $\forall n \geq n_0$. Ou seja, o conjunto de funções

$$\Theta(f(n)) = \{g : \mathbb{N}_0 \to \mathbb{R}^+ : \ \exists k_0,\ k_1,\ n_0 > 0;\ 0 \leq k_0 f(n) \leq g(n) \leq k_1 f(n),\ \forall n \geq n_0\}.$$

Se $g(n) \in \Theta(f(n))$, então $g(n) \in O(f(n))$ e $f(n) \in \Omega(g(n))$, a recíproca também é válida.

Além disso $g(n) \in \Theta(f(n)) \Leftrightarrow f(n) \in \Theta(g(n))$.

Exemplo

Mostre que $\frac{n^2}{2} - 3n \in \Theta(n^2)$.

O problema consiste em encontrar $k_0,\ k_1,\ n_0 > 0$ tais que $k_0 n^2 \leq \frac{n^2}{2} - 3n \leq k_1 n^2\ \forall n \geq n_0$.

Dividindo a expressão por n^2

$$k_0 \leq \frac{1}{2} - \frac{3}{n} \leq k_1\ \forall n \geq n_0.$$

Para a inequação da direita $\frac{1}{2} - \frac{3}{n} \leq k_1$, escolhendo $n_0 = 1$ e $k_1 = \frac{1}{2}$ encontra-se $\frac{1}{2} - 3 \leq \frac{1}{2}$, satisfazendo a primeira desigualdade.

Na segunda inequação $\frac{1}{2} - \frac{3}{n} \geq k_0$, se escolhermos $k_0 = \frac{1}{14} < k_1$ e $n_0 = 7$ a desigualdade é satisfeita.

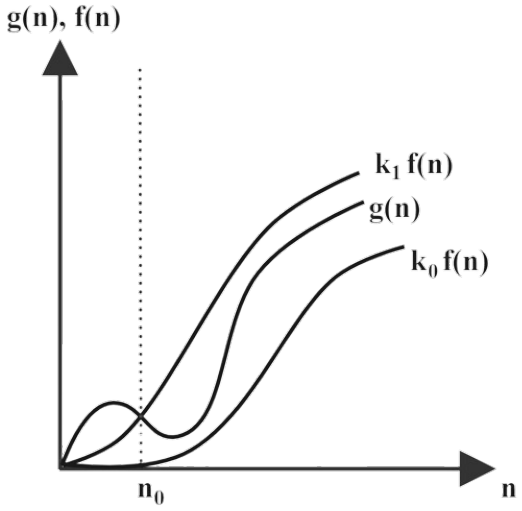

Figura 3: $g(n) \in \Theta(f(n))$. Entre as funções das extremidades existe uma família de funções pertencentes a $\Theta(f(n))$.

Dessa forma, para $n \leq 7$ ou seja$n_0 = 7$, são escolhidas $k_0 = \frac{1}{14}$ e $k_1 = \frac{1}{2}$, e pela definição $\frac{n^2}{2} - 3n \in \Theta(n^2)$.

•

1.1.4 Notação o e ω

As duas últimas notações são usadas para **limites assintoticamente não justos**, o para superiores e ω para inferiores.

Por exemplo, seja λ uma constante maior que zero, $\lambda n^2 = O(n^2)$ mas $\lambda n^2 \neq o(n^2)$. Ademais, $\lambda n = o(n^2)$.

$$o(f(n)) = \{g : \mathbb{N}_0 \to \mathbb{R}^+ : \ \exists k, n_0 > 0;\ 0 \leq g(n) < kf(n),\ \forall n \geq n_0\}.$$

Os limites abaixo revelam que à medida que $n \to \infty$, $g(n)$ torna-se insignificante em relação à $f(n)$.

$$lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty;\ \ lim_{n\to\infty}\frac{g(n)}{f(n)} = 0.$$

De forma análoga, $\lambda n^2 = \omega(n)$ mas $\lambda n^2 \neq \omega(n^2)$, contudo $\lambda n^2 = \Omega(n^2)$.

$$\omega(f(n)) = \{g : \mathbb{N}_0 \to \mathbb{R}^+ : \ \exists k, n_0 > 0;\ 0 \leq kf(n) < g(n),\ \forall n \geq n_0\}.$$

Os limites abaixo revelam que à medida que $n \to \infty$, $f(n)$ torna-se insignificante em relação à $g(n)$.

$$lim_{n\to\infty}\frac{g(n)}{f(n)} = \infty;\ \ lim_{n\to\infty}\frac{f(n)}{g(n)} = 0.$$

É claro que as notações o e ω respeitam a identidade de simetria de transposição entre si.

Exemplo

Prove que $2n \in o(n^2)$ mas $2n^2 \notin o(n^2)$.

$$\lim_{n\to\infty}\frac{2n}{n^2} = 0;\ \ \lim_{n\to\infty}\frac{2n^2}{n^2} = 2.$$

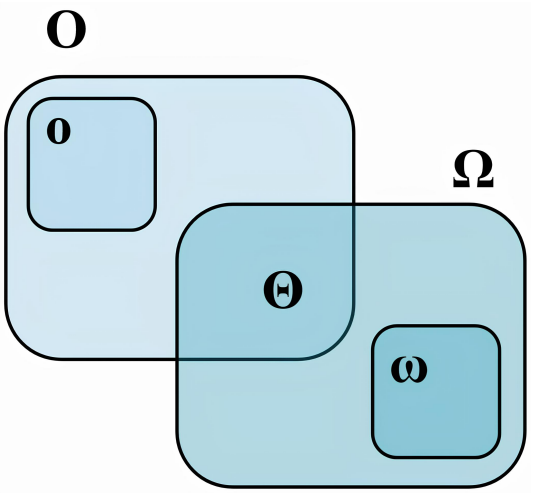

Figura 4: Conjunto de funções nas notações.

Assim, $2n \in o(n^2)$ é uma expressão válida para todos os valores de $k > 0$ e $n_0 = 1$ e como o limite no segundo caso não tende a zero, não existem valores $k > 0$ e n_0 que satisfaçam a definição da notação o, portanto $2n^2 \notin o(n^2)$.

•

1.2 Identidades de notação assintótica

As comparações assintóticas respeitam algumas identidades, considerando funções f e g assintoticamente positivas.

Reflexividade

$$f(n) \in \Theta(f(n)).$$

Aplica-se também as notações O e Ω.

Transitividade

$$f(n) \in \Theta(f(n)),\ g(n) \in \Theta(h(n)) \Rightarrow f(n) \in \Theta(h(n)).$$

Aplica-se para todas as outras notações.

Simetria

$$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n)).$$

Outras

$$O(n+m) = O(n) = O(m).$$

$$O(f(n)) + O(g(n)) = O(max(f(n), g(n))).$$

$$g(n)O(f(n)) = O(f(n))O(g(n)) = O(f(n)g(n)).$$

$$O(kn) = kO(n) = O(n).$$

Quando a notação assintótica é utilizada sozinha no lado direito de uma equação como em $n = O(n)$, o sinal de $=$ não representa igualdade e sim que a função $f(n) = n \in O(n)$, sendo unidirecional, assim nunca se deve escrever coisas do tipo $O(n) = n$.

Em equações como a apresentada abaixo $=$ representa igualdade.

$$n^2 + n + 10 = n^2 + \Theta(n).$$

Na equação supracitada, a notação assintótica pode ser interpretada como uma função que não é importante definir precisamente.

$$n^2 + n + 10 = n^2 + f(n).$$

A função $f(n)$ pertence ao conjunto $\Theta(n)$. Nesse caso, $f(n) = n + 10$ que de fato pertence a $\Theta(n)$.

1.3 O básico das regras de cálculo de complexidade

Como mencionado, a partir de uma inspeção do algoritmo, antes de sua implementação é possível estimar sua complexidade e colocá-la em termos de $O(f(n))$. Algumas estruturas no algoritmo podem ter suas complexidades estimadas rapidamente, outros algoritmos requerem técnicas mais avançadas. O cálculo do número de operações é um problema de contagem, abaixo alguns exemplos simples utilizando os princípios fundamentais da contagem.

Deve-se sempre procurar expressar o custo assintótico de algoritmos da forma mais precisa possível. No caso da notação O, essa precisão envolve (1) expressar o limite superior firme do custo assintótico do algoritmo e (2) indicar o caso do algoritmo para o qual esse custo se aplica.

Exemplo

O algoritmo para encontrar um elemento k em um vetor não ordenado de tamanho n tem custo $O(n)$ no pior caso. É possível dizer que no pior caso o algoritmo é $O(n^2)$, no entanto, esse custo não define um limite superior firme, que é $O(n)$ e a função $f(n) = n$ que deve ser usada.

•

1.3.1 Constantes

Exemplo

Operações aritméticas e atribuições levam um custo constante e distinto para serem realizadas. Por exemplo no pseudocódigo abaixo, cada linha tem um custo constante k_0, k_1 e k_2 respectivamente. Frequentemente, consideramos todas com custo 1, mas na realidade isso é falso, elas possuem custos distintos.

```
int a:= 2
int b:= 7
int a:=a+b
```

A chamada **função de complexidade**, são os valores explícitos $f(n) = k_0 + k_1 + k_2 = k_3$ ou $f(n) = 3$, a **ordem de complexidade** é a ordem expressa em notação assintótica apenas com os termos dominantes, ignorando constantes e termos assintoticamente irrelevantes, $f(n) = O(1)$.

