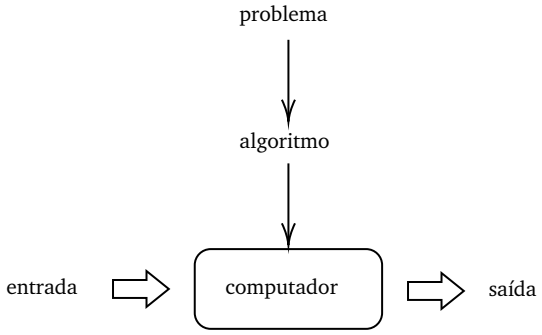
Contents

Semana 1

Introdução ao projeto e à análise de algoritmos

1.1 O que é um algoritmo?



Em um **problema** computacional temos um conjunto de **entradas** válidas, e para cada entrada válida temos uma ou mais possíveis **saídas** esperadas.

Um **algoritmo** é uma sequência de instruções não ambíguas para resolver um problema.

Resolver o problema significa fornecer uma saída esperada para qualquer entrada válida recebida, utilizando para isso uma quantidade finita de tempo.

As instruções do algoritmo devem ser compreendidas por quem irá executá-las. Chamaremos de "**computador**" a pessoa ou equipamento que executa as instruções.

Desta forma, partindo do problema construímos um algoritmo que permite resolvê-lo através de um computador. O computador recebe então uma entrada válida para o problema, e fornece a saída esperada após executar as instruções definidas pelo algoritmo.

Um mesmo algoritmo pode ser representado de diversas formas, e podemos ter vários algoritmos distintos para um mesmo problema, com tempos de execução variados. De fato, para um mesmo problema e uma mesma entrada, podemos ter um algoritmo que executa em poucos segundos e outro que leva bilhões de anos para terminar.

1.1.1 Exemplo: máximo divisor comum

Dizemos que um número inteiro positivo d divide um inteiro não negativo n quando o resto da divisão de n por d é igual a zero. Ex.: 3 divide 12, pois o resto da divisão de 12 por 3 vale zero.

O máximo divisor comum de dois inteiros não negativos m e n, denotado por mdc(m, n), é o maior inteiro que divide m e n. Os valores m e n não podem ser ambos iguais a zero.

O algoritmo de Euclides, criado há mais de 2 mil anos, é baseado na propriedade abaixo:

$$\text{mdc}(m,n) = \begin{cases} m, & \text{se } n = 0 \\ \text{mdc}(n, m \bmod n), & \text{se } n > 0 \end{cases}$$

onde $m \bmod n$ denota o resto da divisão de m por n.

Por exemplo,

$$\text{mdc}(60, 24) = \text{mdc}(24, 12) = \text{mdc}(12, 0) = 12.$$

Temos a seguir uma descrição mais estruturada do algoritmo.

Algoritmo: Euclides para calcular mdc(m,n)

Passo 1: Se $n = 0$, retorne m e pare; caso contrário, vá para o Passo 2.
Passo 2: Divida m por n e armazene o resto da divisão em r.
Passo 3: Copie o valor de n para m, e copie o valor de r para n.

Em um **pseudocódigo** intercalamos instruções em linguagem natural com comandos de linguagens de programação. Os comandos são mais precisos, mas tendem a deixar o algoritmo mais difícil de entender. Busque um meio termo ideal entre precisão e facilidade de compreensão.

Temos a seguir um pseudocódigo para o Algoritmo de Euclides.

Algoritmo: Euclides(m,n)

Entrada: Dois inteiros não negativos m e n.
m e n não podem ser ambos nulos.
Saída : Maior divisor comum de m e n.

```
1 while n ≠ 0 do
2     r ← m mod n
3     m ← n
4     n ← r
5 return m
```

Para ser um algoritmo válido é necessário que **execute em tempo finito**. No algoritmo de Euclides temos que verificar se o valor de n sempre atinge o valor zero, que é a condição de saída do while da linha 1. Em cada iteração a variável n recebe o resto da divisão de m por n, que é um valor entre 0 e $n - 1$, e portanto menor que n. Como o valor de n reduz em cada iteração, e não pode assumir valor negativo, concluímos que o algoritmo para.

Outro algoritmo possível seria iniciar com o menor valor entre m e n, e ir decrementando até encontrar um divisor comum. Este algoritmo está detalhado abaixo. Ao contrário do algoritmo de Euclides, este algoritmo não aceita nenhuma das variáveis de entrada igual a zero, pois provocaria divisão por zero. Isto ilustra a importância de **especificar bem as entradas**. O algoritmo para, pois todo inteiro é divisível por 1.

