MC448 — Análise de Algoritmos I

Cid Carvalho de Souza e Cândida Nunes da Silva

24 de Julho de 2006

Agradecimentos

- Várias pessoas contribuíram direta ou indiretamente com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Uma lista destes "colaboradores" (em ordem alfabética) é dada abaixo:
  - Célia Picinin de Mello
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofillof
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias
- Qualquer erro encontrado neste material é de inteira responsabilidade de Cid C. de Souza !

Problemas e seus algoritmos

- Neste curso estudamos *problemas* que admitem uma solução computacional; isto é, um *programa* de computador que tome como entrada qualquer *instância* do problema e produza sempre uma resposta correta para aquela instância.
- Tais programas evoluem do *projeto e verificação da corretude de um algoritmo*, que é posteriormente codificado em alguma linguagem, compilado, carregado e executado em um computador.

Problemas do nosso interesse

- Problemas que não admitem solução computacional não são de interesse deste curso.
- Decidir quais problemas (obviamente de uma certa natureza) têm solução computacional é, por seu lado, um problema muito difícil, para o qual não se conhece uma solução computacional ;-)
- Dentre aqueles problemas que admitem solução computacional há aqueles que são mais ou menos difíceis. O estudo dessas classes de dificuldade de problemas é iniciado na disciplina MC538.

Propriedades de algoritmos

- Pode haver vários algoritmos para o mesmo problema, que diferem entre si
  - na forma de atacar o problema (como);
  - na quantidade de recursos utilizados (quanto tempo ou memória); e
  - na qualidade da resposta (exata, aproximada, ou com uma dada probabilidade de estar correta).

Propriedades de algoritmos

Em MC438/448 estamos interessados na *corretude* (como) e *complexidade* (quantidade de recursos) dos algoritmos que produzem respostas exatas. Mais especificamente, estudaremos técnicas para

- o projeto e verificação de corretude de algoritmos para um dado problema; e
- técnicas para avaliar a quantidade de recursos utilizados por um algoritmo, permitindo compará-lo a outros algoritmos para o mesmo problema, de forma mais ou menos independente do computador em que venham a ser implementados.

Vejamos a seguir alguns exemplos das idéias que exploraremos ao longo deste curso.

Vamos resolver alguns problemas?

Busca de um elemento em um vetor:

- **Problema** Dado um vetor v de n inteiros e um inteiro x, determinar se x está no vetor.
- **Algoritmo:** Procure em todas as posições do vetor até encontrar i tal que $v[i] = x$.

Este problema é *"fácil"*, tem um algoritmo *simples* e de *complexidade* proporcional ao número de elementos n do vetor. Dizemos que a complexidade é ($\Theta(n)$).

Vamos resolver alguns problemas?

Ordenação dos elementos de um conjunto:

- **Problema:** Ordenar n valores comparáveis.
- **Algoritmo:** Selecionar sucessivamente o menor elemento e retorná-lo, retirando-o do conjunto.

Este problema é *"fácil"*, tem um algoritmo *simples* e de *complexidade* proporcional ao quadrado do número de elementos n do vetor. Dizemos que a complexidade é $\Theta(n^2)$.
Existem algoritmos mais sofisticados, e de *complexidade* menor que este, como o Mergesort, cuja complexidade é $\Theta(n \log n)$.

Vamos resolver alguns problemas?

Atribuição de professores a disciplinas:

- **Problema:** Dados um conjunto de n professores, um conjunto de n disciplinas e as listas de disciplinas que cada professor pode ministrar, determinar se existe uma atribuição de disciplinas a professores de forma que cada professor ministre exatamente uma disciplina.
- **Algoritmo:** Tentar atribuir professores às disciplinas de todas as formas possíveis até encontrar uma que resolva o problema.

Esse algoritmo *simples* tem *complexidade* muito grande, fatorial no número de professores ($\Theta(n!)$ no pior caso).
Existe ainda um algoritmo mais rápido, *polinomial* (de complexidade $\Theta(n^c)$, para c uma constante inteira); no entanto, o algoritmo polinomial não é tão *simples*.

Vamos resolver alguns problemas?

Problema do Caixeiro Viajante:

- **Problema:** Dadas n cidades e as distâncias entre elas, determinar a seqüência em que as cidades (todas) devem ser visitadas de forma que a distância total percorrida seja mínima.
- **Algoritmo:** Calcular a distância total de cada percurso e escolher o menor.

O algoritmo *simples* mencionado possui *complexidade* alta, fatorial no número de cidades($\Theta(n!)$).
Não se conhece algoritmo polinomial para este problema. É um problema NP-completo, um problema *"difícil"*.

Vamos resolver alguns problemas?

Problema da Parada:

- **Problema:** Dado um programa qualquer P e uma entrada E do programa, determinar se o programa P pára quando alimentado com a entrada E.
- **Algoritmo:** ????????.

Este é um problema *indecidível*, ou seja, é possível demonstrar matematicamente que não existe uma solução computacional para ele (dentro do nosso conceito do que seja um computador).

Desiderata

- Esta disciplina tem por objetivo discutir conceitos básicos sobre análise e complexidade de algoritmos como parte do aprendizado de longo prazo para resolução e classificação de problemas computacionais.
- Ao longo deste curso, restringiremos nosso estudo aos problemas que possuem algoritmos polinomiais.
- Ao final do curso, os alunos devem ser capazes de projetar algoritmos, provar sua corretude e analisar sua complexidade.

Técnicas de Projeto de Algoritmos

- Indução.
- Divisão e Conquista.
- Programação Dinâmica.
- Guloso.
- Busca Exaustiva.

Algoritmos para Problemas Clássicos

- Algoritmos de ordenação e estatísticas de ordem.
- Algoritmos para problemas em grafos:
  - Buscas em grafos (largura e profundidade)
  - Árvore geradora mínima
  - Caminhos mínimos

Noções Preliminares

- O que é um algoritmo correto?
  - Aquele que pára para toda instância do problema, retornando uma solução correta.
- O que é analisar um algoritmo?
  - Predizer a quantidade de recursos utilizados (memória, tempo de execução, número de processadores, acessos a disco, etc).
  - Na maioria dos casos estaremos interessados em avaliar o tempo de execução gasto pelo algoritmo.
  - Contaremos o número de operações efetuadas.

Noções Preliminares

- Como representar um algoritmo?
  - Pseudo-código (abstrato, independe de implementação).

Entrada: Vetor A de n inteiros.
Saída: Vetor A ordenado.

```
Ordenação(A, n)
    para j ← 2 até n faça
        valor ← A[j]
        /* Insere A[j] na seqüência ordenada A[1..j − 1] */
        i ← j − 1
        enquanto i > 0 e A[i] > valor faça
            A[i + 1] ← A[i]
            i ← i − 1
        A[i + 1] ← valor
```

Modelo Computacional

- A análise de um algoritmo depende do modelo computacional subjacente.
- O modelo computacional define quais são os recursos disponíveis e quanto custam.
- Queremos analisar algoritmos de forma genérica, independente de implementação.
- Utilizaremos o modelo abstrato RAM:
  - Simula máquinas convencionais.
  - Um único processador que executa instruções seqüencialmente.
  - Somas, subtrações, multiplicações, divisões, comparações e atribuições são feitas em tempo constante.

Pior Caso e Notação Assintótica

- O tempo de execução de um algoritmo depende da instância. Faremos análises de pior caso.
- A complexidade dos algoritmos será expressa como uma função do tamanho da entrada, já que isso espelha, de alguma forma, quão "inteligente" é o algoritmo.
- Basta compararmos a ordem de grandeza das funções de complexidade. Para isso utilizaremos a notação assintótica.

Crescimento de funções

Notação Assintótica

- Vamos expressar complexidade através de funções em variáveis que descrevam o tamanho de instâncias do problema. Exemplos:
  - Problemas de aritmética de precisão arbitrária: número de bits (ou bytes) dos inteiros.
  - Problemas em grafos: número de vértices e/ou arestas
  - Problemas de ordenação de vetores: tamanho do vetor.
  - Busca em textos: número de caracteres do texto ou padrão de busca.
- Vamos supor que funções que expressam complexidade são sempre positivas, já que estamos medindo número de operações.

Comparação de Funções

- Vamos comparar funções assintoticamente, ou seja, para valores grandes, desprezando constantes multiplicativas e termos de menor ordem.