•

1.3.2 Laços de iteração

Exemplo 1

O algoritmo representado por pseudocódigo abaixo está somando elementos de um vetor V utilizando um laço de iteração `for`.

```
int soma(V, n)
{
    int s := 0
    for(i:=0; i<n; i++)
        s := s+V[i]

    return s
}
```

Primeiramente vamos estimar a complexidade do tempo de execução do algoritmo.

(1) Atribuir o valor 0 para s tem custo 1;

(2) No laço de iteração `for` $i := 0$ tem custo 1, $i++$ é feito n vezes e a comparação $i < n$, $n+1$ vezes, é possível então dizer que o `for` tem custo $2n+2$, no entanto o importante é a operação mais realizada, a de *comparação* $i < n$, com custo $n+1$;

(3) A *atribuição* $s := s+V[i]$, dentro do laço de repetição é no total realizada n vezes, portanto tem custo n.

Somando as complexidades em (1), (2) e (3) obtemos $g(n) = 2n+3$. E podemos dizer $g(n) = O(n)$.

A complexidade de espaço nesse algoritmo também pode ser estimada.

(1) As variáveis n, s e i tem custo 1, somando temos 3;

(2) V é um vetor que armazena n variáveis e tem custo n;

Somando as complexidades de (1) e (2), obtemos $S(n) = n+3$ e $S(n) = O(n)$.

•

Exemplo 2 (Laços aninhados)

Um caso extremamente comum são os **laços de iteração aninhados**.

```
for(i:=0; i<n; i++)
    for(j:=0; j<n; j++)
        //procedimento
```

Note que a cada iteração do primeiro laço, o segundo executa $n+1$ comparações, mas como o primeiro laço repete n iterações, obtemos $g(n) = n(n+1) = n^2+n$ e $g(n) = O(n^2)$.

Generalizando, sejam $i_0, i_1, \ldots, i_{k-1}$, k variáveis distribuídas entre k laços de iteração, onde $i_k < n$ e i_k++ para todo laço de iteração, ignorando outros procedimentos, podemos estimar a complexidade como $g(n) = O(n^k)$.

•

Exemplo 3

Nos laços de iteração aninhados abaixo, repare a comparação $j < i$.

```
for(i:=0; i<n; i++)
    for(j:=0; j<i; j++)
    //procedimento
```

(1) O primeiro laço com o iterador i realiza $n+1$ comparações;

(2) No segundo laço j varia de 0 a k a cada vez que $i := k$, sendo $0 \leq k \leq n$. Dessa forma, o procedimento é executado $0+1+2+3+\ldots+n = \frac{n(n+1)}{2}$ vezes.

Sendo assim, a complexidade do algoritmo é $O(n^2)$.

•

Exemplo 4 (Passos diferentes)

Repare no passo do laço de repetição abaixo $i := i+k$. Onde k é alguma constante

```
for ( i :=0;  i<n;  i := i+k )
    // procedimento
```

Se i é incrementado de k em k, o procedimento é executado $\frac{n}{k}$ vezes e o laço realiza $\frac{n}{k}+1$ comparações. A complexidade do algoritmo é $O(n)$.

•

Exemplo 5 (Condição de parada)

Há casos em que o laço de iteração não é tão evidente, podendo ser o caso de inspecionar a condição de parada, ou seja, a condição que deve ser satisfeita para o laço de iteração parar de executar as operações dentro dele. No exemplo abaixo a condição de parada é $j > n$.

```
int  j := 0
for ( i :=0;  j  <= n;  i++)
    j := j+i
```

Primeiramente, qual o comportamento da variável j a medida que i é incrementada ?

(1) O comportamento de j é descrito na tabela abaixo;

i	**j**
0	0
1	$0+1=1$
2	$0+1+2=3$
k	$0+1+...+k=\frac{k(k+1)}{2}$

(2) Vamos assumir que a condição de parada foi alcançada, ou seja $j > n$. Isso significa que para algum valor $i = k$, $j > n$ e como $j = \frac{k(k+1)}{2} > n$;

(3)

$$\frac{k(k+1)}{2} = \frac{k^2+k}{2} > n \Rightarrow k^2+k > 2n \Rightarrow k^2 > n \Rightarrow k > \sqrt{n}.$$

A complexidade desse algoritmo é $O(\sqrt{n})$.

•

Exemplo 6

```
for ( i :=0;  i<n;  i := i *2)
    // procedimento
```

(1) A variável i cresce em potências de base 2 e assume um valor 2^k;

(2) Assumindo que a condição de parada $i \geq n$ foi atingida, então $2^k \geq n$ e quando $2^k = n$, obtemos $k = \log_2(n)$;

(3) O procedimento é executado $\lceil \log_2(n) \rceil$ vezes.

A complexidade do algoritmo é $O(log(n))$.

•

Exemplo 7

```
for ( i :=0;  i*i<n;  i++)
    // procedimento
```

A condição de parada do laço de iteração é $i^2 \leq n$, o algoritmo tem complexidade $O(\sqrt{n})$.

•

1.3.3 Fases

Se um algoritmo possui diversas fases, a complexidade do tempo de execução será a da fase com maior complexidade.

Exemplo 1

No pseudocódigo abaixo, o laço com maior complexidade é o segundo, $O(n^2)$. A complexidade do algoritmo é $O(n^2)$.

```
for(i:= 0; i<n; i++)
    // procedimento

for(i:= 0; i<n; i++)
    for(j = 0; j<n; j++)
        //procedimento
```

•

Exemplo 2 (Dependência de fases)

No pseudocódigo abaixo, a variável p cria uma dependência entre os laços, e embora eles não estejam aninhados, a complexidade do procedimento executado no segundo laço depende da complexidade do primeiro.

```
int p:= 0
for(i:= 0; i<n; i=i*2)
    p++

for(i:= 0; i<p; i=i*2)
    //procedimento
```

(1) A variável p é incrementada $\lceil \log_2(n) \rceil$ vezes no primeiro laço de iteração;
(2) O procedimento do segundo laço de repetição é executado $\log_2(p) = \log_2(\log_2(n))$ vezes.
A complexidade nessa caso é $O(\log\log(n))$.

•

1.3.4 Várias variáveis

Em alguns casos a complexidade pode depender de mais de uma variável e sua ordem de complexidade estará em função dessas variáveis.

Exemplo

```
for(i:= 0; i<n; i++)
    for(j:= 0; m; j++)
        //procedimento
```

Nesse caso a complexidade é $O(nm)$.

•

1.3.5 Recursão

A complexidade do tempo de execução de funções recursivas depende do número de chamadas e da complexidade de cada chamada.

Exemplo 1

```
fatorial(n)
{
    if(n=0)
        return 1

    n*fatorial(n-1)
}
```

(1) Cada chamada faz uma comparação e tem complexidade $O(1)$ (constante);
(2) Dada uma entrada n a função faz $n-1$ chamadas recursivas;
A complexidade total é o produto da complexidade de cada chamada e da complexidade da quantidade de chamadas, dessa forma a complexidade é $O(n)$.

•

Exemplo 2

```
void funcao(n)
{
    if(n=1)
       return

    funcao(n-1)
    funcao(n-1)
}
```

(1) Cada função gera duas chamadas do mesmo tipo, as próximas duas fazem juntas quatro chamadas, as próximas quatro, oito e assim por diante;
(2) quando as chamadas chegam em `funcao(1)` foram feitas 2^{n-1} chamadas.
A complexidade é $O(2^n)$.

•

1.4 Classes de complexidade

Na análise de algoritmos existem classes de complexidade muito comuns. Do ponto de vista assintótico a seguinte hierarquia de funções pode ser definida

$$1 \prec \log(\log(n)) \prec log(n) \prec \sqrt[k]{n} \prec n \prec n\log(n) \prec n^k \prec n^{\log(n)} \prec k^n \prec n! \prec n^n.$$

Onde k é uma constante maior que zero.
A hierarquia de funções supracitadas indica que as funções mais a direita tem uma **ordem de crescimento** maior. Ou seja, quando $n \to \infty$, assumem valores mais significativos.

1.4.1 Lista de classes de complexidade

$O(1)$ Dizemos que a complexidade do tempo de execução é **constante**, geralmente quando é utilizada uma formula fechada, não dependendo do tamanho da entrada;
$O(log(n))$ A complexidade é dita **logarítmica**, geralmente o tamanho da entrada é dividido por um fator, um exemplo de algoritmo é a **busca binária**;
$O(\sqrt{n})$ É uma complexidade de **raiz quadrada** que fica aproximadamente no meio da entrada já que $\sqrt{n} = \frac{\sqrt{n}}{n}$;

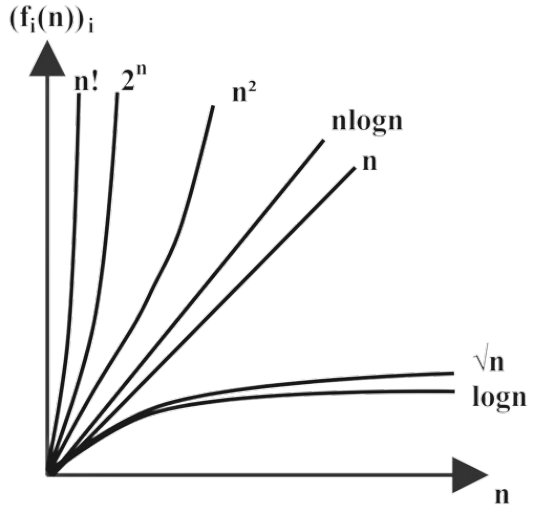

Figura 5: Esboço, algumas funções da hierarquia.