Algoritmo: Checagem de inteiros consecutivos para calcular mdc(m,n)

Passo 1: Atribua para t o mínimo entre m e n.
Passo 2: Divida m por t. Se o resto é zero, vá para o Passo 3; caso contrário, vá para o Passo 4.
Passo 3: Divida n por t. Se o resto é zero, retorne o valor de t e pare; caso contrário, vá para o Passo 4.
Passo 4: Decremente em 1 o valor de t. Vá para o Passo 2.

Outro algoritmo possível seria utilizar a fatoração em primos de m e n, como é feito no ensino médio. Este procedimento está detalhado no algoritmo abaixo. Por exemplo, a fatoração em primos de 60 é $2^2 \cdot 3 \cdot 5$, e a fatoração em primos de 24 é $2^3 \cdot 3$. Como o fator (divisor) primo 2 ocorre 2 vezes na fatoração do 60, e 3 vezes na fatoração do 24, o fator primo 2 será considerado $\min\{2, 3\} = 2$ vezes. O fator primo 3 ocorre uma vez em cada fatoração, e o fator primo 5 ocorre apenas na fatoração do 60. Multiplicando os fatores primos comuns, obtemos $2^2 \cdot 3 = 12$, que é o mdc de 60 e 24.

Algoritmo: Procedimento do ensino médio para calcular mdc(m,n)

Passo 1: Encontre os divisores primos de m.
Passo 2: Encontre os divisores primos de n.
Passo 3: Identifique os divisores primos que aparecem tanto no Passo 1 quanto no Passo 2. Se um mesmo divisor primo ocorrer k_1 vezes no Passo 1 e k_2 no Passo 2, este divisor primo será considerado $\min\{k_1, k_2\}$ vezes no passo seguinte.
Passo 4: Calcule o produto dos divisores primos comuns identificados no Passo 3, e retorne o resultado deste produto.

Estas instruções **são ambíguas** caso o executor (computador) não saiba como encontrar os divisores primos de um inteiro positivo. Também pode ser necessário detalhar o Passo 3 para execução em computador, pois precisamos encontrar elementos que pertençam simultaneamente às duas listas.

Para **encontrar os divisores primos** de um inteiro positivo n, precisamos antes determinar os números primos com valor no máximo n. Para isso podemos usar o algoritmo Crivo de Eratóstenes, cujo pseudocódigo está fornecido abaixo. O procedimento consiste em ir removendo os múltiplos de todos os valores que formos encontrando ao percorrer os valores de 2 até n. Como os primos não são múltiplos de números maiores que 1, ao final restarão apenas os números primos.

```
Exemplo: encontrar os números primos até 25.
 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Remoção dos múltiplos de 2:
[2] 3  _  5  _  7  _  9 __ 11 __ 13 __ 15 __ 17 __ 19 __ 21 __ 23 __ 25
Remoção dos múltiplos de 3:
 2 [3]    5     7     _    11    13    __    17    19    __    23    25
Remoção dos múltiplos de 5:
 2  3    [5]    7          11    13          17    19          23    __
```

O algoritmo recebe um inteiro positivo n e retorna uma lista contendo os números primos com valor no máximo n. Nas linhas 1 e 2 uma lista auxiliar A é inicializada com o valor p em cada posição p indo de 2 até n. No for da linha 3 removemos de A todo elemento que não é primo, ou seja, que é múltiplo de algum outro valor em A. Cada valor entre colchetes no exemplo corresponde a um valor de p no for da linha 3. A remoção de um valor p em A é feita na linha 7 gravando o valor zero na posição p. Como em cada iteração removemos os múltiplos de p, podemos avançar o índice j em p posições na linha 8. Como todos os múltiplos de valores menores que p já foram excluídos, os múltiplos de p menores que $p \cdot p$ já foram excluídos, permitindo assim começar a exclusão na posição $p \cdot p$ (linha 5). Como a exclusão começa em $p \cdot p$ e vai no máximo até n, concluímos que para produzir remoções o valor de p não ultrapassa $\lfloor \sqrt{n} \rfloor$, permitindo assim restringir o for da linha 3.

Algoritmo: Crivo_Eratóstenes(n)

Entrada: Inteiro positivo n.
Saída : Lista L com todos os números primos com valor no máximo n.

```
1  for p de 2 até n do
2      A[p] ← p                                  // inicialização da lista auxiliar A
3  for p de 2 até ⌊√n⌋ do
4      if A[p] ≠ 0 then                          // se p não foi removido ainda
5          j ← p · p          // múltiplos menores que p · p já foram removidos
6          while j ≤ n do                        // múltiplo até n são removidos
7              A[j] ← 0                  // valor zero indica que j foi removido
8              j ← j + p                 // próximo removido é múltiplo de p
9  i ← 1
10 for p de 2 até n do                  } Copia os elementos
11     if A[p] ≠ 0 then                 } não nulos de A
12         L[i] ← A[p]                  } para as primeiras
13         i ← i + 1                    } posições de L.
14 return L
15
```

Como a lista de divisores primos é vazia para o caso de n igual a 1, o procedimento do ensino médio para calcular mmc(m, n) não funciona quando pelo menos uma das entradas é igual a 1, ilustrando novamente a importância de **especificar bem as entradas**.