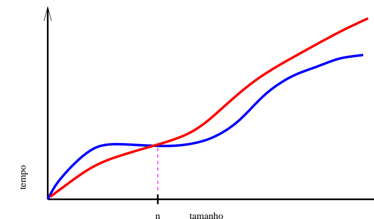
	$n = 100$	$n = 1000$	$n = 10^4$	$n = 10^6$	$n = 10^9$
$\log n$	2	3	4	6	9
n	100	1000	10^4	10^6	10^9
$n \log n$	200	3000	$4 \cdot 10^4$	$6 \cdot 10^6$	$9 \cdot 10^9$
n^2	10^4	10^6	10^8	10^{12}	10^{18}
$100n^2 + 15n$	$1,0015 \cdot 10^6$	$1,00015 \cdot 10^8$	$\approx 10^{10}$	$\approx 10^{14}$	$\approx 10^{20}$
2^n	$\approx 1,26 \cdot 10^{30}$	$\approx 1,07 \cdot 10^{301}$	?	?	?

Classe O

Definição:

$O(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in O(g(n))$, então $f(n)$ cresce no máximo tão rapidamente quanto $g(n)$.



Classe O

Definição:

$O(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in O(g(n))$, então $f(n)$ cresce no máximo tão rapidamente quanto $g(n)$.

Exemplo:

$\frac{1}{2}n^2 - 3n \in O(n^2)$

Valores de c e n_0 que satisfazem a definição são

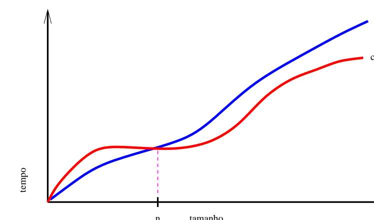
$$c = \frac{1}{2} \text{ e } n_0 = 7.$$

Classe Ω

Definição:

$\Omega(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Omega(g(n))$, então $f(n)$ cresce no mínimo tão lentamente quanto $g(n)$.



Classe Ω

Definição:

$\Omega(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq cg(n) \leq f(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Omega(g(n))$, então $f(n)$ cresce no mínimo tão lentamente quanto $g(n)$.

Exemplo:

$\frac{1}{2}n^2 - 3n \in \Omega(n^2)$

Valores de c e n_0 que satisfazem a definição são

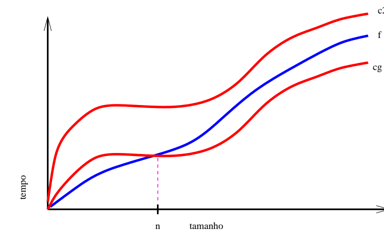
$$c = \frac{1}{14} \text{ e } n_0 = 7.$$

Classe Θ

Definição:

$\Theta(g(n)) = \{f(n) :$ existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$.



Classe Θ

Definição:

$\Theta(g(n)) = \{f(n) :$ existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in \Theta(g(n))$, então $f(n)$ cresce tão rapidamente quanto $g(n)$.

Exemplo:

$\frac{1}{2}n^2 - 3n \in \Theta(n^2)$

Valores de c_1, c_2 e n_0 que satisfazem a definição são

$$c_1 = \frac{1}{14},\ c_2 = \frac{1}{2} \text{ e } n_0 = 7.$$

Classe o

Definição:

$o(g(n)) = \{f(n) :$ para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq f(n) < cg(n)$, para todo $n \geq n_0\}$.

Informalmente, dizemos que, se $f(n) \in o(g(n))$, então $f(n)$ cresce mais lentamente que $g(n)$.

Exemplo:

$1000n^2 \in o(n^3)$

Para todo valor de c, um n_0 que satisfaz a definição é

$$n_0 = \left\lceil \frac{1000}{c} \right\rceil + 1.$$

Classe ω

Definição:

$\omega(g(n)) = \{f(n) :$ para toda constante positiva c, existe uma constante $n_0 > 0$ tal que $0 \leq cg(n) < f(n)$, para todo $n \geq n_0$.$\}$

Informalmente, dizemos que, se $f(n) \in \omega(g(n))$, então $f(n)$ cresce mais rapidamente que $g(n)$.

Exemplo:

$\frac{1}{1000}n^2 \in \omega(n)$

Para todo valor de c, um n_0 que satisfaz a definição é

$$n_0 = \lceil 1000c \rceil + 1.$$

Definições equivalentes

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

$f(n) \in O(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

$f(n) \in \Theta(g(n))$ se $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

$f(n) \in \Omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$.

$f(n) \in \omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Propriedades das Classes

Transitividade:

Se $f(n) \in O(g(n))$ e $g(n) \in O(h(n))$, então $f(n) \in O(h(n))$.
Se $f(n) \in \Omega(g(n))$ e $g(n) \in \Omega(h(n))$, então $f(n) \in \Omega(h(n))$.
Se $f(n) \in \Theta(g(n))$ e $g(n) \in \Theta(h(n))$, então $f(n) \in \Theta(h(n))$.
Se $f(n) \in o(g(n))$ e $g(n) \in o(h(n))$, então $f(n) \in o(h(n))$.
Se $f(n) \in \omega(g(n))$ e $g(n) \in \omega(h(n))$, então $f(n) \in \omega(h(n))$.

Propriedades das Classes

Reflexividade:

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

Simetria:

$f(n) \in \Theta(g(n))$ se, e somente se, $g(n) \in \Theta(f(n))$.

Simetria Transposta:

$f(n) \in O(g(n))$ se, e somente se, $g(n) \in \Omega(f(n))$.
$f(n) \in o(g(n))$ se, e somente se, $g(n) \in \omega(f(n))$.

Exemplos

Quais as relações de comparação assintótica das funções:

- 2^{π}
- $\log n$
- n
- $n \log n$
- n^2
- $100n^2 + 15n$
- 2^n

Demonstração Direta

A *demonstração direta* de uma implicação $p \Rightarrow q$ é uma sequência de passos lógicos (implicações):

$$p \Rightarrow p_1 \Rightarrow p_2 \Rightarrow \cdots \Rightarrow p_n \Rightarrow q,$$

que resultam, por transitividade, na implicação desejada.
Cada passo da demonstração é um axioma ou um teorema demonstrado previamente.

Exemplo:

Provar que $\sum_{i=1}^{k} 2i - 1 = k^2$.

Demonstração pela Contrapositiva

A *contrapositiva* de $p \Rightarrow q$ é $\neg q \Rightarrow \neg p$.
A contrapositiva é equivalente à implicação original. A veracidade de $\neg q \Rightarrow \neg p$ implica a veracidade de $p \Rightarrow q$, e vice-versa.
A técnica é útil quando é mais fácil demonstrar a contrapositiva que a implicação original.
Para demonstrarmos a contrapositiva de uma implicação, podemos utilizar qualquer técnica de demonstração.

Exemplo:

Provar que se $2 \mid 3m$, então $2 \mid m$.

Demonstração por Contradição

A *Demonstração por contradição* envolve supor absurdamente que a afirmação a ser demonstrada é falsa e obter, através de implicações válidas, uma conclusão contraditória.
A contradição obtida implica que a hipótese absurda é falsa e, portanto, a afirmação é de fato verdadeira.
No caso de uma implicação $p \Rightarrow q$, equivalente a $\neg p \vee q$, a negação é $p \wedge \neg q$.

Exemplo:

Dados os inteiros positivos n e $n+1$, provar que o maior inteiro que divide ambos n e $n+1$ é 1.

Demonstração por Casos

Na *Demonstração por Casos*, particionamos o universo de possibilidades em um conjunto finito de casos e demonstramos a veracidade da implicação para cada caso.
Para demonstrar cada caso individual, qualquer técnica de demonstração pode ser utilizada.

Exemplo:
Provar que a soma de dois inteiros x e y de mesma paridade é sempre par.

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.

Exemplo:
Provar que $\sum_{i=1}^{k} 2i - 1 = k^2$, agora por indução.

Indução Fraca × Indução Forte

A *indução forte* difere da *indução fraca* (ou *simples*) apenas na suposição da hipótese.
No caso da indução forte, devemos supor que a propriedade vale para todos os casos anteriores, não somente para o anterior, ou seja:

- **Base da Indução:** Demonstramos $P(1)$.
- **Hipótese de Indução Forte:** Supomos que $P(k)$ é verdadeiro, para todo $k \leq n$.
- **Passo de Indução:** Provamos que $P(n+1)$ é verdadeiro, a partir da hipótese de indução.

Exemplo:
Prove que todo inteiro n pode ser escrito como a soma de diferentes potências de 2.