$O(n)$ Executa em tempo **linear**, geralmente é o melhor caso de complexidade possível de alcançar, já que muitas vezes é preciso acessar n posições pelo menos uma vez;

$O(nlog(n))$ A complexidade **nlog(n)** é um forte indício de que a entrada está sendo ordenada ou que há uma **estrutura de dados** com operações de complexidade em tempo de execução de $O(log(n))$;

$O(n^2)$ Um algoritmo com complexidade de tempo de execução **quadrática** geralmente aparece em laços de iteração aninhados;

$O(n^3)$ É dito um algoritmo **cúbico**;

$O(2^n)$ Em geral os casos k^n são chamados **exponenciais**, no entanto 2^n é um caso particular, pois é o número de subconjuntos de um conjunto discreto finito, o que pode revelar que o algoritmo itera por todos os subconjuntos da entrada;

$O(n!)$ Uma complexidade **fatorial** geralmente indica que o algoritmo itera por todas as permutações da entrada.

1.5 Pior caso, melhor caso e caso médio

Um mesmo algoritmo pode desempenhar diferentes complexidades dependendo da entrada. O melhor caso de uma busca em um vetor por exemplo, ocorre quando o primeiro elemento é o elemento procurado, e o pior caso quando o elemento não se encontra no vetor.

O algoritmo de ordenação *quick sort* por exemplo, apresenta seu caso médio com complexidade $O(n\log(n))$ e seu pior caso com complexidade $O(n^2)$.

O cálculo de complexidade do caso médio é bem mais complicado, Ele requer a probabilidade P da ocorrência de uma entrada. Na análise probabilista é necessário conhecer alguma informação sobre a distribuição das entradas.

Felizmente, na maioria da vezes, o pior caso do algoritmo é o de interesse.

Exemplo (Busca sequencial)

Considere um algoritmo que busca um valor c em um vetor V de tamanho n. Esse é um dos mais simples exemplos para exemplificar os três casos de complexidade. Considere que o vetor não está ordenado e que analisamos as posições em ordem crescente.

(1) No melhor dos casos, o elemento procurado c está na primeira posição, logo foi necessário apenas uma comparação e um custo constante; A função complexidade $T(n)$ será então $T(n) = 1 \in O(1)$.

(2) No caso médio algumas restrições devem ser tomadas; Primeiramente, considere que a chave c procurada está com certeza no vetor, ou seja, uma posição valida sempre será retornada pelo algoritmo;

Em segundo lugar, a probabilidade P de uma das posições alocar a chave c é uma situação equiprovável, portanto $\frac{1}{n}$.

Se a chave estiver em uma posição r, serão feitas r comparações até encontra-la. Portanto, seja k o numero de comparações feitas até a posição r, tem-se $r = k$. E seja a probabilidade da posição r, $P_k = \frac{1}{n}\ \forall k \in \mathbb{N}$, então a função complexidade é

$$T(n) = P_1 + 2P_2 + ... + nP_n = \frac{1}{n}(1 + ... + n) = \frac{1}{n}\sum_{k=1}^{n} k$$

$$T(n) = \frac{n(n+1)}{2n} = \frac{n+1}{2} \in O(n).$$

(3) No pior caso, todas as comparações são feitas, isso acontece se o elemento não estiver no vetor ou se ele assumir a ultima posição, sendo feita n comparações $T(n) = n$ e a função complexidade e $T(n) \in O(n)$ é a ordem de complexidade.

•

1.6 Classes de algoritmo e o algoritmo ótimo

Até então, foram avaliados algoritmos em particular. No entanto é possível avaliar uma **classe de algoritmos** para solucionar um problema algorítmico em específico como encontrar um determinado valor ou ordenar um arranjo; Essa análise e bem mais rara, em geral, o que é feito é determinar um limite inferior de complexidade para o problema, ou seja, encontrar a menor complexidade possível para resolver o problema dado. Por muitas vezes, avaliar esse limite é difícil e requer métodos como autômatos (oráculos) e heurísticas. O algoritmo que apresenta complexidade igual a da cota inferior é chamado **algoritmo ótimo**.

2 Relações de recorrência

Na sessão anterior, o pseudocódigo para o algoritmo fatorial recursivo foi apresentado. Através dele, é observável que a solução para $a_r = r!$ $(r \in \mathbb{N})$, tendo como **condição inicial** $a_0 = 1$, pode ser computada através do que é chamado de **relação de recorrência**, nesse caso $a_r = ra_{r-1}$. Em uma **função definida recursivamente**, dado uma condição inicial e uma descrição dos estágios subsequentes em função dos anteriores é possível avaliar estágios da função para um dado n. Se f for definida recursivamente, então $f(n)$ é única para qualquer inteiro positivo n.

A **solução de uma relação de recorrência** é uma expressão que de o valor da função em termos de seus argumentos e que não dependa de sub expressões. Não existe um método geral para solucionar uma relação de recorrência arbitrária. Existem classes de relações de recorrência onde técnicas de solução são conhecidas.

Por vezes, encontrar a solução da recorrência é difícil e é possível contentar-se apenas com uma boa estimativa assintótica.

2.1 Método da expansão de termos

O método da expansão de termos ou substituição é capaz de resolver recorrências simples, basta escrever os termos em função de seus antecessores e assim dar um palpite sobre a forma geral, analisar o caso base e então encontrar uma solução para a relação de recorrência.

Exemplo 1

Uma simples relação de recorrência, $a_n = a_{n-1} + 1$ e $a_0 = 0$ pode ter uma solução encontrada por substituição de alguns termos.

$$a_n = a_{n-1} + 1 \Rightarrow a_{n-1} = a_{n-2} + 1 \Rightarrow a_n = a_{n-2} + 2$$

$$a_n = a_{n-k} + k$$

Assumindo $k = n$ temos que $a_n = a_0 + n$ pois $n - k = 0$, como $a_0 = 0$, então $a_n = n$ é a solução da recorrência. Por último, a notação $T(n) = T(n-1) + 1$ e $T(0) = 0$ com solução $T(n) = n$ é frequentemente utilizada, é claro que $T(n) \in O(n)$.

•

Exemplo 2

Para encontrar a solução de $a_n = a_{n-1} + n + 1$ com $a_0 = 0$ também é possível reescrever a_n

$$a_n = a_{n-1} + n + 1 = [a_{n-2} + (n-1) + 1] + n + 1 = a_{n-2} + (n-1) + n + 2 = a_{n-3} + (n-2) + (n-1) + n + 3$$

$$a_n = a_{n-k} + (n-k+1) + ... + (n-1) + n + k$$

Para atingir a condição inicial tome $k = n$

$$a_n = a_0 + 1 + ... + (n-1) + 2n = \frac{n(n+1)}{2} + n \in O(n^2).$$

•

Exemplo 3

Utilizando a outra notação apresentada, tome a relação de recorrência $T(n) = 2T\left(\frac{n}{2}\right) + n$ com condição inicial $T(1) = k_0$, onde k_0 é uma constante real positiva.

Expandindo os termos

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2^2 T\left(\frac{n}{2^2}\right) + n + n = 2^3 T\left(\frac{n}{2^3}\right) + n + n + n.$$

A partir da expansão acima, o palpite $T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$ é apropriado.

O caso da condição inicial deve ser avaliado com atenção. A condição $T(1) = k_0$ não é atingida quando $k = n-1$ e sim quando $T\left(\frac{n}{2^k}\right) = T(1)$

$$T\left(\frac{n}{2^k}\right) = T(1) \Rightarrow \frac{n}{2^k} = 1 \Rightarrow k = \log_2 n.$$

Substituindo $k = \log_2 n$ no palpite, é obtido

$$T(n) = 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + n\log_2 n = nT(1) + n\log_2 n = k_0 n + n\log_2 n \in O(n\log n).$$

•

2.2 Generalização para relações de recorrência comuns

A seguir são apresentadas as soluções (sugestão de exercício) assintóticas para algumas relações de recorrência. A partir da observação, é possível conjecturar a respeito de uma forma generalizada de solução assintótica para algumas relações de recorrência comuns.

$$T(n) = T(n-1) + 1 \in O(n);$$

$$T(n) = T(n-1) + n \in O(n^2);$$

$$T(n) = T(n-1) + \log n \in O(n\log n);$$

$$T(n) = 2T(n-2) + 1 \in O(2^{\frac{n}{2}});$$

$$T(n) = 3T(n-1) + 1 \in O(3^n);$$

$$T(n) = 2T(n-1) + n \in O(n2^n).$$

Uma conjectura apropriada, que pode ser demonstrada como teorema é que para uma relação de recorrência da forma $T(n) = aT(n-b) + n^k$, com $a, b > 0$ e $k \geq 0$, existem três soluções assintóticas que podem ser sumarizadas:

(1) Se $a < 1$ então $T(n) \in O(n^k)$;

(2) Se $a = 1$ então $T(n) \in O(n^{k+1})$;

(3) Se $a > 1$ então $T(n) \in O(n^k a^{\frac{n}{b}})$.