A **notação** $L[1..n]$ indica que os elementos na lista L **são indexados de 1 até** n, onde n é o tamanho da lista. Caso a faixa de índices não seja explicitamente indicada, vamos assumir que os índices das listas começam em 1.

1.2 Passos comuns no projeto e análise

1.2.1 Entender o problema

Antes de começar a projetar o algoritmo é fundamental **entender o problema completamente**. Leia a descrição com cuidado, faça perguntas se surgirem dúvidas, resolva alguns exemplos pequenos manualmente, e pense a respeito de casos especiais.

Pode acontecer de **já existir algoritmo** para o problema, ou de existir algoritmo que facilite muito a tarefa. Portanto, quanto mais algoritmos você conhecer, maior será sua capacidade de resolver problemas computacionais. Para poder utilizar estes algoritmos, é importante **entender como eles funcionam**, pois assim você será capaz de adaptá-lo para o seu problema. Também é importante **entender as limitações** do algoritmo, e **como ele se compara com outros algoritmos** para o mesmo problema, pois assim você poderá escolher melhor entre as opções disponíveis. Por exemplo, temos várias implementações para um mesmo tipo abstrato de dados, mas algumas implementações tornarão seu algoritmo mais eficiente que outras. Se não encontrar nenhum algoritmo, terá que **projetar seu próprio algoritmo**.

Uma entrada do algoritmo especifica uma **instância** do problema, ou seja, uma situação particular que precisa ser resolvida. Como vimos no exemplo do máximo divisor comum, é fundamental **especificar exatamente o conjunto de instâncias** que o algoritmo precisa considerar, pois o algoritmo **precisa funcionar corretamente para todas as entradas válidas**.

Se não realizar bem esta etapa de entendimento do problema, provavelmente vai **desperdiçar tempo em retrabalho** nas etapas seguintes.

1.2.2 Avaliar a capacidade computacional disponível

Como atualmente computadores com vários processadores são amplamente disponíveis, podemos projetar **algoritmos paralelos** que exploram esta capacidade de execução concorrente. Apesar disso, os algoritmos são geralmente projetados para executar em uma **random-access machine (RAM)**, onde as instruções são executadas de forma sequencial. Nesta disciplina vamos considerar apenas o modelo RAM.

Algumas aplicações exigem atenção para a **limitação de processamento e memória disponível** durante o projeto do algoritmo, como por exemplo (i) aplicações com grande demanda de processamento ou com muitos dados para processar; (ii) sistemas embarcados, onde o dispositivo dispõe de poucos recursos; e (iii) sistemas de tempo real, onde o algoritmo não pode ultrapassar um tempo limite de execução. Porém, quando um algoritmo é estudado do ponto de vista teórico, como será o caso nesta disciplina, a análise é feita **independente das características particulares de um computador**.

1.2.3 Buscar solução exata ou aproximada

Alguns problemas não podem ser resolvidos de forma exata. Por exemplo, não é possível determinar todas as casas decimais de uma raiz quadrada que resulta em um número irracional, como $\sqrt{2}$. Outros problemas podem ser resolvidos de forma exata, mas a quantidade demandada de processamento geralmente exige um **tempo de execução inviável**, como é o caso da versão de otimização de problemas NP-completos que estudaremos. Neste casos temos que recorrer a algoritmos que fornecem **solução aproximada**, que chamamos de **heurísticas**.

1.2.4 Projetar

Existem algumas **técnicas gerais para projeto de algoritmos**, como divisão e conquista, programação dinâmica e algoritmos gulosos. Ao projetar um algoritmo, você tem a opção de tentar aplicar algumas destas técnicas gerais. Embora elas não resolvam todos os problemas, existe boa chance de funcionar para problema que você está considerando, ou de fornecer um ponto de partida útil para o projeto do algoritmo. Estudaremos várias destas técnicas de projeto ao longo da disciplina.

Mesmo que algumas das técnicas gerais permitam projetar o algoritmo, identificar a técnica apropriada e aplicar a técnica geralmente não são tarefas simples, pois **exigem criatividade** do projetista. Porém, **com a prática estas tarefas tornam-se mais fáceis**.

A **escolha apropriada das estruturas de dados** impacta no tempo de execução do algoritmo. Portanto, é muito importante conhecer bem os principais tipos abstratos de dados, e as complexidade de suas principais implementações. Por este motivo teremos uma aula reservada para revisar e discutir este assunto.

1.2.5 Especificar

No exemplo do máximo divisor comum, vimos que é possível **expressar um algoritmo** (i) em linguagem natural livre, (ii) em linguagem natural organizada em passos, e (iii) em pseudocódigo. Em livros antigos de computação encontramos também fluxogramas, que foram abandonados por representar de forma conveniente apenas algoritmos muito simples.