Indução matemática

Exemplo 1

Demonstre que, para todos os números naturais x e n, $x^n - 1$ é divisível por $x - 1$.

Demonstração:

- A **base da indução** é, naturalmente, o caso $n = 1$. Temos que $x^n - 1 = x - 1$, que é obviamente divisível por $x - 1$. Isso encerra a demonstração da base da indução.

Exemplo 1 (cont.)

- A **hipótese de indução** é: *Suponha que, para um valor qualquer de $n \geq 1$, $x^{n-1} - 1$ seja divisível por $x - 1$ para todos os naturais x.*
- O **passo de indução** é: *Supondo a h.i., vamos mostrar $x^n - 1$ é divisível por $x - 1$, para todos os naturais x.*
  Primeiro reescrevemos $x^n - 1$ como

  $$x^n - 1 = x(x^{n-1} - 1) + (x - 1).$$

  Pela h.i., $x^{n-1} - 1$ é divisível por $x - 1$. Portanto, o lado direito da equação acima é, de fato, divisível por $x - 1$.

A demonstração por indução está completa. ■

Exemplo 2

Demonstre que a equação

$$\sum_{i=1}^{n} (3 + 5i) = 2.5n^2 + 5.5n$$

vale para todo inteiro $n \geq 1$.

Demonstração:

- A **base da indução** é, naturalmente, o caso $n = 1$. Temos

  $$\sum_{i=1}^{1} (3 + 5i) = 8 = 2.5 \times 1^2 + 5.5 \times 1.$$

  Portanto, a somatória tem o valor previsto pela fórmula fechada, demonstrando que a equação vale para $n = 1$.

Exemplo 2 (cont.)

- A **hipótese de indução** é: *Suponha que a equação valha para um valor de n qualquer, $n \geq 1$.*
- O **passo de indução é:** *Supondo a h.i., vamos mostrar que a equação vale para o valor $n + 1$.* O caminho é simples:

$$\begin{aligned}
\sum_{i=1}^{n+1} (3 + 5i) &= \sum_{i=1}^{n} (3 + 5i) + (3 + 5(n + 1)) \\
&= 2.5n^2 + 5.5n + (3 + 5(n + 1)) \text{ (pela h.i.)} \\
&= 2.5n^2 + 5.5n + 5n + 8 \\
&= 2.5n^2 + 5n + 2.5 + 5.5n + 5.5 \\
&= 2.5(n + 1)^2 + 5.5(n + 1).
\end{aligned}$$

A última linha da dedução mostra que a fórmula vale para $n + 1$. A demonstração por indução está completa. ■

Exemplo 3

Demonstre que a inequação

$$(1+x)^n \geq 1+nx$$

vale para todo natural n e real x tal que $(1+x) > 0$.

Demonstração:

- A **base da indução** é, novamente, $n = 1$. Nesse caso ambos os lados da inequação são iguais a $1+x$, mostrando a sua validade. Isto encerra a prova do caso base.

Exemplo 3 (cont.)

- A **hipótese de indução** é: *Suponha que a inequação valha para um valor qualquer de* $n \geq 1$*; isto é, que* $(1+x)^n \geq 1+nx$*, para todo real* x *tal que* $(1+x) > 0$.
- O **passo de indução é:** *Supondo a h.i., vamos mostrar que a inequação vale para o valor* $n+1$*, isto é, que* $(1+x)^{n+1} \geq 1+(n+1)x$*, para todo* x *tal que* $(1+x) > 0$.
  Novamente, a dedução é simples:

$$\begin{aligned}(1+x)^{n+1} &= (1+x)^n(1+x)\\ &\geq (1+nx)(1+x) \text{ (pela h.i. e } (1+x)>0)\\ &= 1+(n+1)x+nx^2\\ &\geq 1+(n+1)x. \text{ (já que } nx^2 \geq 0)\end{aligned}$$

A última linha mostra que a inequação vale para $n+1$, encerrando a demonstração. ■

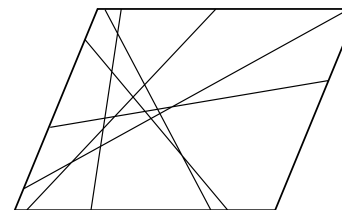
Exemplo 4

Demonstre que o número T_n de regiões no plano criadas por n retas em *posição geral*, isto é, duas a duas concorrentes e não mais que duas concorrentes no mesmo ponto, é igual a

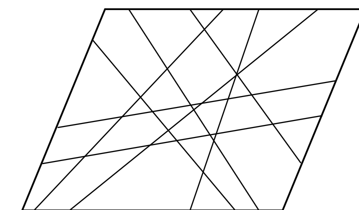
$$T_n = \frac{n(n+1)}{2} + 1.$$

Exemplo 4 (cont.)

Antes de prosseguirmos com a demonstração vejamos exemplos de um conjunto de retas que está em posição geral e outro que não está.



Em posição geral



Não estão em posição geral

Exemplo 4 (cont.)

Demonstração: A idéia que queremos explorar para o passo de indução é a seguinte: supondo que T_n vale para um valor qualquer de n, adicionar uma nova reta em posição geral e tentar assim obter a validade de T_{n+1}.

- A **base da indução** é, naturalmente, $n = 1$. Uma reta sozinha divide o plano em duas regiões. De fato,

$$T_1 = (1 \times 2)/2 + 1 = 2.$$

Isto conclui a prova para $n = 1$.

Exemplo 4 (cont.)

- A **hipótese de indução** é: *Suponha que* $T_n = (n(n+1)/2) + 1$ *para um valor qualquer de* $n \geq 1$.
- O **passo de indução é:** *Supondo a h.i., vamos mostrar que para* $n + 1$ *retas em posição geral vale*

$$T_{n+1} = \frac{(n+1)(n+2)}{2} + 1.$$

Considere um conjunto L de $n + 1$ retas em posição geral no plano e seja l uma dessas retas. Então, as retas do conjunto $L' = L \setminus \{l\}$ obedecem à hipótese de indução e, portanto, o número de regiões distintas do plano definidas por elas é $(n(n+1))/2 + 1$.

Exemplo 4 (cont.)

- Além disso, l intersecta as outras n retas em n pontos distintos. O que significa que, saindo de uma ponta de l no infinito e após cruzar as n retas de L', a reta l terá cruzado $n + 1$ regiões, dividindo cada uma destas em duas outras.
- Assim, podemos escrever que

$$\begin{aligned} T_{n+1} &= T_n + n + 1 \\ &= \frac{n(n+1)}{2} + 1 + n + 1 \text{ (pela h.i.)} \\ &= \frac{(n+1)(n+2)}{2} + 1. \end{aligned}$$

Isso conclui a demonstração. ∎

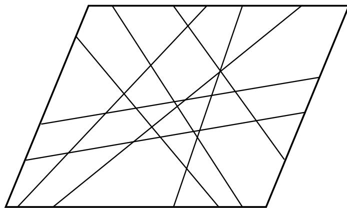
Exemplo 5

Definição:

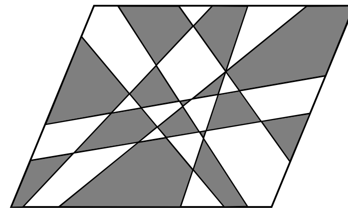
Um conjunto de n retas em posições arbitrárias no plano define regiões convexas cujas bordas são segmentos das n retas. Duas dessas regiões são *adjacentes* se as suas bordas se intersectam em algum segmento de reta não trivial, isto é contendo mais que um ponto. Uma *k-coloração* dessas regiões é uma atribuição de uma de k cores a cada uma das regiões, de forma que regiões adjacentes recebam cores distintas.

Exemplo 5 (cont.)

Veja exemplos dessas definições:



As regiões convexas



Uma 2-coloração do plano

Exemplo 5 (cont.)

Demonstre que para todo $n \geq 1$, existe uma 2-coloração das regiões formadas por n retas em posição arbitrária no plano.

Demonstração:

- A **base da indução** é, naturalmente, $n = 1$. Uma reta sozinha divide o plano em duas regiões. Atribuindo-se cores diferentes a essas regiões obtemos o resultado desejado.

  Isto conclui a prova para $n = 1$.

Exemplo 5 (cont.)