2.3 Teorema mestre

Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função e $T(n)$ definida em $\mathbb{N}_0$ pela recorrência $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, $T(n)$ tem os seguintes limites assintóticos

(1) Se $f(n) \in O\left(\frac{n^{\log_b a}}{n^\varepsilon}\right)$ para alguma constante $\varepsilon > 0$, então $T(n) \in \Theta(n^{\log_b a})$;

(2) Se $f(n) \in \Theta(n^{\log_b a})$, então $T(n) \in \Theta(n^{\log_b a}\log n)$;

(3) Se $f(n) \in \Omega(n^{\log_b a} n^\varepsilon)$ para alguma constante $\varepsilon > 0$ e $\exists c;\ af\left(\frac{n}{b}\right) \leq cf(n)$ então $T(n) \in \Theta(f(n))$.

O Teorema mestre dá a solução assintótica para recorrências da forma $aT\left(\frac{n}{b}\right) + f(n)$. Muitas vezes saber a ordem de crescimento da recorrência já é suficiente e uma solução de fórmula fechada não é necessária.

De modo intuitivo, o teorema mestre que ao comparar $f(n)$ e $n^{\log_b a}$ o caso (1) ocorre se $f(n)$ for polinomialmente menor que $n^{\log_b a}$, no caso (2) se forem iguais e no caso (3) se $f(n)$ for polinomialmente maior que $n^{\log_b a}$. Como essa diferença por um fator polinomial n^ε é obrigatória existem classes de

funções $f(n)$ que são menores que $n^{\log_b a}$ mas não polinomialmente menores, dessa forma, o primeiro caso do teorema mestre não se aplicaria, isso é análogo para o terceiro caso.

Exemplo 1

$T(n) = 9T\left(\frac{n}{3}\right) + n$. Primeiramente, identificando os termos se obtém, $a = 9$, $b = 3$ e $f(n) = n$.

Em segundo lugar, avaliando $n^{\log_b a} = n^{\log_3 9} = n^2$. Tomando $\varepsilon = 1$, $f(n) = O(n^{2-1})$ e $f(n) = n \prec n^2$, o primeiro caso é satisfeito e $T(n) \in \Theta(n^2)$.

•

Exemplo 2

$T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$. Para essa recorrência, $a = 2$, $b = 4$ e $f(n) = \sqrt{n}$, ao avaliar $n^{\log_b a} = n^{\log_4 2} = \sqrt{n}$. Como $f(n) = \Theta(\sqrt{n})$ o segundo caso é satisfeito e $T(n) \in \Theta(\sqrt{n}\log n)$.

•

Exemplo 3 (Teorema não se aplica)

$T(n) = 6T\left(\frac{n}{6}\right) + n\log n$. Neste caso $n^{\log_b a} = n \prec n\log n$, entretanto, o terceiro caso não se aplica, observando que $f(n)$ não é polinomialmente maior que n, é apenas maior por um fator $\log n$.

•

Exemplo 4

$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$. Identificando os termos na recorrência, $n^{\log_4 3} = n^{0,793}$. Para uma constante $\varepsilon \approx 0,2$, $f(n) = \Omega(n^{0,793}n^{0,2})$. Verificando a condição de regularidade $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right)\log\left(\frac{3}{4}\right) \leq \left(\frac{3}{4}\right)n\log n \leq \frac{3}{4}f(n)$ logo $c = \frac{3}{4}$. Assim, o terceiro caso do teorema mestre pode ser aplicado e $T(n) \in \Theta(n\log n)$.

•

2.4 Algoritmos recursivos

Uma relação de recorrência é uma fórmula recursiva. Um procedimento recursivo na ciência da computação, chama a si próprio para resolver um problema similar porém menor, se a instância é pequena, ele a resolve diretamente como puder. Um exemplo de algoritmo recursivo é o de computar a soma dos números de Fibonacci até uma indexação n.

```
int fibonacci(n)
{
    if(n <= 1) return n

    else return fibonacci(n-1) + fibonacci(n-2)
}
```

Na sessão anterior, fora apresentado, com poucos detalhes, um algoritmo, sem muita utilidade, que fazia chamadas de forma a configurar uma complexidade exponencial $O(2^n)$. Observando o algoritmo recursivo de Fibonacci, é fácil notar que o tempo de uma chamada depende de outras duas e que `return` em `if` tem complexidade $k \in O(1)$, obtendo uma complexidade

$$T(n) = T(n-1) + T(n-2) + O(1).$$

A chamada `fibonacci(n-1)` faz mais chamadas subsequentes do que a chamada `fibonacci(n-2)`, entretanto, assumindo que as duas chamadas a mais são, por hora, irrelevantes é possível dizer que $T(n-1) \approx T(n-2)$ e encontrar uma cota assintótica superior pelo método da substituição. Os casos base $T(0)$ e $T(1)$ tem complexidade de tempo de execução constante.

$$T(n-1) \approx T(n-2) \Rightarrow T(n) = 2T(n-1) + k = 2[2T(n-2) + k] + k = 4T(n-2) + 3k$$

$$T(n) = 2^r[T(n-r)] + (2^r - 1)k.$$

Fazendo em $T(0) = T(1) = 1$, o termo $T(n-r) = T(0)$ leva a $n = r$. $T(n) = 2^n + (2^n - 1)k \in O(2^n)$. Logo o limite assintótico da cota superior é exponencial.

Entretanto, algo importante deve ser dito, aproximar as duas chamadas do algoritmo recursivo nos entrega, no caso anterior, a cota superior da complexidade de tempo de execução do algoritmo, com efeito, é errado dizer que a complexidade de tal algoritmo recursivo está em $\Theta(2^n)$. A complexidade real do algoritmo recursivo de Fibonacci é na verdade $O(\phi^n) \approx O(1,62^n)$.

Um fato interessante é que a relação de recorrência encontrada no momento do cálculo de complexidade $T(n) = T(n-1) + T(n-2) + O(1)$ se assemelha bastante a relação de recorrência da própria função de Fibonacci $a_n = a_{n-1} + a_{n-2}$. Em uma próxima sub sessão será apresentado um método capaz de encontrar uma solução $g(n)$ para o termo geral da série de Fibonacci que tem o termo mais significativo $\left(\frac{1+\sqrt{5}}{2}\right)^n < 2^n$. Sabendo que se $a < b$ então $a^n \in O(b^n)$ então é possível dizer $a_n \in O(1,62^n) \subset O(2^n)$.

2.4.1 Método da árvore de recursão para resolução de relações de recorrências

A complexidade do algoritmo recursivo de Fibonacci pode ter uma solução assintótica a partir do **método da árvore de recursão**. O método consiste em atribuir a cada nó um custo de um subproblema dentro do conjunto de chamadas da função recursiva e pode ser aplicado quando for difícil um palpite sobre a forma no método da substituição por exemplo.

Na árvore de recursão para o algoritmo recursivo de Fibonacci, o custo de uma função dependera das próximas chamadas e observamos o crescimento de uma árvore binária. A figura 6 ilustra um diagrama da árvore de recursão. Note que a quantidade de chamadas aproxima-se de 2^n, encontrando uma cota superior para a complexidade.

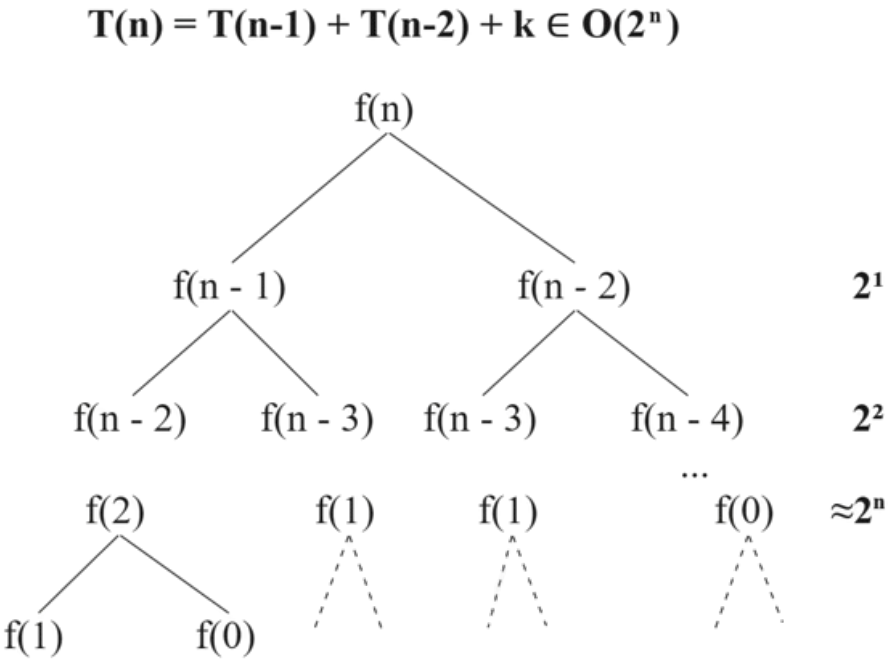

Figura 6: Árvore de recursão para o algoritmo recursivo de Fibonacci.