A **linguagem natural** é de fácil compreensão, mas é muito difícil descrever um algoritmo de forma clara e concisa sem produzir ambiguidades. Como o **pseudocódigo** combina linguagem natural com comandos de linguagens de programação, ele fornece uma descrição mais precisa e geralmente mais sucinta que utilizar apenas linguagem natural.

Não existe uma convenção para escrita de pseudocódigo, e portanto cada livro/artigo adota uma forma particular. Apesar disso, as convenções adotadas são parecidas com as linguagens de programação modernas, facilitando assim a compreensão. Vou **adotar a seguinte convenção**: comandos (palavras reservadas) em negrito, omissão de declaração de variáveis (exceto tipos abstratos de dados), blocos determinados pela identação (como no python), ← para atribuição, e // para comentários.

O **código** em uma determinada linguagem de programação é certamente uma forma de especificar um algoritmo, mas o nível de detalhes exigidos por linguagens de programação dificultam a compreensão. Portanto, se seu algoritmo estiver ficando muito grande, pode ficar mais fácil de compreendê-lo se você trocar parte dele por descrição em linguagem natural (não ambígua). **Encontrar o meio termo ideal entre linguagem natural e comandos de linguagem de programação** exige dedicação no refinamento do algoritmo. Lembre que **seu objetivo é deixar o pseudocódigo o mais fácil possível** de entender, **sem inserir ambiguidades**.

1.2.6 Provar a corretude

Depois de especificar o algoritmo temos que provar que, para toda entrada válida, o algoritmo **fornece a saída correta e executa em tempo finito**. A técnica de demonstração geralmente utilizada é a **prova por indução**. Veremos que é possível projetar o algoritmo já considerando esta prova por indução, facilitando assim a demonstração de corretude.

Note que **apresentar instâncias onde o algoritmo funciona corretamente não é suficiente** para provar sua corretude, pois alguém ainda pode construir uma nova instância onde o algoritmo falha. Por outro lado, apresentar uma instância em que o algoritmo falha é suficiente para provar que o algoritmo não é correto.

No caso de heurísticas, podemos tentar provar que o valor da solução fornecida não é pior que um determinado percentual do valor da solução ótima. Neste caso a heurística é chamada de **algoritmo aproximativo**.

1.2.7 Analisar

Depois de demonstrar que o algoritmo é correto, desejamos analisar sua **eficiência de tempo** e sua **eficiência de espaço**. A eficiência de tempo indica o consumo de CPU, e a eficiência de espaço indica o consumo de memória.

A função que descreve o tempo de execução em função do tamanho da entrada é chamada de **complexidade de tempo**, e a função que descreve o consumo de memória em função do tamanho da entrada é chamada de **complexidade de espaço**. A forma exata destas funções geralmente é complexa, mas podemos simplificá-las utilizando **notação assintótica**. Na disciplina vamos considerar principalmente a complexidade de tempo, e vamos chamá-la simplesmente de *complexidade*.

Se a eficiência de tempo ou espaço não é viável, temos que retornar para a fase de projeto para **refinar o algoritmo**. Mesmo que seja viável, em aplicações onde a entrada é grande, a redução da complexidade produz grande economia em hardware, valendo a pena o esforço para melhorar a eficiência do algoritmo.

Para alguns problemas existe **demonstração de limite para a eficiência** que pode ser alcançada. Por exemplo, sabe-se que algoritmos de ordenação baseados em comparação de elementos não conseguem complexidade abaixo de $c \cdot n \log n$, onde n é tamanho do array de entrada e c é uma constante. Então não é possível baixar a complexidade de um algoritmo de ordenação que já possui a complexidade mínima $c \cdot n \log n$. Porém, para a maioria dos problemas não dispomos deste tipo de demonstração. Sem este limite, existe sempre a esperança de conseguir baixar a complexidade. Este é o caso até de problemas simples, como a multiplicação de inteiros.

1.2.8 Codificar

Uma vez convencido que o algoritmo é correto e suficientemente eficiente, podemos partir para a **implementação** em alguma linguagem de programação. Se o algoritmo foi bem especificado, a codificação provavelmente será simples. Podemos, porém, cometar **erros de codificação**. Existem **métodos formais** que provam que uma codificação não possui bugs, mas infelizmente estas técnicas são viáveis apenas para programas pequenos. Na prática temos que recorrer à execução de **testes**, que não garantem código totalmente correto, mas são fundamentais para reduzir a chance de erros.

Para simplificar a especificação dos algoritmos durante a disciplina, eles não terão instruções para **testar se as entradas são válidas**. Na implementação, porém, esta verificação deveria ser feita.

