**Introdução**

* Dado um problema, como encontramos um algoritmo eficiente para sua solução?
* Encontrado um algoritmo, como comparar este algoritmo com outros algoritmos que solucionam o mesmo problema?
* Como deveríamos julgar a qualidade de um algoritmo?
* Qual é o algoritmo de menor custo possível para resolver um problema particular?

**Medidas de Complexidade**

Como selecionar um algoritmo quando existem vários que solucionam o problema?

**Soluções**

* Escolher um algoritmo de fácil entendimento codificação e depuração;
* Escolher um algoritmo que faz uso eficiente dos recursos do computador;

**Meados dos anos 60**: tempos de execução de um programa que implementava um algoritmo – os resultados encontrados dependiam:

* Do computador utilizado
* Do compilador utilizado

**Atualmente**: análise de um algoritmo compreende dois passos:

* Análise prévia: a qual determina a função que delimita o tempo de computação do algoritmo;
* Coleta de estatísticas reais sobre o algoritmo;

**Finalidades**

* Dá oportunidade de se projetar novas maneiras de se fazer uma mesma tarefa de um modo mais rápido;
* Obter estimativas de tempos de execução de programas que implementam um algoritmo. A complexidade do algoritmo dá idéia do esforço computacional do programa, que é uma medida do número de operações executadas pelo programa;

**Motivação**

* Como estimar a quantidade de **recursos** (tempo, memória) que um algoritmo consome/gasta**(análise de complexidade);**
* Como provar a “corretude” de um algoritmo;
* Como projetar algoritmos eficientes (= rápidos) para vários problemas computacionais;

**Técnicas de Demonstração**

**Demonstração Exaustiva**

Uma **conjectura** é uma sentença sendo proposta como verdade, mas que precisa ser provada para virar teorema. A **conjectura** pode ser provada verificando-se que ela é verdadeira para todos os elementos da coleção. Para provar a falsidade da conjectura, basta achar um contra-exemplo.

**Exemplo**: Prove a conjectura: “Para todo inteiro positivo n, n! <= n2".

**Solução**: A conjectura é falsa, pois não é verdade para todo n: é falsa para n = 4.

**Demonstração Direta**

**Demonstração por Contraposição**

**Demonstração por Absurdo**

**Demonstração por Indução**

* Na demonstração por indução, queremos demonstrar a validade de P(n), uma propriedade P com um parâmetro natural n associado, para todo valor de n;
* Há um número infinito de casos a serem considerados, um para cada valor de n. Demonstramos os infinitos casos de uma só vez:
	+ **Base da Indução**: Demonstramos P(1)
	+ **Hipótese de Indução**: Supomos que P(n) é verdadeiro
	+ **Passo de Indução**: Provamos que P(n+1) é verdadeiro, a partir da hipótese de indução

**Técnicas de Projeto de Algoritmos**

Um conjunto de técnicas de projeto de algoritmos compreendem os métodos de codificação de algoritmos de forma salientar sua complexidade, levando em conta a forma pela qual determinado algoritmo chega a solução desejada.

**Indução**

* Formulação do algoritmo é baseado na demonstração por indução. **Exemplos**: Cálculo de polinômio, problema do skyline, fatores de balanceamento (diferença entre a altura de uma subárvore esquerda e a altura de uma subárvore direita de um nó N), o problema da celebridade.

**Recursividade**

* Um procedimento que chama a si mesmo, direta ou indiretamente, é dito ser **recursivo**;
* Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.