2.5 Divisão e conquista

Relembrando alguns tópicos, uma relação de recorrência é recursiva. Um algoritmo com procedimento recursivo resolve uma instância pequena como pode, reduz instância grandes a menores e após resolvidas retorna as originais, essa característica de trabalhar para trás é muito comum.

A **divisão e conquista** é uma estratégia para desenvolver algoritmos, com uma definição similar a apresentada para algoritmos recursivos. A técnica de divisão e conquista, frequentemente mais eficiente que métodos de força bruta para a resolução de problemas algorítmicos consiste em algumas etapas.

Primeiramente, o problema geral é **dividido** em problemas não sobrepostos de tamanhos aproximadamente iguais, esses subproblemas são instâncias menores do problemas geral. Em segundo lugar, ocorre a etapa de **conquista**, que consiste em solucionar recursivamente os subproblemas. Por último, os subproblemas são **combinados** solucionando o problema geral.

Dentre os algoritmos que seguem essa estratégia estão o `algoritmo numérico da bisseção`, os algoritmos de ordenação `quick sort` e `merge sort` onde a estratégia é bastante explicita e o algoritmo para multiplicação de matrizes de `Strassen` que reduz a complexidade do algoritmo ingenuo.

Exemplo (Busca binária)

A busca binária é um algoritmo para encontrar um valor em um vetor ordenado. O algoritmo consiste em buscar sempre o valor no meio do vetor, como este está ordenado, caso o valor procurado seja menor que o palpite, o algoritmo reduz a busca apenas a metade a esquerda do valor, o mesmo ocorre se o valor for maior, mas com a metade a direita. O problema então tem seu tamanho dividido, aproximadamente, pela metade do tamanho original sucessivamente, a complexidade do tempo de execução da busca binária, `binary_search`, está em $O(\log n)$.

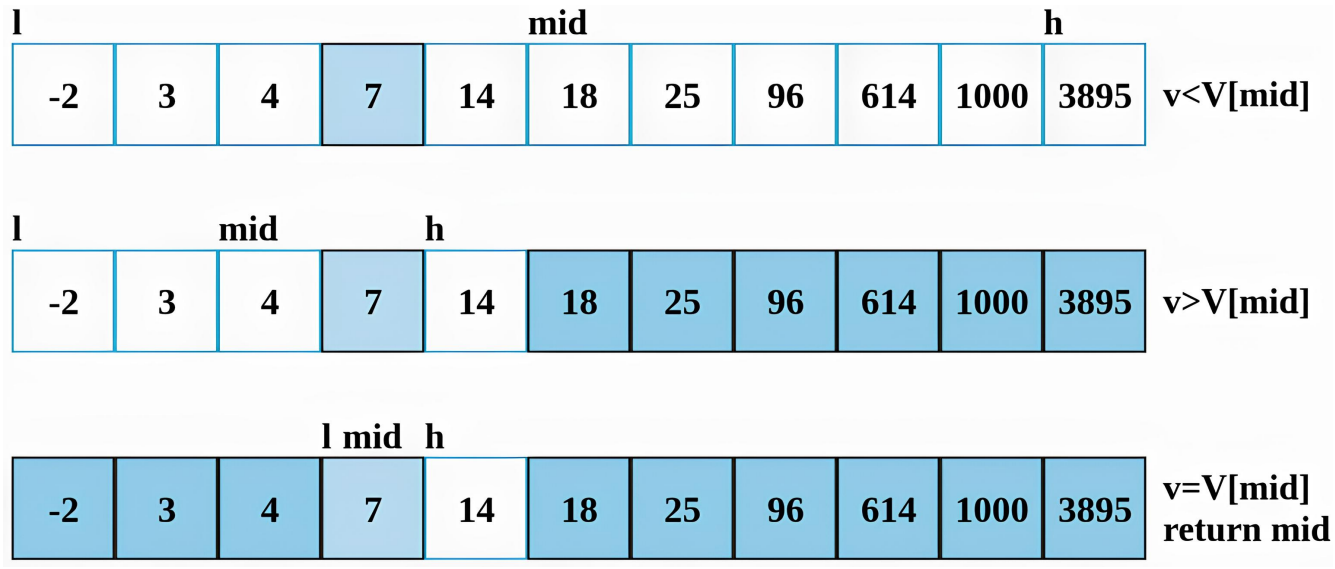

Figura 7: Diagrama de representação da busca binária, os elementos em azul são os eliminados do vetor a medida que o algoritmo é executado, o algoritmo busca pela chave de valor sete também marcada em azul.

No pseudocódigo abaixo, `v` é o valor procurado, `h` e `l` indicam as extremidades do vetor, que será dividido em sub vetores.

```
int binary_search(l,h,v,V)
{
    if(h==l)
    {
        if(V[l]==v) return v
        else return (-1)
    }

    else
    {
        mid = (l+h)/2
        if(v==V[mid]) return mid
        if(v<V[mid]) return binary_search(l, mid-1, v)
        else return(mid+1, h, v)
    }
}
```

Chamando de $T(n)$ a complexidade de tempo de execução da busca binária. O caso base da recursão está em $O(1)$ e as chamadas de subproblemas são $T\left(\frac{n}{2}\right)$. A solução assintótica da relação de recorrência $T(n) = T\left(\frac{n}{2}\right) + 1$ pode ser obtida a partir do **teorema mestre**.

Tem-se que $a = 1$, $b = 2$, $f(n) = n^0 = 1$ e $n^{\log_2 1} = n^0$. Tendo em vista que $f(n) = n^0 = 1 \in \Theta(1) = \Theta(n^{\log_2 1})$, o segundo caso do teorema mestre se aplica e $T(n) \in \Theta(1 \log n)$ portanto

$$T(n) \in (\log n).$$

2.6 Relações de recorrência homogêneas e não homogêneas

O método da substituição pode ser difícil para algumas relações de recorrência e o método mestre é restrito para formatos de relações. Ainda existem classes que podem ser solucionadas de outras formas, similares a encontrar soluções para equações diferenciais simples.

Através dos métodos a seguir é possível solucionar recorrências como as somas de quadrados ou até mesmo a famosa sequência de Fibonacci.

2.6.1 Princípio da superposição

Se funções $g_i(n)(i = 1, ..., k)$ forem soluções para uma relação de recorrência linear com coeficientes constantes de ordem r, então a combinação linear das k soluções, $A_1 g_1(n) + ... + A_k g_k(n)$, é solução da relação de recorrência, $A_i \in \mathbb{R}$.

2.6.2 Relações de recorrência homogêneas

Uma **relação de recorrência linear com coeficientes constantes de ordem r** é definida como uma relação de recorrência da forma $a_n = k_1 a_{n-1} + ... + k_r a_{n-r} + f(n)$ sendo k_i $(i = 1, ..., r)$ constantes e $f(n) = 0$. Se $g(n)$ é uma função onde $a_n = g(n)$ então $g(n)$ é chamada de **solução** da relação de recorrência.

Digamos que $a_r = x^r$ é solução da relação, então $x^n = k_1 x^{n-1} + ... + k_r x^{n-r}$, e ignorando a solução trivial $x = 0$, obtém-se a equação polinomial de grau r, $x^n - k_1 x^{n-1} - ... - k_r x^{n-r} = 0$, chamada de **equação característica** da relação de recorrência.

Se as raízes da equação características forem todas **reais e distintas**, a solução homogênea será da forma

$$a_n = A_1(x_1)^n + ... + A_n(x_n)^n.$$

Exemplo 1

$T(n) = 9T(n-2);\ T(0) = 6,\ T(1) = 12$. Obtendo a equação característica, $x^2 - 9 = 0$, as raízes são avaliadas em $x_i = -3, 3$. A solução da relação de recorrência é da forma $T(n) = A_1(-3)^n + A_2(3)^n$. Dadas as condições iniciais, $A_1 + A_2 = 6$ e $-3A_1 + 3A_2 = 12$, assim a solução é

$$T(n) = (-3)^n + 5(3)^n.$$

Exemplo 2 (Termo geral da sequência de Fibonacci)

A sequência de Fibonacci com relação de recorrência $a_n = a_{n-1} + a_{n-2}$ com $a_0 = a_1 = 1$, pode ter uma solução obtida através do método apresentado.

As raízes da equação característica $x^2 - x - 1 = 0$ são $x_1 = \frac{1+\sqrt{5}}{2}$ e $x_2 = \frac{1-\sqrt{5}}{2}$.

Usando a forma $a_n = A_1(x_1)^n + A_2(x_2)^n$, e as condições iniciais, avalia-se $A_1 = \frac{1+\sqrt{5}}{2\sqrt{5}}$ e $A_2 = \frac{\sqrt{5}-1}{2\sqrt{5}}$.

Dessa forma, multiplicando os termos, a solução para a relação de recorrência de Fibonacci é

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

•

Caso as raízes apresentarem **multiplicidade**, sendo reais, ou seja, **reais e repetidas**, o formato de solução para raízes reais e distintas não funcionará mais pois ele violaria o princípio da superposição da combinação linear.

Se x for uma raiz de multiplicidade então a solução particular será da forma

$$u_p = (x)^n(A_1 + ... + A_k x^{k-1}).$$

Exemplo 3

Se $x_1 = 2$ tiver multiplicidade 3 e $x_2 = 6$ multiplicidade 2 e ambas forem raízes de uma mesma equação característica então o formato da solução será

$$a_n = 2^n(A_1 + A_2 n + A_3 n^2) + 6^n(B_1 n + B_2 n^2).$$

•

2.6.3 Relações de recorrência não homogêneas

Quando uma relação de recorrência é do formato $a_n = h_n + f(n)$ com $f(n) \neq 0$, dizemos que h_n é a **parte homogênea** da relação de recorrência não homogênea. A solução geral será a soma da solução da parte homogênea com a solução não homogênea.