- A **hipótese de indução** é: *Suponha que exista uma 2-coloração das regiões formadas por n retas no plano, para um inteiro qualquer $n \geq 1$.*
- O **passo de indução é:** *Supondo a h.i., vamos exibir uma 2-coloração para as regiões formadas por $n + 1$ retas em posição arbitrária.*

  A demonstração do passo consiste em observar que a adição de uma nova reta l divide cada região atravessada por l em duas, e definir a nova 2-coloração da seguinte forma: as regiões em um lado de l mantém a cor herdada da hipótese de indução; as regiões no outro lado de l têm suas cores trocadas.

  Você é capaz de demonstrar que a 2-coloração obtida nesse processo obedece à definição ?

Exemplo 6

Vejamos agora um exemplo onde a indução é aplicada de forma um pouco diferente.

Demonstre que a série S_n definida abaixo obedece a

$$S_n = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^n} < 1,$$

para todo inteiro $n \geq 1$.

Demonstração: A base é $n = 1$, para a qual a inequação se reduz a $\frac{1}{2} < 1$, obviamente verdadeira.
Para a hipótese de indução, supomos que $S_n < 1$ para um valor qualquer de $n \geq 1$. Vamos mostrar que $S_{n+1} < 1$.

Exemplo 6 (cont.)

Pela definição de S_n, temos $S_{n+1} = S_n + \frac{1}{2^{n+1}}$.
Pela hipótese de indução, $S_n < 1$. Entretanto, nada podemos dizer acerca de S_{n+1} em conseqüência da hipótese, já que não há nada que impeça que $S_{n+1} \geq 1$.
A idéia aqui é manipular S_{n+1} um pouco mais:

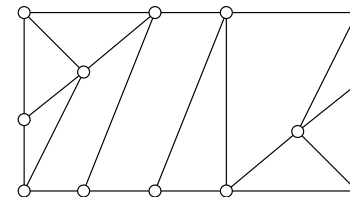
$$\begin{aligned} S_{n+1} &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^{n+1}} \\ &= \frac{1}{2} + \frac{1}{2}\left[\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n}\right] \\ &< \frac{1}{2} + \frac{1}{2} \times 1 \text{ (pela h.i.)} \\ &= 1. \end{aligned}$$

Isto conclui a demonstração. ■

Exemplo 7

Veremos a seguir um exemplo da aplicação de indução em Teoria dos Grafos.

Definição:
Um *grafo planar* é um grafo que pode ser desenhado no plano sem que suas arestas se cruzem. Um *grafo plano* é um desenho de grafo planar no plano, sem cruzamento de arestas (há inúmeros desenhos possíveis). Veja um exemplo de um grafo planar e um desenho possível dele no plano.



Exemplo 7 (cont.)

Definição:

- Um grafo plano define um conjunto F de *faces* no plano, que são as regiões continuas **maximais** do desenho, livre de segmentos de retas ou pontos.
- Os *componentes* de um grafo são seus subgrafos maximais para os quais existe um caminho entre quaisquer dois de seus vértices.
- Dado um grafo plano G, com v vértices, e arestas, f faces e c componentes, a *Fórmula de Euler* (F.E.) é a equação

$$v - e + f = 1 + c.$$

Queremos demonstrar a Fórmula de Euler por indução.

Exemplo 7 (cont.)

- Há várias possibilidades para se fazer indução neste caso. No livro de U. Manber encontra-se uma indução em duas variáveis, a chamada *indução dupla*, primeiro em v depois em f. Além disso, lá a Fórmula de Euler está descrita diferentemente, sem especificar o número de componentes. Isso torna a indução um pouco mais complicada.
- Nossa formulação é mais geral simplificando a demonstração. Esse um fenômeno comum em matemática: formulações mais poderosas quase sempre resultam em demonstrações mais simples.
- Vamos demonstrar a F.E. por indução em e, o número de arestas do grafo plano G.

Exemplo 7 (cont.)

Demonstração:

- A base da indução é $e = 0$. Temos $f = 1$ e $c = v$ e

$$v - e + f = v + 1 = 1 + c$$

como desejado. Isso demonstra a base.

- A hipótese de indução é: *Suponha que a F.E. valha para todo grafo com* $e - 1$ *arestas, para um inteiro* $e > 0$ *qualquer.*
- Seja G um grafo plano com e arestas, v vértices, f faces e c componentes. Seja a uma aresta qualquer de G. A remoção de a de G cria um novo grafo plano G' com $v' = v$ vértices e $e' = e - 1$ arestas, f' faces e c' componentes.
  A remoção de a de G pode ou não ter desconectado um componente de G. Caso tenha, $c' = c + 1$ e $f' = f$ (por que?). Caso contrário, teremos $c' = c$ e $f' = f - 1$ (por que?).

Exemplo 7 (cont.)

- No caso em que houve a criação de novo componente, temos

$$\begin{aligned} v - e + f &= v' - (e' + 1) + f' \\ &= 1 + c' - 1 \text{ (pela h.i.)} \\ &= c' \\ &= 1 + c. \end{aligned}$$

- Caso contrário obtemos

$$\begin{aligned} v - e + f &= v' - (e' + 1) + (f' + 1) \\ &= v' - e' + f' \\ &= 1 + c' \text{ (pela h.i.)} \\ &= 1 + c. \end{aligned}$$

Em ambos casos obtemos o resultado desejado. ■

Exercícios

1. Demonstre que
$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
para todo $n \geq 1$.
2. Demonstre que um número natural é divisível por 3 se e somente se a soma de seus dígitos é divisível por 3.

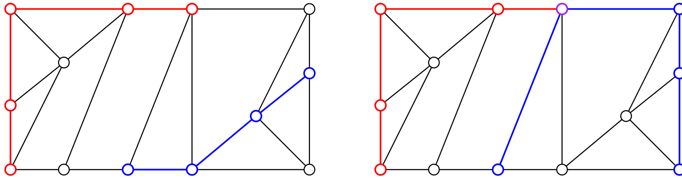
Exemplo 8

Este é um exemplo de indução forte. Antes algumas definições:

- Seja G um grafo não-orientado. O **grau** de um vértice v de G é o número de arestas incidentes a v, onde laços (arestas cujos extremos coincidem) são contados duas vezes.
- Um vértice é **ímpar** (**par**) se o seu grau é ímpar (par).
  O número de vértices ímpares em um grafo é sempre par (por quê ?).

Exemplo 8 (cont.)

Dois caminhos em G são **aresta-disjuntos** se não têm arestas em comum. Veja exemplos de caminhos aresta-disjuntos em um grafo:



Exemplo 8 (cont.)

Teorema:
Seja G um grafo (não-orientado) conexo e I o conjunto de vértices ímpares de G. Então é possível particionar os vértices de I em $|I|/2$ pares e encontrar caminhos aresta-disjuntos cujos extremos são os vértices de cada par.

Demonstração: A demonstração é por indução no número de arestas de G. Seja e esse número.

- A base da indução é o caso $e = 1$. Nesse caso $|I| = 0$ ou $|I| = 2$ e o teorema é válido trivialmente. (É possível usar a base $e = 0$, como o resto da demonstração deixará claro.)

Exemplo 8 (cont.)

- A hipótese de indução (forte) é: *Dado um inteiro $e > 0$ qualquer, suponha que para todos os grafos conexos com menos que e arestas valha o resultado do enunciado do teorema.*

  Vamos mostrar que o resultado vale para todo grafo conexo com e arestas.
- Seja então G um grafo qualquer com e arestas. Se $I = \emptyset$ não há nada que provar. Caso contrário existem pelo menos dois vértices u, v em I. Como G é conexo, existe um caminho π em G cujos extremos são u, v.
- Seja G' o grafo obtido removendo-se de G as arestas de π. G' tem menos que e arestas e menos dois vértices ímpares.
- Embora seja tentador aplicar a h.i. a G', nada garante que G' seja conexo. Se não for, a h.i. não se aplica.

Exemplo 8 (cont.)

- É possível consertar a situação mudando a hipótese para: *Dado um inteiro $e > 0$ qualquer, suponha que para todos os grafos com menos que e arestas valha o resultado do enunciado do teorema*

  Veja que removemos a restrição de conexidade, fortalecendo a hipótese.
- Com essa nova hipótese, a demonstração é a mesma até este ponto, com a ressalva de que π é um caminho com extremos em um mesmo componente de G. Mas agora podemos aplicar a h.i. a G': os caminhos de G' e π são todos aresta-disjuntos e formam o conjunto de caminhos desejados para G. ■

  Exercício: Você consegue demonstrar esse mesmo teorema usando indução fraca ?