Existem estratégias para deixar o código mais rápido, mas elas geralmente reduzem o tempo de execução apenas por um fator constante. Dependendo da aplicação pode valer o esforço para fazer estas **otimizações de código**. Estas estratégias não serão abordadas nesta disciplina. Nosso foco será a redução do crescimento da função de complexidade, que permite reduzir o tempo de execução para entradas grandes de forma bem mais significativa que a otimização de código.

Após implementar o algoritmo, podemos realizar uma **análise empírica de eficiência**. O método consiste em coletar os tempos de execução para um determinado conjunto de instâncias, e inferir uma função que mapeia o tempo de execução com o tamanho da entrada.

1.2.9 Conclusões e discussões

Alguns problemas **não podem ser resolvidos através de algoritmos**. Temos exemplos não ambíguos e com saídas simples ('sim' ou 'não'), como é o caso do *problema da parada* estudado na disciplina de teoria da computação.

Realizar um bom projeto de algoritmo envolve **muitos refinamentos**, ou seja, é resultado de esforço repetitivo e retrabalho. Neste sentido não é diferente de produzir um bom texto, ou uma boa obra de arte. Porém não é um esforço mecânico, pois exige **criatividade** do projetista, tornando assim a tarefa estimulante para quem acha divertido solucionar este tipo de desafio.

1.3 Tipos de problemas importantes

1.3.1 Ordenação

O **problema de ordenação** consiste em rearranjar os itens de uma lista em ordem não decrescente, ou em ordem não crescente. Para isso precisamos ser capazes de comparar todo par de itens (**relação de ordem total**). Esta comparação é natural para números e caracteres. No caso de strings, podemos usar a **ordem lexicográfica**, onde comparamos os caracteres de mesma posição até encontrar a primeira posição onde os dois caracteres são distintos, ou até uma das strings terminar. Por exemplo, as palavras em um dicionário da língua portuguesa estão em ordem lexicográfica. Quando os itens são registros, geralmente escolhemos um **campo chave** que irá determinar como os registros são comparados. Por exemplo, se os registros possuem as informações dos alunos, podemos querer ordenar pelo nome, ou pelo número de matrícula.

O uso mais comum da ordenação é **facilitar a tarefa de busca**. Imagine como seria difícil encontrar uma palavra em um dicionário, ou um telefone em uma lista telefônica, se eles não estivessem ordenados. A ordenação é também muito **utilizada como passo auxiliar** de diversos algoritmos. Temos até uma técnica projeto (*algoritmos gulosos*) exige uma ordenação dos itens.

Como vimos, sabemos que o tempo de execução de algoritmos de ordenação é pelo menos da ordem de $n \log n$ para o caso geral onde é necessário comparar pares de itens. Já dispomos de algoritmos de ordenação cujo **tempo de execução é da ordem de** $n \log n$ operações, como o Quicksort, o Mergesort e o Heapsort. Na prática, não existe um algoritmo mais rápido em todas as aplicações, mas para entradas grandes devemos escolher um destes algoritmos com complexidade $n \log n$. Em alguns casos especiais é possível ordenar realizando menos de $n \log n$ operações, mas estes casos não são comuns. Se desejar saber mais sobre estes casos particulares, veja o vídeo sobre *ordenação em tempo linear*.

Uma **ordenação é estável** se preserva a ordem original dos itens em caso de empate. Por exemplo, suponha que os alunos estejam ordenados por nome, e você realiza uma ordenação estável pelo IRA. Neste caso os alunos com mesmo IRA ficarão ordenados pelo nome. Um algoritmo de ordenação é chamado **in-place** quando a memória extra demandada não depende do tamanho da lista que está ordenando.

1.3.2 Busca

O **problema de busca** consiste em encontrar um dado elemento em um conjunto ou em um multiconjunto (conjunto que permite elementos repetidos). Temos a **busca sequencial**, que no pior caso percorre todos os n elementos. Se os elementos estão ordenados, podemos realizar a busca em tempo da ordem de $\log n$ usando uma **busca binária**. Também conseguimos buscas em tempo $\log n$ se os elementos são mantidos em **árvores de busca balanceadas**. Podemos realizar a busca em tempo médio constantes quando os elementos são armazenados em **tabelas hash**.

Não existe uma estratégia de busca que é a **melhor em todas as aplicações**. Por exemplo, algumas demandam memória adicional, e outras demandam que os elementos sejam mantidos ordenados ou balanceados. Em aplicações em que muitas inserções e remoções são intercaladas com as buscas, o tempo de execução destas operações deve ser levado em conta na escolha da estrutura de dados. Caso não lembre destes conceitos, sugiro que assista os vídeos *dicionário, árvores binárias de busca, e tabelas hash*.