Existem dois casos especiais onde técnicas de solução são conhecidas. Primeiramente, o caso onde $f(n) = k(q)^n$, onde $q \neq 1$ é um número racional e k uma constante conhecida. A escolha para uma solução particular é $u_n = A(q)^n$, a menos que a q seja uma raiz da equação característica, nesse caso a escolha seria $u_n = A(n)^r(q)^n$, onde r é a multiplicidade da raiz.

O segundo caso é quando $f(n) = k(n)^r$, se a equação característica não possuir raiz $x = 1$, a solução é do formato $A_1 + ... + A_k(n)^{k-1}$. Se 1 for uma raiz de multiplicidade r então a escolha será $A_1 n^t + ... + A_k n^{t+(k-1)}$.

Exemplo (Soma de quadrados)

Avaliar a soma dos quadrados dos n primeiros inteiros positivos.

A relação de recorrência é dada por $a_n = a_{n-1} + n^2$ com $a_0 = 0$. A solução homogênea é calculada como $x = 1$, de multiplicidade 1, então a solução será da última forma apresentada.

A escolha para a solução particular não homogênea é $A_1 n + A_2 n^2 + A_3 n^3$. Como $a_0 = 0$ a solução é da forma.

$$a_n = A_1 n + A_2 n^2 + A_3 n^3,$$

$$a_n = A_1(n-1) + A_2(n-1)^2 + A_3(n-1)^3 + n^2.$$

Colando os termos n e n^2 em evidencia, um sistema pode ser formado para calcular $A_1 = \frac{1}{6}$, $A_2 = \frac{1}{2}$ e $A_3 = \frac{1}{3}$.

Dessa forma a solução é $a_n = \frac{n}{6} + \frac{n^2}{2} + \frac{n^3}{3}$, chegando no resultado

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

•

3 Teoria da Ordenação

O **problema da ordenação** é recorrente na ciência da computação, o motivo para isso é bem simples: Muitos problemas algorítmicos são mais fáceis de serem resolvidos quando os dados estão distribuídos de forma ordenada.

Nas próximas sub sessões serão apresentados alguns algoritmos de ordenação, inicialmente serão apresentados algoritmos ingênuos com complexidade polinomial, logo após, algoritmos de ordenação eficientes e por fim algoritmos com complexidade de tempo de execução linear. Para fins de simplificação, os algoritmos serão apresentados para ordenar vetores de números inteiros, no entanto, qualquer tipo de dado pode ser ordenado por um parâmetro.

3.1 Ordenação por bolha

O algoritmo de ordenação por bolha ou ordenação por flutuação, ou ainda, do inglês *bubble sort*, é um algoritmo de ordenação de complexidade de tempo de execução polinomial. O *bubble sort* funciona (1) **comparando** um elemento com o imediatamente seguinte e (2) **trocando-os** caso os elementos adjacentes estejam desordenados. Esse algoritmo repete tais ações para sub vetores até que o vetor esteja ordenado.

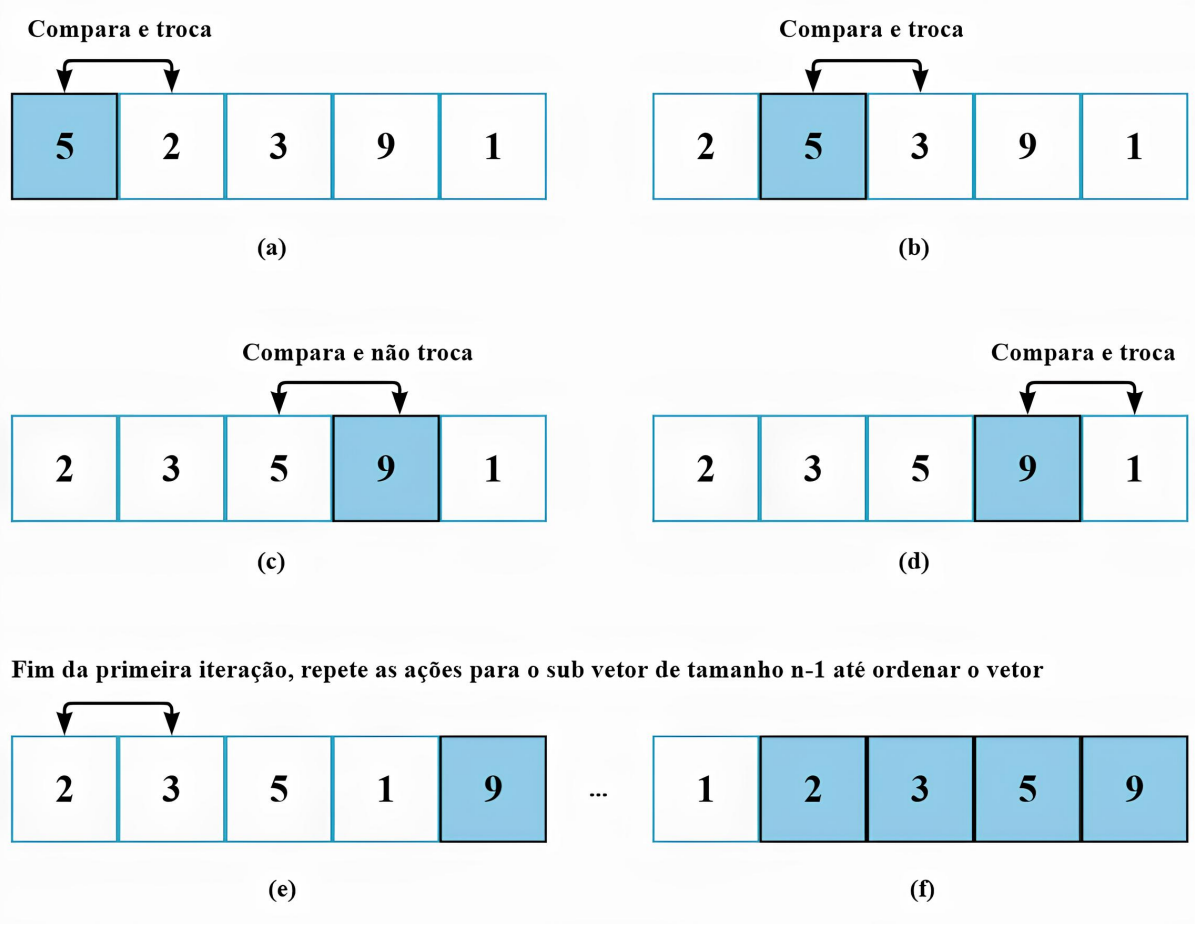

Figura 8: Ordenação por bolha atuando em um vetor cujo os elementos são números inteiros; Os elementos em azul são os maiores a cada iteração ou chamada do algoritmo e são postos no final do sub vetor e por fim ordenando o vetor por completo.

É perceptível que as operações mais feitas no algoritmo em questão são as **comparações** e as **trocas**. O código em C a seguir descreve as instruções para algoritmo `bubble_sort` que ordena um vetor de

tamanho n indexado de 0 a $n-1$ cujo os elementos pertencem à $\mathbb{Z}$. A função `swap` é responsável por trocar os elementos de posição, a flag `trocou` tem a funcionalidade de verificar se alguma troca foi efetuada, ela será útil pois, caso não seja realizadas trocas o vetor já está ordenado, no entanto as iterações ou chamadas podem ainda não ter chegado ao fim. Na melhor das hipóteses, caso o vetor já esteja ordenado basta uma iteração, pois a flag `trocou` dirá que nenhuma troca foi efetuada.

```
void bubble_sort(int *V, int n)
{
    for(int i=0; i<n-1; i++)
    {
        int trocou = 0;
        for(int k=0; k<n-1-i; k++)
        {
            if(*(V+k)>*(V+k+1))
            {
                swap(V+k, V+k+1);
                trocou=1;
            }
        }
        if(trocou==0) break;
    }
}
```

O código supracitado é uma versão iterativa do algoritmo que utiliza a flag de verificação de troca, também é possível implementar uma versão recursiva baseada em chamadas de sub vetores.

```
void bubble_sort_recursive(int *V, int n)
{
    if(n<=1) return;
    int trocou = 0;
    for(int i=0; i<n-1; i++)
    {
        if(*(V+i)>*(V+i+1))
            {
                swap(V+i, V+i+1);
                trocou=1;
            }
    }
    if(trocou==0) return;

    bubble_sort_recursive(V, n-1);
}
```

3.1.1 Análise de complexidade

A intuição por trás da análise de complexidade da ordenação por bolha é pensar que, no **pior caso**, todas as operações possíveis são realizadas, para que isso aconteça o vetor deve estar em ordem decrescente caso queiramos uma ordenação crescente.