Exemplo 9

Este é um exemplo de **indução reversa**, cujo princípio pode ser enunciado da seguinte forma:

> *Se a proposição P vale para um subconjunto infinito dos números naturais, e se é possível mostrar que a validade de P para um valor qualquer de n implica na validade para $n-1$, então P vale para todos os números naturais.*

Você consegue ver por que esse é um processo indutivo igualmente legítimo ?

Exemplo 9 (cont.)

Teorema:
Se $x_1, x_2, \ldots, x_n$ são todos números reais positivos, então

$$(x_1 x_2 \ldots x_n)^{\frac{1}{n}} \leq \frac{x_1 + x_2 + \ldots + x_n}{n}.$$

Demonstração: Em dois passos:

1. Vamos mostrar que a inequação vale para todos os valores de n que são potências 2, isto é $n = 2^k$, para k inteiro ≥ 0. Faremos esse passo por indução simples em k. Esse é o conjunto infinito de valores da indução reversa.
2. Mostraremos que se a inequação é verdadeira para um valor qualquer de $n > 1$, então é verdadeira para $n - 1$.

Exemplo 9 (cont.)

- Se $n = 2^0 = 1$, então o teorema vale trivialmente.
- Se $n = 2^1 = 2$, a inequação também é válida já que

$$\sqrt{x_1 x_2} \leq \frac{x_1 + x_2}{2}$$

pode ser verificada tomando-se o quadrado dos dois lados.

$$\begin{array}{rcll}
\sqrt{x_1 x_2} & \leq & (x_1 + x_2)/2 & \Leftrightarrow \\
x_1 x_2 & \leq & (x_1^2 + 2x_1 x_2 + x_2^2)/4 & \Leftrightarrow \\
2x_1 x_2 & \leq & x_1^2 + x_2^2 & \Leftrightarrow \\
0 & \leq & x_1^2 - 2x_1 x_2 + x_2^2 & \Leftrightarrow \\
0 & \leq & (x_1 - x_2)^2 &
\end{array}$$

Exemplo 9 (cont.)

(h.i.) Vamos supor agora que a inequação vale para $n = 2^k$, para k um inteiro qualquer ≥ 0.
Considere $2n = 2^{k+1}$ e reescreva o lado esquerdo da inequação como

$$(x_1 x_2 \ldots x_{2n})^{\frac{1}{2n}} = \sqrt{(x_1 x_2 \ldots x_n)^{\frac{1}{n}} (x_{n+1} x_{n+2} \ldots x_{2n})^{\frac{1}{n}}}.$$

Tome $y_1 = (x_1 x_2 \ldots x_n)^{\frac{1}{n}}$ e $y_2 = (x_{n+1} x_{n+2} \ldots x_{2n})^{\frac{1}{n}}$. Portanto

$$(x_1 x_2 \ldots x_{2n})^{\frac{1}{2n}} = \sqrt{y_1 y_2} \leq \frac{y_1 + y_2}{2}$$

pelo caso $n = 2$ já demonstrado.

Exemplo 9 (cont.)

Além disso, podemos aplicar a h.i. a y_1 e y_2, obtendo

$$y_1 \leq \frac{x_1 + x_2 + \ldots + x_n}{n},$$

$$y_2 \leq \frac{x_{n+1} + x_{n+2} + \ldots + x_{2n}}{n}.$$

Substituindo esses dois valores na inequação acima obtemos o resultado desejado para $2n$.

Exemplo 9 (cont.)

Vamos agora utilizar o princípio de indução reversa. Suponha que o resultado vale para um valor qualquer de $n > 1$ e vamos mostrar que vale para $n - 1$.

Dados $n - 1$ números positivos $x_1, x_2, \ldots, x_{n-1}$, defina

$$z := \frac{x_1 + x_2 + \ldots + x_{n-1}}{n-1}.$$

Por h.i., o teorema aplica-se a $x_1, x_2, \ldots, x_{n-1}, z$. Portanto

$$\begin{aligned}(x_1 x_2 \ldots x_{n-1} z)^{\frac{1}{n}} &\leq \frac{x_1 + x_2 + \ldots + x_{n-1} + z}{n} \\ &= z.\end{aligned}$$

Exemplo 9 (cont.)

Então

$$(x_1 x_2 \ldots x_{n-1} z)^{\frac{1}{n}} \leq z.$$

Elevando ambos os lados à potência $\frac{n}{n-1}$ obtemos

$$(x_1 x_2 \ldots x_{n-1} z)^{\frac{1}{n-1}} \leq z^{\frac{n}{n-1}}.$$

Finalmente, multiplicando por $z^{-\frac{1}{n-1}}$ ambos os lados, obtemos

$$(x_1 x_2 \ldots x_{n-1})^{\frac{1}{n-1}} \leq z = \frac{x_1 + x_2 + \ldots + x_{n-1}}{n-1},$$

o que prova a asserção para $n - 1$, encerrando a demonstração. ■

Algumas armadilhas - redução × expansão

- A demonstração do passo da indução simples supõe a proposição válida para um $n - 1$ qualquer e mostra que é válida para n.
- Portanto, devemos sempre partir de um caso geral n e **reduzí-lo** ao caso $n - 1$. Às vezes porém, parece mais fácil pensar no caso $n - 1$ e **expandí-lo** para o caso geral n.
- O perigo do procedimento de expansão é que ele não seja suficientemente geral, de forma que obtenhamos a implicação, a partir do caso $n - 1$, para um caso **geral** n.
- As conseqüências de um lapso como esse podem ser a obtenção de uma estrutura de tamanho n fora da hipótese de indução, ou a a prova da proposição para casos particulares de estruturas de tamanho n e não todos, como se espera.

Algumas armadilhas - redução $\times$ expansão

Por exemplo, se na demonstração da Fórmula de Euler tivéssemos optado por adicionar uma aresta $a = (i, j)$ a um grafo plano de $e - 1$ arestas, deveríamos:

- garantir que i, j estivessem na mesma face; e
- considerar os casos em que i, j estivessem em componentes iguais e diferentes.

Caso contrário, ou produziríamos um cruzamento de arestas, ou provaríamos o teorema para uma subclasse de grafos planos apenas.

Mesmo com cuidados extras, a sensação de estar esquecendo algum caso é maior na expansão do que na redução, naturalmente, por estarmos aumentando o nosso universo de possibilidades.

Algumas armadilhas - outros passos mal dados

O que há de errado com a demonstração da seguinte proposição, claramente falsa ?

Proposição:
Considere n retas no plano, concorrentes duas a duas. Então existe um ponto comum a todas as n retas.

Demonstração:

- A base da indução é o caso $n = 1$, claramente verdadeiro.
- Para o caso $n = 2$, também é fácil ver que a proposição é verdadeira.
- Considere a proposição válida para $n - 1$ qualquer, $n > 2$, e considere n retas no plano concorrentes duas a duas.

Algumas armadilhas - outros passos mal dados

Pela h.i., todo subconjunto de $n - 1$ das n retas têm um ponto em comum. Sejam S_1, S_2 dois desses subconjuntos, distintos entre si.

A interseção $S_1 \cap S_2$ tem $n - 2$ retas; portanto, o ponto em comum às retas de S_1 tem que ser igual ao ponto comum às retas de S_2 já que, caso contrário, duas retas distintas de $S_1 \cap S_2$ se tocariam em mais que um ponto, o que não é possível.

Portanto, a asserção vale para n, completando a demonstração.

Certo ?

Engano:
O passo de indução vale para todos os $n > 2$, exceto para $n = 3$ pois, nesse caso, $S_1 \cap S_2$ tem apenas uma reta. Como a validade da asserção para $m > 2$ qualquer depende da sua validade para cada valor de $n = 2, 3, \ldots, m - 1$, a falha em $n = 3$ implica em falha para todos os $m > 3$.

Invariantes de laço e indução matemática

Definição:
Um invariante de um laço de um algoritmo é uma propriedade que é satisfeita pelas variáveis do algoritmo independente de qual iteração do laço tenha sido executada por último.

- Usados em provas de corretude de algoritmos
- Tipicamente um algoritmo é composto de vários laços executados em seqüência
- Para cada laço pode-se obter um invariante que, uma vez provado, garanta o funcionamento correto daquela parte *específica* do algoritmo
- A corretude do algoritmo como um todo fica provada se for provado que os invariantes de todos os laços estão corretos
- O difícil é encontrar o invariante que leva à prova da corretude do algoritmo

Invariantes de laço e indução matemática

Um exemplo:

Usando *invariante de laços*, provamos a corretude de um algoritmo que converte um número inteiro para a sua representação binária.