1.3.3 Processamento de strings

Uma **string** é uma sequência de caracteres de um dado alfabeto. Por exemplo, podemos usar strings para modelar texto, ou sequências de bits (zeros e uns), ou sequências de genes (alfabeto contendo apenas os caracteres A, C, G e T). Um problema particularmente importante é o **casamento de strings** (*string matching*), que consiste em localizar uma dada palavra em um texto.

1.3.4 Problemas em grafos

Informalmente, um **grafo** é um conjunto de vértices (também chamados de nós) e um conjunto de segmentos de reta (chamados de arestas) conectando pares de vértices. Um grafo modela uma relação binária qualquer, e por isso é tem **muitas aplicações** em computação.

Os **algoritmos básicos em grafos** abstraem o que estes grafos estão modelando, e desta forma podem ser utilizados em uma variedade de aplicações. O **caminhamento em grafos** consiste em percorrer todos os vértices e arestas alcançáveis através de arestas a partir de um nó inicial. Assumindo um custo em cada aresta, o problema do **caminho mais curto** entre dois nós fornece um conjunto de arestas de custo mínimo que conecta estes nós. Vários algoritmos em grafos vão aparecer ao longo da disciplina como exemplos de aplicações de técnicas de projeto.

Para alguns problemas em grafos, como o problema do caixeiro viajante (TSP) e o problema de coloração em grafos, **não dispomos de algoritmo eficiente**, ou seja, só conseguimos obter solução exata para grafos pequenos. O problema do **caixeiro viajante (TSP)** consiste em encontrar um conjunto de arestas de custo mínimo que permite passar exatamente uma vez em cada vértice e retornar para o vértice inicial. O problema de **coloração em grafos** consiste em utilizar o menor número de cores atribuindo uma cor a cada vértice de modo que nenhuma aresta liga dois vértices com a mesma cor.

Algoritmos em grafos podem **resolver problemas que originalmente não são definidos em termos de grafos**. Por exemplo, suponha que alguns pares de aulas não podem ocorrer em uma mesma sala, pois compartilham pelo menos um horário ao longo da semana, e desejamos determinar o menor número de salas necessárias. Podemos modelar este problema como coloração em grafos, onde cada aula é um vértice, uma aresta indica que as duas aulas não podem ocupar uma mesma sala, e uma cor representa uma sala.

1.3.5 Problemas combinatórios

São problemas em que uma solução é um objeto combinatório (como **permutação, combinação ou subconjunto**). Temos também uma função que atribui um **valor/custo** para cada uma destas possíveis soluções. Desejamos encontrar uma solução que satisfaz algumas restrições e que possui valor máximo ou custo mínimo.

Como são objetos combinatórios, o número de possíveis soluções cresce exponencialmente com o tamanho da entrada. Portanto, em aplicações práticas geralmente é **inviável avaliar cada possível solução**. Apesar disso, alguns problemas combinatórios **dispõem de algoritmo eficiente**, baseados em propriedades particulares do problema. Por exemplo, o problema de encontrar um **caminho mais curto** entre dois nós de um grafo é combinatório, pois uma solução é subconjunto das arestas, e este problema possui algoritmo eficiente.

Apesar disso, **para a maioria dos problemas combinatórios não conhecemos algoritmo eficiente**, exigindo assim a utilização de heurísticas (solução aproximada). O problema do caixeiro viajante e a coloração em grafos são exemplos de problemas combinatórios sem algoritmo eficiente conhecido. No problema do **caixeiro viajante** uma solução é um subconjunto das arestas, e na **coloração em grafos** uma solução é uma permutação das cores disponíveis (com repetição), sendo a 1a cor para o 1o vértice, a 2a cor para o 2o vértice, assim sucessivamente.

1.3.6 Problema geométricos

Problemas geométricos manipulam objetos geométricos como **pontos, linhas e polígonos**. Várias aplicações utilizam estes algoritmos, como computação gráfica, robótica e análise de imagens médicas. Como exemplo de problema geométrico temos o problema do **par mais próximo**, que consiste em localizar, em um conjunto de pontos, um par de pontos com a menor distância entre eles. Outro exemplo é o problema da **casca convexa**, que consiste em determinar o menor polígono convexo que contém todos os pontos de um dados conjunto de pontos.

1.3.7 Problemas numéricos

Os problemas numéricos manipulam **números reais**, como por exemplo obter a solução de um sistemas de equações, a solução numérica de integrais, ou a avaliação de funções. Um subconjunto dos números reais – os números irracionais – possuem precisão infinita e portanto só podem ser **representados de forma aproximada** em computadores. Desta forma, à medida que cálculos envolvendo números reais vão sendo realizados, **erros de arredondamento** podem ir se acumulando, produzindo assim um resultado final muito distante do valor correto. Portanto, o projetista de algoritmos para problemas numéricos deve estar atento para esta dificuldade. Existem estratégias para controlar estes erros, mas estão fora do escopo desta disciplina.