Considere a versão não recursiva, é visível que

(1) O primeiro laço de iteração `for` realiza seu procedimento interno $n-1$ vezes;

(2) O segundo laço tem um comportamento baseado no iterador `i`, onde seu procedimento interno é executado $n-1-i$ vezes, tem-se então

$$(n-1-0)+(n-1-1)+(n-1-2)+...+1=\frac{(n-1)n}{2}\in O(n^2).$$

A análise acima é suficiente para demonstrar a cota superior da complexidade do tempo de execução. Se os índices do vetor forem pensados como elementos pertencentes à $\mathbb{N}$ então é possível escrever

$$\sum_{i=1}^{n-1}\sum_{k=1}^{i-1}\kappa.$$

Onde κ é uma constante que representa o custo da operação de troca. No **pior caso** essa constante é somada a cada iteração, já que a operação de troca é sempre feita.

$$\sum_{i=1}^{n-1}\sum_{k=1}^{i-1}\kappa=\sum_{i=1}^{n-1}\kappa(i-1)=\kappa[1+2+...+(n-2)+(n-1)]=\frac{\kappa(n-1)n}{2}.$$

Esse é o resultado obtido na análise intuitiva, agora basta provar que a progressão aritmética encontrada está em $O(n^2)$. Lembrando que $O(kg(n))=kO(g(n))=O(g(n))$, para demonstrar que $\frac{\kappa(n-1)n}{2}\in O(n^2)$, basta encontrar constantes n_0 , $k>0$ tais que $\frac{(n-1)n}{2}\leq kn^2\ \forall n\geq n_0$.

$$\frac{(n-1)n}{2}=\frac{n^2-n}{2}\leq kn^2\Rightarrow\Big(\frac{1}{2}-k\Big)n^2-\frac{n}{2}\leq 0\ \forall n\geq n_0.$$

Para satisfazer a inequação acima, basta escolher $k=\frac{1}{2}$ e $n_0=1$, substituindo

$$\Big(\frac{1}{2}-\frac{1}{2}\Big)-\frac{1}{2}\leq 0\Rightarrow-\frac{1}{2}\leq 0;\ \therefore\ \frac{\kappa(n-1)n}{2}\in O(n^2).$$

■

A análise intuitiva do **melhor caso** é bem simples, quando a flag `trocou` não receber o valor 1 significa que o vetor está ordenado. Se o vetor já estiver ordenado, apenas $n-1$ iterações no laço exterior são realizadas e ele termina a execução; $n-1\in O(n)$, então a complexidade do tempo de execução do melhor caso é linear.

Considere no melhor caso que atribuir valor a flag tenha um custo constante $O(1)$ e que são realizadas $n-1$ comparações, mas nenhuma chamada de `swap`, portanto, nenhuma troca. Dessa forma, a flag será 0 e a comparação final devolvera um retorno para o método, portanto a função complexidade é $f(n)=n-1+O(1)$, considerando que $\exists k>0;\ n-1\leq kn$, então $f(n)\in O(n)$. Sendo assim, a complexidade do tempo de execução no melhor caso para `bubble_sort` está em $O(n)$.

■

Em penúltimo lugar, a análise de complexidade do **caso médio** para o tempo de execução do algoritmo de ordenação por bolha deve receber algumas considerações. Em primeiro lugar, considere que em metade das comparações são realizadas trocas, isso pode ser justificado se atribuirmos 0 "uma comparação foi executada mas uma troca não foi executada" e 1 "uma comparação foi executada e uma troca foi executada"; Assim sendo, o valor esperado $E[x]$ é igual a $\frac{1}{2}$. Obtem-se então

$$\sum_{i=1}^{n-1}\sum_{k=1}^{i-1}\frac{1}{2}\kappa=\frac{1}{2}\sum_{i=1}^{n-1}\kappa(i-1)=\frac{\kappa(n-1)n}{4}$$

de $\frac{(n-1)n}{2}\in O(n^2)$ e $kO(f(n))=O(f(n))$ encontra-se o limite assintótico $\frac{\kappa(n-1)n}{4}\in O(n^2)$ que é a complexidade do tempo de execução do caso médio do algoritmo em questão.

■

Por fim, mas não menos importante, para avaliar o **complexidade de espaço** requerido para a ordenação por bolha não é considerado o espaço onde o vetor a ser ordenado está armazenado, pois o algoritmo em si não o armazena e nem cria cópias para manipular os elementos do vetor a ser ordenado.

No algoritmo para os iteradores `i` e `k` e para a flag `trocou` considere um custo constante σ, assim a função $s(n)$ complexidade de espaço é $s(n) = 3\sigma \in O(1)$.

3.1.2 Estabilidade

Dado um vetor com elementos repetidos, a ordem relativa entre esses elementos é definida pela posição inicial que aparecem no vetor. Considere o vetor $[2,2,1,1,3]$; Os elementos repetidos $2's$ e $1's$ são idênticos, mas começam com uma posição relativa, é possível então, diferencia-los como $[2_a, 2_b, 1_a, 1_b, 3]$. Um algoritmo de ordenação baseado em comparação pode ser considerado estável quando mantém a ordem relativa das posições iniciais entre elementos idênticos de um vetor.

No vetor supracitado o algoritmo de ordenação por bolha trabalharia da seguinte forma:

$$[2_a, 2_b, 1_a, 1_b, 3] \to [2_a, 2_b, 1_a, 1_b, 3] \to [2_a, 1_a, 2_b, 1_b, 3] \to [2_a, 1_a, 1_b, 2_b, 3] \to ... \to [1_a, 1_b, 2_a, 2_b, 3].$$

Observe que, na ordenação acima, não houve uma mudança na posição relativa dos elementos idênticos e que o vetor foi ordenado. O *bubble sort* é um algoritmo de ordenação baseado em comparação estável.

De maneira mais formal, (1) considere que para uma lista de apenas um elemento, não há necessidade de ordenação pois a mesma já se encontra ordenada; (2) suponha que o algoritmo *bubble sort* mantenha a posição relativa em um vetor de tamanho n; (3) Considere um vetor de tamanho $n+1$. Durante a passagem do algoritmo pelos elementos, uma troca só ocorrerá caso o elemento na posição k seja menor que o elemento da posição $k+1$. No entanto, se os elementos possuírem o mesmo valor, isso significa que permanecerão na mesma ordem relativa em relação uns aos outros, portanto, o algoritmo de ordenação por bolha é **estável**.

■

3.1.3 Sumário ordenação por bolha

O algoritmo de ordenação por bolha é um algoritmo com complexidade de tempo de execução quadrática na maioria das vezes, ademais, é um algoritmo estável, simples de entender e que funciona bem para um conjunto pequeno de dados.

Algoritmo	Melhor caso	Caso médio	Pior caso	Espaço	Estabilidade
Ordenação por bolha	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Estável

Tabela 1: Complexidade e estabilidade do algoritmo de ordenação por bolha

3.2 Ordenação por seleção

O algoritmo de ordenação por seleção, *selection sort*, é um algoritmo de complexidade de tempo de execução de ordem polinomial. A ordenação por seleção funciona (1) A partir da posição k ($k = 1, ..., n-1$) encontra o menor a partir de **comparações** e (2) **troca** o menor encontrado com o elemento da posição k. O algoritmo realiza repetidamente tais instruções a cada sub vetor até que o vetor esteja ordenado.

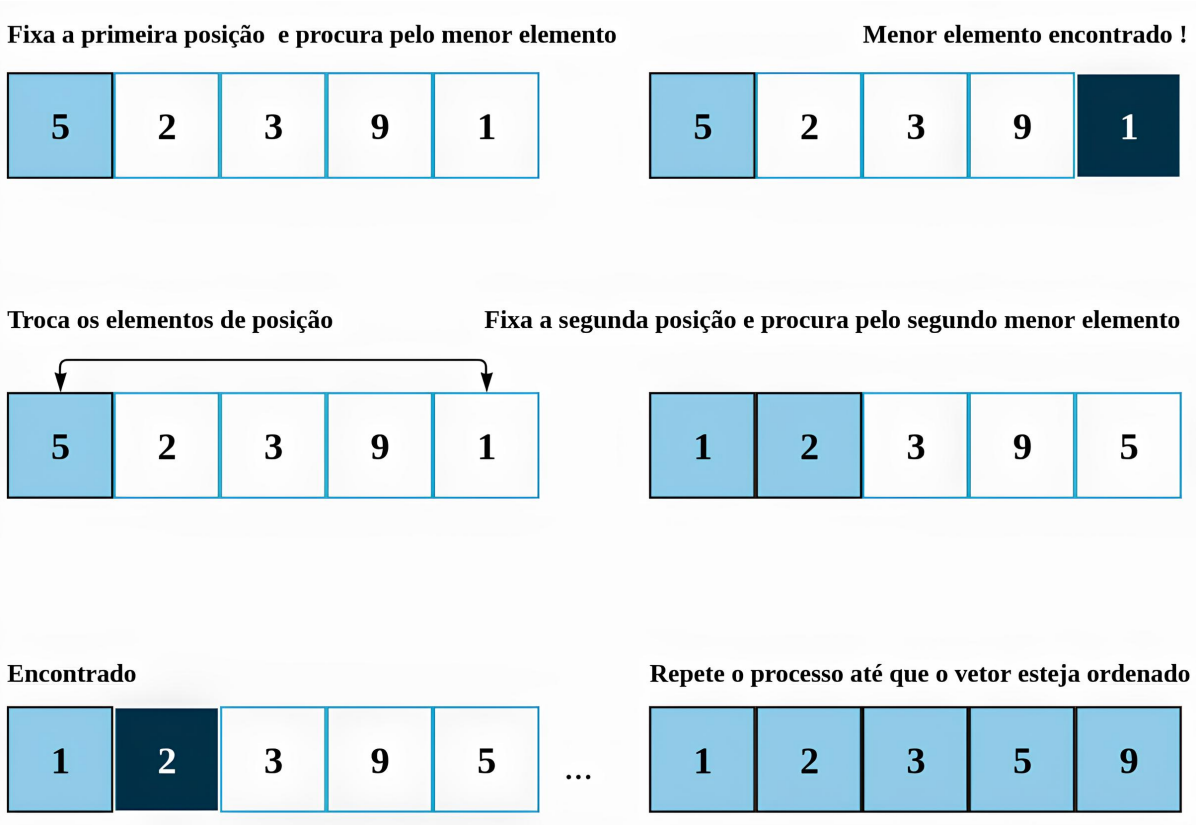

Figura 9: Ordenação por seleção atuando em um vetor cujo os elementos são números inteiros; Os elementos em azul são as posições iniciais de cada sub vetor e os elementos já ordenados; Os elementos em azul escuro são os menores encontrados a cada sub vetor iterado.