Converte_Binário(n)

```
1  ▷ na saída, b contém a representação binária de n
2  t ← n;
3  k ← −1;
4  enquanto  t > 0 faça
5      k ← k + 1;
6      b[k] ← t  mod 2;
7      t ← t div 2;
8  retornar b.
```


Invariantes de laço e indução matemática

Invariante:

Ao entrar no laço 4–7, o inteiro m representado pelo subvetor $b[0 \dots k]$ é tal que $n = t.2^{k+1} + m$.

Demonstração: seja $j = k + 1$ ($j = \#$ execuções da linha 4)

Deseja-se provar por induçao em j que $n = t.2^j + m$.

Base da indução: $j = 0$. Trivial, pois antes de fazer o laço, $m = 0$ (subvetor vazio de b) e $n = t$ pela linha 2.

Hipótese de indução: no início da execução da linha 4 na j^{a} iteração, tem-se que $n = t(j).2^j + m(j)$, sendo $m(j) = \sum_{i=0}^{j-1} 2^i.b[i]$.

Invariantes de laço e indução matemática

Passo de indução: antes da execução da linha 4 na $(j+1)^{\text{a}}$ iteração, deve valer que $n = t(j+1).2^{j+1} + m(j+1)$, com $m(j) = \sum_{i=0}^{j} 2^i.b[i]$.

Pelas linhas 6 e 7, respectivamente, tem-se que:

$t(j+1) = t(j) \texttt{ div } 2$ e $m(j+1) = [t(j) \mod 2].2^j + m(j)$.

Caso $t(j) = 2p$ (par):

$$\begin{aligned} t(j+1).2^{j+1} + m(j+1) &= p.2^{j+1} + m(j) = 2p.2^j + m(j) \\ &= t(j).2^j + m(j) = n \quad \text{(HI)} \end{aligned}$$

Caso $t(j) = 2p + 1$ (ímpar):

$$\begin{aligned} t(j+1).2^{j+1} + m(j+1) &= p.2^{j+1} + m(j) + 2^j = (2p+1).2^j + m(j) \\ &= t(j).2^j + m(j) = n \quad \text{(HI)} \quad \blacksquare \end{aligned}$$

O algoritmo está correto pois, ao término do laço, $t = 0$ e passa-se da linha 4 direto para a linha 8. Pelo invariante, neste momento $n = m$.

Recorrências

Resolução de Recorrências

- Por que resolver recorrências ?
- Relações de recorrência expressam a complexidade de algoritmos recursivos como, por exemplo, os algoritmos de divisão e conquista.
- É preciso saber resolver as recorrências para que possamos efetivamente determinar a complexidade dos algoritmos recursivos.
- Existem alguns métodos para a resolução de recorrências: método iterativo, árvore de recorrência e método da substituição.
- Resolvendo algumas recorrências:
  - $T(n) = 2T(\frac{n}{2}) + 1;\ T(1) = 1.$
  - $T(n) = 2T(\frac{n}{2}) + n^2;\ T(1) = 1.$
  - $T(n) = 2T(\frac{n}{2}) + n;\ T(1) = 1.$

Resolução de Recorrências de Divisão e Conquista

- Existe uma fórmula geral para resultado de recorrências ?
- Expressão geral de recorrência de um algoritmo de divisão e conquista:

$$T(n) = aT(n/b) + f(n),$$

  onde a representa o número e n/b o tamanho dos subproblemas obtidos na divisão, e $f(n)$ é a função que dá a complexidade das etapas de divisão e de conquista.
- Vamos resolver essa recorrência para termos uma fórmula geral para o resultado de recorrências de divisão e conquista.
- Simplificando a demonstração: supor que $f(n) = cn^k$ e $n = b^m$.

Resolução de Recorrências de Divisão e Conquista

Expandindo a fórmula anterior para $T(n)$ temos:

$$\begin{aligned}
T(n) &= aT(n/b) + cn^k \\
&= a(aT(n/b^2) + c(n/b)^k) + cn^k \\
&= a(a(aT(n/b^3) + c(n/b^2)^k) + c(n/b)^k) + cn^k \\
&= \cdots \\
&= a(a(\cdots aT(n/b^m) + c(n/b^{m-1})^k) + \cdots) + cn^k \\
&= a(a(\cdots aT(1) + cb^k) + cb^{2k}) + \cdots) + cb^{mk}.
\end{aligned}$$

Resolução de Recorrências de Divisão e Conquista

Supondo $T(1) = c$, concluímos que:

$$\begin{aligned}
T(n) &= ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk} \\
&= c\sum_{i=0}^{m} a^{m-i}b^{ik} \\
&= ca^m \sum_{i=0}^{m} (b^k/a)^i.
\end{aligned}$$

Para finalizar a resolução da recorrência, temos três casos a considerar: $a > b^k$, $a = b^k$ e $a < b^k$.

Caso 1: $a > b^k$

- Neste caso, o somatório $\sum_{i=0}^{m}(b^k/a)^i$ converge para uma constante.
- daí temos que $T(n) \in \Theta(a^m)$.
- como $n = b^m$, então $m = \log_b n$ e,
- como $a^{\log_b n} = n^{\log_b a}$, concluímos que

$$T(n) \in \Theta(n^{\log_b a}).$$

Caso 2: $a = b^k$

- como $b^k/a = 1$, $\sum_{i=0}^{m}(b^k/a)^i = m + 1$.
- daí, temos que $T(n) \in \Theta(a^m m)$.
- como $m = \log_b n$ e $a = b^k$, então $a^m m = n^{\log_b a} \log_b n = n^k \log_b n$,
- o que nos leva à conclusão de que

$$T(n) \in \Theta(n^k \log n).$$

Caso 3: $a < b^k$

- Neste caso, a série não converge quando $m \to \infty$, mas pode-se calcular sua soma para um número finito de termos.

$$T(n) = ca^m \sum_{i=0}^{m} \left(b^k/a\right)^i = ca^m \left(\frac{(b^k/a)^{m+1} - 1}{(b^k/a) - 1}\right)$$

- desprezando as constantes na última linha da expressão acima e sabendo que $a^m \left(\frac{(b^k/a)^{m+1}-1}{(b^k/a)-1}\right) \in \Theta(b^{km})$ e $b^m = n$, ...
- concluímos que

$$T(n) \in \Theta(n^k).$$

Demonstramos então o seguinte Teorema:

Teorema (Teorema 3.4 do Manber)

Dada uma relação de recorrência da forma $T(n) = aT(n/b) + cn^k$, onde $a, b \in \mathbb{N}, a \geq 1, b \geq 2$ e $c, k \in \mathbb{R}^+$,

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}), & \text{se } a > b^k \\ \Theta(n^k \log n), & \text{se } a = b^k \\ \Theta(n^k), & \text{se } a < b^k \end{cases}$$

É possível generalizar esse Teorema para os casos em que $f(n)$ não é um polinômio.

Resolução de Recorrências de Divisão e Conquista

Teorema (Teorema Master (CLRS))

Sejam $a \geq 1$ e $b \geq 2$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida para os inteiros não-negativos pela relação de recorrência

$$T(n) = aT(n/b) + f(n).$$

Então $T(n)$ pode ser limitada assintoticamente da seguinte maneira:

1. *Se $f(n) \in O(n^{\log_b a-\epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) \in \Theta(n^{\log_b a})$*
2. *Se $f(n) \in \Theta(n^{\log_b a})$, então $T(n) \in \Theta(n^{\log_b a} \log n)$*
3. *Se $f(n) \in \Omega(n^{\log_b a+\epsilon})$, para algum $\epsilon > 0$ e se $af(n/b) \leq cf(n)$, para alguma constante $c < 1$ e para n suficientemente grande, então $T(n) \in \Theta(f(n))$*

Mais Exemplos de Recorrências

Exemplos onde o Teorema Master se aplica (e $f(n) \neq cn^k$):

- Caso 1: $T(n) = 4T(n/2) + n \log n$, $T(1) = 0$.
- Caso 2: $T(n) = 2T(n/2) + (n + \log n)$, $T(1) = 0$.
- Caso 3: $T(n) = T(n/2) + n \log n$, $T(1) = 0$.