1.4 Fundamentos matemáticos

1.4.1 Logaritmos e exponenciais

Notação:

- $\log x$ é um logaritmo na base 10.
- $\lg x$ é um logaritmo na base 2.
- $\ln x$ é um logaritmo na base $e = 2{,}71828\ldots$ (número de Euler).

Nas propriedades abaixo assumimos que a base do logaritmo é maior que 1:

1. $\log_a 1 = 0$
2. $\log_a a = 1$
3. $\log_a x^y = y \log_a x$
4. $\log_a x \cdot y = \log_a x + \log_a y$
5. $\log_a x/y = \log_a x - \log_a y$
6. $a^{\log_b x} = x^{\log_b a}$
7. $\log_a x = \dfrac{\log_b x}{\log_b a} = \log_a b \cdot \log_b x$

1.4.2 Contagem

Número de permutações de um conjunto com n elementos: $n!$
Número de k combinações de um conjunto com n elementos:

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

Número de subconjuntos de um conjunto com n elementos: 2^n.

Fórmula de Stirling: $n! \approx \sqrt{2\pi n}\left(\dfrac{n}{e}\right)^n$ quando $n \to \infty$.

1.4.3 Somatórios

Somatórios importantes

1. Se u e ℓ são inteiros com $\ell \le u$,
   $\displaystyle\sum_{i=\ell}^{u} = \underbrace{1+1+\cdots+1}_{u-\ell+1 \text{ vezes}} = u-\ell+1$
   Ex.: $\sum_{i=1}^{n} 1 = n$.
2. $\displaystyle\sum_{i=1}^{n} i = 1+2+\cdots+n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$
3. $\displaystyle\sum_{i=1}^{n} i^2 = 1^2+2^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$
4. $\displaystyle\sum_{i=1}^{n} i^k = 1^k+2^k+\cdots+n^k \approx \frac{1}{k+1}n^{k+1}$
5. Para $a \neq 1$,
   $\displaystyle\sum_{i=0}^{n} a^i = 1+a+\cdots+a^n = \frac{a^{n+1}-1}{a-1}$
   Ex.: $\sum_{i=0}^{n} 2^i = 2^{n+1}-1$.
6. $\displaystyle\sum_{i=1}^{n} i \cdot 2^i = 1\cdot 2 + 2\cdot 2^2 + \cdots + n\cdot 2^n = (n-1)2^{n+1}+2$
7. $\displaystyle\sum_{i=1}^{n} \frac{1}{i} = 1+\frac{1}{2}+\cdots+\frac{1}{n} \approx \ln n + \gamma,$
   onde $\gamma \approx 0{,}5772\ldots$ (constante de Euler).
8. $\displaystyle\sum_{i=1}^{n} \lg i \approx n \lg n$

Manipulação de somatórios

1. $\displaystyle\sum_{i=\ell}^{u} c \cdot a_i = c \cdot \sum_{i=\ell}^{u} a_i$
2. $\displaystyle\sum_{i=\ell}^{u} (a_i \pm b_i) = \sum_{i=\ell}^{u} a_i \pm \sum_{i=\ell}^{u} b_i$
3. $\displaystyle\sum_{i=\ell}^{u} a_i = \sum_{i=\ell}^{m} a_i + \sum_{i=m+1}^{u} a_i,$ onde $\ell \le m < u$
4. $\displaystyle\sum_{i=\ell}^{u} (a_i - a_{i-1}) = a_u - a_{\ell-1}$

Aproximação de somatório por integral

Para $f(x)$ não decrescente,

$$\int_{\ell-1}^{u} f(x)dx \le \sum_{i=\ell}^{u} f(i) \le \int_{\ell}^{u+1} f(x)dx$$

Para $f(x)$ não crescente,

$$\int_{\ell}^{u+1} f(x)dx \le \sum_{i=\ell}^{u} f(i) \le \int_{\ell-1}^{u} f(x)dx$$

1.4.4 Pisos e tetos

O **piso** de um número real x, denotado por $\lfloor x \rfloor$ é o maior inteiro menor ou igual a x. Ex.: $\lfloor 4{,}7 \rfloor = 4$, $\lfloor 4 \rfloor = 4$. O **teto** de um número real x, denotado por $\lceil x \rceil$ é o menor inteiro maior ou igual a x. Ex.: $\lceil 4{,}7 \rceil = 5$, $\lceil 4 \rceil = 4$.

1. $x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$
2. $\lfloor x + n \rfloor = \lfloor x \rfloor + n$ e $\lceil x + n \rceil = \lceil x \rceil + n$, para x real e n inteiro
3. $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$
4. $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$

1.4.5 Aritmética modular

Se n e m são inteiros, e p é inteiro positivo,

$$(n+m) \bmod p = (n \bmod p + m \bmod p) \bmod p$$
$$(nm) \bmod p = ((n \bmod p)(m \bmod p)) \bmod p$$

1.4.6 Técnicas de demonstração

Suponha que desejamos mostrar que a **implicação** $p \to q$ é verdadeira. Chamamos p de **premissa** e q de **conclusão**. A implicação $p \to q$ é falsa apenas quando a premissa p é verdadeira e a conclusão q é falsa. Temos então que mostrar que esta situação não pode ocorrer.