**Exemplo**: Árvore binária de pesquisa (Todos os registros com chaves menores estão na sub-árvore esquerda;Todos os registros com chaves maiores estão na sub-árvore direita);

* Usa-se uma **pilha** para armazenar os dados usados em cada chamada de um procedimento que ainda não terminou;
* Todos os dados não globais vão para a pilha, registrando o estado corrente da computação;
* Quando uma ativação anterior prossegue, os dados da pilha são recuperados;
* **Procedimentos recursivos introduzem a possibilidade de iterações que podem não terminar**;
* Existe a necessidade de considerar o problema de **terminação**. É fundamental que a chamada recursiva a um procedimento P esteja sujeita a uma condição B, a qual se torna não-satisfeita em algum momento da computação;
* Ao se fazer a análise de um algoritmo recursivo, deve-se também analisar o crescimento da pilha;

**Divisão e Conquista**

* Consiste em dividir o problema em partes menores, encontrar soluções para essas partes (supostamente mais fácil) e combiná-las em uma solução global;
* Geralmente leva a soluções eficientes e elegantes, principalmente se forem recursivas;
* Essa técnica consiste das seguintes fases (executadas nesta ordem):

1. Divisão (particionamento) do problema original em sub-problemas similares ao original mas que são menores em tamanho;

2. Resolução de cada sub-problema sucessivamente e independentemente (em geral de forma recursiva);

3. Combinação das soluções individuais em uma solução global para todo o problema.

**Exemplos:**

* **Ordenação**: Mergesort, Quicksort (tecnicamente falando, o Quicksort poderia ser chamado de um algoritmo conquista e divisão);
* **Pesquisa**: Pesquisa binária;
* **Algoritmos aritméticos**: multiplicação de inteiros, multiplicação de matrizes;
* **Algoritmos geométricos**: Fecho convexo (Dado um conjunto de pontos, determine o menor [polígono](http://pt.wikipedia.org/wiki/Pol%C3%ADgono%22%20%5Co%20%22Pol%C3%ADgono) convexo contendo todos os pontos), par mais próximo (dado um conjunto de pontos, determine o par de pontos com menor distância entre eles);

**Aplicações:**

* Processar independentemente partes do conjunto de dados. **Exemplo**: Mergesort.
* Eliminar partes do conjunto de dados a serem examinados. **Exemplo**: Pesquisa binária.
* Processar separadamente partes do conjunto de dados, mas onde a solução de uma parte influencia no resultado da outra.

**Programação Dinâmica**

* Aplicada quando recursão produz repetição dos mesmos subproblemas;
* **Proposta**: Reusar computação (tabelas);

Para que o paradigma da programação dinâmica possa ser aplicado, é preciso que o problema tenha estrutura recursiva*:* a solução de toda instância do problema deve "conter" soluções de subinstâncias da instância. Essa estrutura recursiva pode, em geral, ser representada por uma [recorrência](http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/recorrencias.html)s e a recorrência pode ser traduzida em um algoritmo recursivo. O algoritmo recursivo é tipicamente ineficiente porque refaz, muitas vezes, a solução de cada subinstância. Uma vez escrito o algoritmo recursivo, entretanto, é fácil construir uma tabela para armazenar as solução das subinstância e assim evitar que elas sejam recalculadas. A tabela é a base de um algoritmo de programação dinâmica.

**Algoritmos Gulosos**

* Aplicado a problemas de **otimização**. **Exemplos**: Como uma empresa de mudança deve alocar os móveis em um caminhão baú? Como uma companhia telefônica deve rotear chamadas de modo a fazer um melhor uso de suas linhas e conexões?
* Decisões tomadas em cada passo;
* Estratégia de “pegar” o melhor no momento;

Para resolver um problema, um algoritmo guloso escolhe, em cada iteração, o objeto mais "apetitoso" que vê pela frente. O objeto escolhido passa a fazer parte da solução que o algoritmo constrói.

Um algoritmo guloso é "míope": ele toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro. Um algoritmo guloso jamais se arrepende ou volta atrás: as escolhas que faz em cada iteração são definitivas.

Embora algoritmos gulosos pareçam obviamente corretos, a prova de sua correção é, em geral, muito sutil. Para compensar, algoritmos gulosos são muito rápidos e eficientes. (É preciso dizer, entretanto, que os problemas que admitem soluções gulosas são um tanto raros).