O código em C abaixo implementa o algoritmo de ordenação por seleção no método `selection_sort` que ordena um vetor de forma crescente cujo os elementos são pertencentes à $\mathbb{Z}$. A variável `min` armazena o índice do elemento mínimo a cada iteração em um sub vetor.

```
void selection_sort(int* V, int n)
{
    for(int i=0;i<n-1;i++)
    {
        int min = i;
        for(int k=i+1;k<n;k++)
        {
            if(*(V+min)>*(V+k))
            {
                min = k;
            }
```

```
            }
            swap(V+i, V+min);
        }
    }
```

3.2.1 Análise de complexidade

3.2.2 Estabilidade

O algoritmo de ordenação por seleção não é estável. Para demonstrar tal afirmação basta encontrar um vetor que ao ser operado pelo *selection sort* não mantém a posição relativa de elementos idênticos.

Considere o vetor $[1_a, 2_a, 2_b, 1_b]$; O algoritmo em questão executara os seguintes passos no vetor

$$[1_a, 2_a, 2_b, 1_b] \rightarrow [1_a, 2_a, 2_b, 1_b] \rightarrow [1_a, 1_b, 2_b, 2_a].$$

Observe que o vetor foi ordenado, no entanto, a posição relativa entre os $2's$ não foi mantida. Portanto o algoritmo de ordenação por seleção é **instável**.

■

3.2.3 Sumário ordenação por seleção

Com base no que foi apresentado, o algoritmo de ordenação por seleção é um algoritmo de ordenação baseado em comparações com complexidade sempre quadrática, é um algoritmo não estável e de fácil compreensão.

Algoritmo	Melhor caso	Caso médio	Pior caso	Espaço	Estabilidade
Ordenação por seleção	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Instável

Tabela 2: Complexidade e estabilidade do algoritmo de ordenação por seleção

3.3 Ordenação por inserção

O algoritmo de ordenação por inserção, do inglês *insertion sort*, é um algoritmo projetado para listas encadeadas. Por outro lado, ele pode ser aplicado a vetores, desde que um laço de iteração desloque elementos como será apresentado.

A essência por trás do algoritmo de ordenação por inserção é encontrar um local para inserir um elemento do vetor. Suas instruções podem ser descritas como (1) um elemento na posição $k \in \mathbb{N}$ é selecionado (2) seu valor é comparado com todos os elementos anteriores até que se encontre um elemento na posição $r < k$ que seja menor que o elemento selecionado (3) no caso de um vetor todos os elementos posteriores a posição r e anteriores a posição k são deslocados para uma posição a frente e (4) por fim, o elemento selecionado é adicionado na posição $r+1$.

No caso de uma lista encadeada, ao ser encontrado o local para inserir o elemento, apenas se deve redirecionar os apontadores para alocar o elemento entre outros dois outrora adjacentes.

O seguinte código implementa o algoritmo de ordenação por inserção para um vetor cujo os elementos pertencem a $\mathbb{Z}$. O laço `while` é responsável pelo deslocamento dos elementos do vetor afim de "abrir"um espaço para a chave a ser inserida. Por último, é importante observar que `r` deve ser maior que -1 para que o `while` execute, afinal, uma posição indesejada seria acessada a qual o conteúdo é desconhecido, isso também poderia ser contornado por uma **sentinela**.

```
void insertion_sort(int* V, int n)
{
    for(int k=1; k<n; k++)
    {
        int x = *(V+k); //elemento ou chave
        r = k-1;
        while(r>-1 && *(V+r)>x)
        {
            *(V+r+1)=*(V+r);
            r--;
        }
        *(V+r+1)=x;
    }
}
```

3.4 Ordenação por intercalação (merge sort)

Anexo I - Revisitando a linguagem de programação C

Apontadores

Para acessar o endereço de memória virtual de uma variável em C é utilizado o operador de referência &, o endereço de memória é representado por 0x seguido de um número em base hexadecimal.

Um apontador é um tipo de variável cuja função é armazenar um endereço de memória virtual, se um apontador p armazena o endereço de uma variável x é possível então dizer que p aponta para o endereço de x.

O processo de definir um apontador para um endereço significa que uma **referência** para o endereço é feita, já armazenar o identificador do endereço em uma variável é um processo de **dereferência**.

```
int x = 35;
int* p = &x; //Referencia

int d = *p; //Dereferencia
```

Aritmética de endereços

A aritmética de endereços é importante para a alocação dinâmica. Na memória virtual, os endereços dos elementos de um vetor são armazenados de modo sequencial. Ao declarar um vetor V, sendo i uma constante $i = 0, 1...$ *tamanho do vetor* -1, acessar $V[i]$ é idêntico a acessar $*(V+i)$.

```
int* V;
V = malloc (10*sizeof(int));
//V[i]=*(V+i)
```

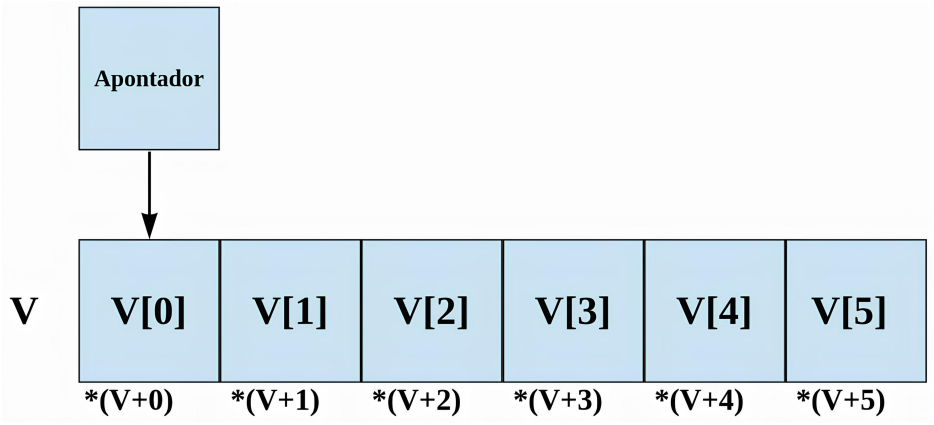

Figura 10: Representação da aritmética de endereços

Para o caso de matrizes $M[i][j] = *(*(M+i)+j)$.

Alocação dinâmica

Diferentemente da alocação estática, a alocação dinâmica permite que a memória utilizada durante a execução do programa seja administrada através de funções.

A função `malloc` é utilizada para alocar um bloco de memória na *heap* onde o argumento é o número de *bytes* alocados, `sizeof` é um operador que indica quantos *bytes* tem o parâmetro.

```
void* malloc(size_t size)
```

A função `calloc` aloca o bloco de memória solicitado retornando um apontador para ele, a diferença entre as funções `malloc` e `calloc` é que na segunda a memória alocada é definida para zero.

```
void* calloc(size_t nitems, size_t size)
```

Para redimensionar a memória já alocada é utilizada a função `realloc`, essa função traz uma dinamicidade maior para a memória. Os argumentos necessários são um apontador para o bloco previamente alocado e o novo tamanho.

```
void* realloc(void* ptr, size_t size)
```

As variáveis armazenadas estaticamente são excluídas no final da função correspondente, no entanto, para a memória dinamicamente alocada é necessário o uso da função `free` é necessário. Essa função desaloca um bloco de memória previamente alocado.

```
void free(void* ptr)
```

Vetores e matrizes dinâmicos

É possível, através de apontadores alocar vetores e matrizes. Uma matriz basicamente pode ser interpretada como apontadores duplos, os códigos abaixo demonstram como fazer essa alocação de memória e como libera-la com `free`.

```
//alocando vetor dinamico
int* V;

V = malloc (n*sizeof(int));

for(int i = 0; i < n; i++)
{
    //procedimentos
}

//desalocando vetor dinamico
free (V);
```

```
//alocando matriz dinamica
int **M;
M = malloc (m*sizeof(int*));

for (int i = 0; i < linhas; i++)
{
   *(M+i) = malloc(colunas*sizeof(int));
}

//desalocando matriz dinamica
for (int i = 0; i < linhas; i++) {
    free(*(M+i));
}
```