Prova direta

Na prova direta **assumimos que a premissa p é verdadeira**, e aplicamos uma série de inferências/equivalências até **deduzir que a conclusão também é verdadeira**.

Ex.: Mostrar que se n é um inteiro ímpar, então n^2 é ímpar.
Prova: Premissa: n é ímpar. Conclusão: n^2 é ímpar.
O inteiro n é par sse existe um inteiro k tal que $n = 2k$, e é ímpar sse existe um inteiro k tal que $n = 2k+1$. A premissa diz que n é ímpar, então assumindo que esta premissa é verdadeira temos que existe inteiro k tal que $n = 2k+1$. Desta forma, podemos escrever n^2 como

$$n^2 = (2k+1)^2 = 4k^2 + 4k + 1 = 2 \cdot (2k^2 + 2k) + 1.$$

A expressão $2k^2 + 2k$ resulta em um número inteiro, pois é produto e soma de inteiros. Logo, como existe inteiro $k' = 2k^2 + 2k$ tal que $n^2 = 2k' + 1$, concluímos que n^2 é ímpar. Ou seja, deduzimos que a conclusão é verdadeira. □

Prova por contraposição

Baseia-se na equivalência $p \to q \equiv \neg q \to \neg p$, ou seja, para provar a implicação $p \to q$ podemos realizar uma prova direta de $\neg q \to \neg p$, que consiste em **assumir que a conclusão é falsa e deduzir que a premissa é falsa**. Esta técnica é utilizada quando é mais fácil partir da conclusão que da premissa.

Ex.: Mostrar que se n^2 é par, então n é par.
Prova: Assumindo que n^2 é par temos que existe inteiro k tal que $n^2 = 2k$, mas partindo desta informação como vamos mostrar que n é par? A demonstração é mais simples partindo da contrapositiva, que é a expressão "se n é ímpar, então n^2 é ímpar", que já provamos no exemplo anterior. □

Prova por contradição

Consiste em assumir que $p \to q$ é falso, e com base nisso deduzir uma sentença falsa qualquer. A negação de $p \to q$ é equivalente a $p \wedge \neg q$, portanto a técnica consiste em **assumir que a premissa é verdadeira e que a conclusão é falsa, e com isso deduzir qualquer sentença que seja falsa**.

Ex.: Mostrar que se um número somado a ele mesmo é igual a ele mesmo, então este número é o zero.
Prova: Desemos mostrar que $x+x = x \ \to \ x = 0$. Por contradição assumimos que $2x = x$ e $x \neq 0$. Como $x \neq 0$, podemos dividir os dois lados de $2x = x$ por x, resultando em $2 = 1$, que é uma expressão falsa. Com isso provamos que $2x = x \ \to \ x = 0$. □

Prova por indução

A prova por indução é utilizada para mostrar que uma dada propriedade é verdadeira para todo inteiro n maior ou igual a um determinado valor inicial n_0. Primeiro temos que provar o **caso base**, ou seja, mostrar que a propriedade vale para $n = n_0$. Em seguida provamos o **passo indutivo**, ou seja, mostramos que se propriedade vale para n então também vale para $n+1$.

Ex.: Mostrar que $\sum_{i=1}^{n} i = n \cdot (n+1)/2$ para todo n inteiro não negativo.
Prova:

Caso base: Para $n = 1$ temos que $\sum_{i=1}^{1} i = 1$ e $1 \cdot (1+1)/2 = 1$, o que nos permite concluir que a propriedade vale para $n = 1$.

Passo indutivo:

- **Hipótese de indução**: assuma que vale para um dado $n \geq n_0$, ou seja,
$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}.$$
- Queremos mostrar que vale para $n+1$, ou seja,
$$\sum_{i=1}^{(n+1)} i = \frac{(n+1) \cdot ((n+1)+1)}{2} = \frac{(n+1)(n+2)}{2}.$$
- Aplicando a hipótese de indução em $\sum_{i=1}^{n+1} i$, obtemos
$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^{n} i\right) + (n+1) = \frac{n \cdot (n+1)}{2} + (n+1) = \frac{(n+1) \cdot (n+2)}{2}.$$

□

Leitura complementar

Capítulo introdutório dos livros:

- CORMEN, Thomas et al. Algoritmos - Teoria e Prática.
- DASGUPTA, Sanjoy et al. Algoritmos.
- SKIENA, Steven. The algorithm design manual.

Fundamentos matemáticos:

- ROSEN, Kenneth H. Matemática discreta e suas aplicações.