Exemplos onde o Teorema Master **não se aplica**:

- $T(n) = T(n-1) + n$; $T(1) = 1$.
- $T(n) = T(n-a) + T(a) + n$; $T(b) = 1$. (para $a \geq 1$, $b \leq a$, a e b inteiros)
- $T(n) = T(\alpha n) + T((1-\alpha)n) + n$; $T(1) = 1$. (para $0 < \alpha < 1$)
- $T(n) = T(n-1) + \log n$; $T(1) = 1$.
- $T(n) = 2T(\frac{n}{2}) + n \log n$; $T(1) = 1$.

Método da substituição para resolver recorrências

A técnica:

O método da substituição envolve dois passos:

1. "Adivinhar" uma forma de solução, ou seja, uma função $g(n)$ tal que $T(n) \in \Diamond(g(n))$, onde $\Diamond \in \{\Theta, O, \Omega\}$;
2. Usar indução matemática para encontrar as *constantes* requeridas pela definição de $\Diamond$ e provar o resultado previsto.

Exemplo: resolver $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$.

- *Chute*: $T(n) \in O(n \log n)$ ...
- Problemas com as condições de contorno: E se $T(1) = 1$?
- **Conclusão**: muitas vezes a dificuldade de provar a hipótese indutiva para uma condição de contorno específica pode ser facilmente contornada tirando proveito da flexibilidade das definições da notação assintótica !

Método da substituição para resolver recorrências

Como fazer bons *chutes* ?

- Recorrências parecidas devem ter soluções parecidas ...
  Exercício: mostre que
  $T(n) = 2T(\lfloor \frac{n}{2} \rfloor + 17) + n \in O(n \log n)$.
- Fazer aproximações sucessivas dos limitantes inferiores e superiores de $T(n)$.
  Exemplo: para $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$, podíamos ter começado provando que $T(n) \in \Omega(n)$ e $T(n) \in O(n^2)$ e, em seguida, tentar apertar os limitantes por um fator de $\log n$.
- Combinar os métodos anteriores com o uso do Master:
  Exemplo: o Master **não** se aplica à recorrência
  $T(n) = 2T(\frac{n}{2}) + n \log n$ mas, eu posso dizer que
  $T'(n) \leq T(n) \leq T''(n)$ para $T'(n) = 2T'(\frac{n}{2}) + n$ e
  $T''(n) = 2T''(\frac{n}{2}) + n^2$.
  Agora resolvo $T'(n)$ e $T''(n)$ pelo Master e aplico o método de aproximações sucessivas.

Método da substituição para resolver recorrências

Algumas sutilezas ...

Resolvendo $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1$ por substituição.

Chute: $T(n) \in O(n)$, ou seja, $T(n) \leq cn$ para uma escolha apropriada de c.

$$T(n) \leq c\lfloor \frac{n}{2} \rfloor + c\lceil \frac{n}{2} \rceil + 1 = cn + 1,$$

Cuidado ! Não era isso que eu queria !

Truque: que tal tentar encontrar duas constantes positivas c e b tal que $T(n) \leq cn - b$?

Mas $cn - b < cn$, ou seja, **restringimos mais** a nossa hipótese e a prova deu certo. Não é contra-intuitivo ?

Será que eu aprendi indução ?

Método da substituição para resolver recorrências

Cuidado com as armadilhas da notação assintótica ...

Encontre o erro na prova abaixo:

$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$.
Chute: $T(n) \leq cn$, para algum $c > 0$, i.e., $T(n) \in O(n)$.
$T(n) \leq 2(c\lfloor \frac{n}{2} \rfloor) + n \leq cn + n \in O(n)$!!!!

Troca de variáveis: $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$.

Faça $m = \log n$ (i.e., $n = 2^m$): $T(2^m) = 2T(2^{m/2}) + m$ ...

Faça $S(m) = T(2^m)$ ....

Recorrências com história completa

Definição:
Uma recorrência T com **história completa** é aquela onde o valor de $T(n)$ depende de **todos** valores de $T(k)$ para $k \leq n$.

- **Exemplo 1**: $T(n) = c + \sum_{i=1}^{n-1} T(i)$ para $n \geq 2$ e $T(1) = d$, sendo c e d duas constantes **distintas**.
- Como resolver ? Escrever a expressão para $T(n+1)$, subtrair $T(n)$ cancelando termos, etc.
- Chega-se a $T(n+1) - T(n) = T(n)$ e, portanto, $T(n+1) = 2T(n)$ ou: $T(n+1) = T(1)2^n$.
- Vamos provar este último resultado por indução ?
- Que fácil !!! porém $T(2) = c + T(1) \neq 2T(1)$!
- Oh, não ! Está tudo errado ! O que aconteceu ?
- **Será que eu aprendi indução ?**

Recorrências com história completa

- **Exemplo 2**: $T(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} T(i)$, para $n \geq 2$ e $T(1) = 0$.
  (recorrência útil para a prova do caso médio do `QUICKSORT`)
- Resolvendo: escreve expressão para $T(n+1)$, multiplica por $n+1$ e subtrai do resultado a expressão de $T(n)$ multiplicada por n.
- Após cancelamento de termos, chega-se a:

$$T(n+1) = \frac{n+2}{n+1}T(n) + \frac{2n}{n+1} \leq \frac{n+2}{n+1}T(n) + 2$$

- Expandindo por i iterações, chega-se a:

$$T(n+1) \leq \frac{n+2}{n+1-i}T(n-i) + 2(n+2)\sum_{j=0}^{i} \frac{1}{n+2-j}$$

Recorrências com história completa

- Iterando-se $n - 1$ vezes e usando o fato de $T(1)$ ser nulo, chega-se a:

$$T(n+1) \leq 2(n+2)\left[\frac{1}{n+2} + \frac{1}{n+1} + \ldots + \frac{1}{3}\right]$$

- Se $H(n+2)$ é o valor da **série harmônica** calculada para $n+2$, tem-se:

$$T(n+1) \leq 2(n+2)(H(n+2) - 1.5)$$

- Logo, $T(n) \in O(n \log n)$
  Por quê ?

Indução matemática no projeto de algoritmos

Projeto de algoritmos por indução

- A seguir, usaremos a técnica de indução para desenvolver algoritmos para certos problemas.
- Isto é, a formulação do algoritmo vai ser análoga ao desenvolvimento de uma demonstração por indução.
- Assim, para resolver o problema P falamos do projeto de um algoritmo em dois passos:
  1. Exibir a resolução de uma ou mais instâncias pequenas de P (casos base);
  2. Exibir como a solução de toda instância de P pode ser obtida a partir da solução de uma ou mais instâncias menores de P.

Projeto de algoritmos por indução

Este processo indutivo resulta em algoritmos recursivos, em que:

- a base da indução corresponde à resolução dos casos base da recursão;
- a aplicação da hipótese de indução corresponde a uma ou mais chamadas recursivas; e
- o passo da indução corresponde ao processo de obtenção da resposta para o caso geral a partir daquelas retornadas pelas chamadas recursivas.

Projeto de algoritmos por indução

- Um benefício imediato é que o uso (correto) da técnica nos dá uma prova da corretude do algoritmo.
- A complexidade do algoritmo resultante é expressa numa recorrência.
- Freqüentemente o algoritmo é eficiente, embora existam exemplos simples em que isso não acontece.
- Iniciaremos com dois exemplos que usam indução fraca:
  1. cálculo do valor de polinômios e
  2. obtenção de subgrafos maximais com restrições de grau.

Exemplo 1 - Cálculo de polinômios

Problema:

Dada uma seqüência de números reais $a_n, a_{n-1}, \ldots, a_1, a_0$, e um número real x, calcular o valor do polinômio

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

Primeira solução indutiva:

- Caso base: $n = 0$. A solução é a_0.
- Suponha que para um dado n saibamos calcular o valor de

$$P_{n-1}(x) = a_{n-1} x^{n-1} + \ldots + a_1 x + a_0.$$

Então, para calcular $P_n(x)$, basta calcular x^n, multiplicar o resultado por a_n e somar o valor obtido com $P_{n-1}(x)$.

Exemplo 1 - Solução 1 - Algoritmo

CálculoPolinômio(A,x)

▷ **Entrada:** Coeficientes $A = a_n, a_{n-1}, \ldots, a_1, a_0$ e real x.
▷ **Saída:** O valor de $P_n(x)$.

1. **se** $n = 0$ **então** $P \leftarrow a_0$
2. **senão**
3. $A' \leftarrow a_{n-1}, \ldots, a_1, a_0$
4. $P' \leftarrow$ CálculoPolinômio(A', x)
5. $xn \leftarrow 1$
6. **para** $i \leftarrow 1$ **até** n **faça** $xn \leftarrow xn * x$
7. $P \leftarrow P' + a_n * xn$
8. **retorne**(P)

Exemplo 1 - Solução 1 - Complexidade

Chamando de $T(n)$ o número de operações aritméticas realizadas pelo algoritmo, temos a seguinte recorrência para $T(n)$:

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + n \text{ multiplicações} + 1 \text{ adição}, & n > 0. \end{cases}$$

Não é difícil ver que

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} (i \text{ multiplicações} + 1 \text{ adição}) \\ &= (n+1)n/2 \text{ multiplicações} + n \text{ adições}. \end{aligned}$$

Nota: o número de multiplicações pode ser diminuído calculando x^n com o algoritmo de exponenciação rápida que veremos mais tarde.

Segunda solução indutiva

- Desperdício cometido na primeira solução: re-cálculo de potências de x.
- Alternativa: eliminar essa computação desnecessária trazendo o cálculo de x^{n-1} para dentro da hipótese de indução.

Hipótese de indução reforçada:

Suponha que para um dado n saibamos calcular não só o valor de $P_{n-1}(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ mas também o valor de x^{n-1}.

- Então, no passo de indução, primeiro calculamos x^n multiplicando x por x^{n-1}, conforme exigido na hipótese. Em seguida, calculamos $P_n(x)$ multiplicando x^n por a_n e somando o valor obtido com $P_{n-1}(x)$.
- Note que para o caso base $n = 0$, a solução agora é $(a_0, 1)$.

Exemplo 1 - Solução 2 - Algoritmo

CálculoPolinômio(A,x)

▷ **Entrada:** Coeficientes $A = a_n, a_{n-1}, \ldots, a_1, a_0$ e real x.
▷ **Saída:** O valor de $P_n(x)$ e o valor de x^n.

1. **se** $n = 0$ **então** $P \leftarrow a_0; xn \leftarrow 1$
2. **senão**
3. $\quad A' \leftarrow a_{n-1}, \ldots, a_1, a_0$
4. $\quad P', x' \leftarrow$ CálculoPolinômio(A', x)
5. $\quad xn \leftarrow x * x'$
6. $\quad P \leftarrow P' + a_n * xn$
7. **retorne**(P, xn)

Exemplo 1 - Solução 2 - Complexidade

Novamente, se $T(n)$ é o número de operações aritméticas realizadas pelo algoritmo, $T(n)$ é dado por:

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 2 \text{ multiplicações} + 1 \text{ adição}, & n > 0. \end{cases}$$

A solução da recorrência é

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} (2 \text{ multiplicações} + 1 \text{ adição}) \\ &= 2n \text{ multiplicações} + n \text{ adições}. \end{aligned}$$

Terceira solução indutiva

- A escolha de considerar o polinômio $P_{n-1}(x)$ na hipótese de indução não é a única possível.
- Podemos **reforçar** ainda mais a h.i. e ter um ganho de complexidade:

Hipótese de indução mais reforçada:

Suponha que para um dado $n > 0$ saibamos calcular o valor do polinômio $P'_{n-1}(x) = a_n x^{n-1} + a_{n-1}x^{n-2} \ldots + a_1$. Então $P_n(x) = xP'_{n-1}(x) + a_0$.

- Note que, basta uma multiplicação e uma adição para obtermos $P_n(x)$ a partir de $P'_{n-1}(x)$.
- O caso base é trivial pois, para $n = 0$, a solução é a_0.

Exemplo 1 - Solução 3 - Algoritmo

CálculoPolinômio(A,x)

▷ **Entrada:** Coeficientes $A = a_n, a_{n-1}, \ldots, a_1, a_0$ e real x.
▷ **Saída:** O valor de $P_n(x)$.
1. **se** $n = 0$ **então** $P \leftarrow a_0$
2. **senão**
3. $\quad A' \leftarrow a_n, a_{n-1}, \ldots, a_1$
4. $\quad P' \leftarrow$ CálculoPolinômio(A', x)
5. $\quad P \leftarrow x * P' + a_0$
6. **retorne**(P)

Exemplo 1 - Solução 3 - Complexidade

Temos que $T(n)$ é dado por

$$T(n) = \begin{cases} 0, & n = 0 \\ T(n-1) + 1 \text{ multiplicação} + 1 \text{ adição}, & n > 0. \end{cases}$$

A solução é

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} (1 \text{ multiplicação} + 1 \text{ adição}) \\ &= n \text{ multiplicações} + n \text{ adições}. \end{aligned}$$

Essa forma de calcular $P_n(x)$ é chamada de **regra de Horner**.

Projeto por indução - Exemplo 2 Subgrafos Maximais

- Suponha que você esteja planejando uma festa onde queira maximizar as interações entre as pessoas. Uma festa animada, mas sem recursos ilícitos para aumentar a animação.
- Uma das maneiras de conseguir isso é fazer uma lista dos possíveis convidados e, para cada um desses, a lista dos convidados com quem ele(a) interagiria durante a festa.
- Em seguida, é só localizar o maior subgrupo de convidados que interagiriam com, no mínimo, digamos, k outros convidados dentro do subgrupo.

Exemplo 2 - Subgrafos Maximais

Formulação do problema usando grafos:

Dado um grafo simples G (não-orientado) com n vértices e um inteiro $k \leq n$, encontrar em G um subgrafo induzido H com o número máximo de vértices, tal que o grau de cada vértice de H seja $\geq k$. Se um tal subgrafo H não existir, o algoritmo deve identificar esse fato.

Definições:

Se H é um *subgrafo induzido* de um grafo G, uma aresta de G está em H se, e somente se, tem ambos os seus extremos em H.
Um subgrafo qualquer G' de G é *maximal* com relação a uma propriedade P se G' satisfizer P e não existir em G outro subgrafo próprio que contenha G' e satisfaça P.

Exemplo 2 (cont.)

- Exemplo de um subgrafo que é induzido e de outro subgrafo que não é induzido em um grafo G:
- Como exemplo de subgrafos maximais (propriedade relativa à continência) que não são máximos (propriedade relativa ao tamanho) em um grafo G, veja definição de clique em grafos.
- No nosso exemplo, o conceito de *maximal* é equivalente ao de *máximo* (número de vértices).

  Você pode provar esta afirmativa ?

Exemplo 2 - Indução

Usando o método indutivo:

Hipótese de indução:
Dados um grafo G e um inteiro n qualquer, sabemos como encontrar subgrafos maximais H de G com grau mínimo $\geq k$, desde que G tenha menos que n vértices.

Caso base: o primeiro valor de n para o qual faz sentido buscarmos tais subgrafos H é $n = k + 1$, caso contrário não há como satisfazer a restrição de grau mínimo.

Assim, quando $n = k + 1$, a única forma de existir em G um subgrafo induzido com grau mínimo k é que **todos** os vértices tenham grau igual a k.

Se isso ocorrer, a resposta é $H = G$; caso contrário $H = \emptyset$.

Exemplo 2 - Indução (cont.)

- Seja então G um grafo com n vértices e $k \geq 0$ um inteiro tal que $n > k + 1$.
- Se todos os vértices de G têm grau $\geq k$ não há nada mais a fazer. Retorne $H = G$ e fim.
- Caso contrário, seja v um vértice com grau $< k$. É fácil ver que nenhum subgrafo que é solução para o problema conterá v. Assim, v pode ser removido e a hipótese de indução aplicada ao grafo resultante G'.
- Seja H' o subgrafo retornado pela aplicação da h.i. em G'. Então H' também é resposta para G. ■

Exemplo 2 - Algoritmo

SubgrafoMaximal(G, k)

▷ **Entrada:** Grafo G com n vértices e um inteiro $k \geq 0$.
▷ **Saída:** Subgrafo induzido H de G com grau mínimo k.
1. **se** $(n < k + 1)$ **então** $H \leftarrow \emptyset$
2. **senão se** todo vértice de G tem grau $\geq k$
3. **então** $H \leftarrow G$
4. **senão**
5. seja v um vértice de G com grau $< k$
6. $H \leftarrow$ SubgrafoMaximal$(G - v, k)$
7. **retorne**(H)

Projeto por indução - Exemplo 3
Fatores de balanceamento em árvores binárias

Definição:

Árvores binárias balanceadas são estruturas de dados que minimizam o tempo de busca de informações nela armazenadas. A idéia é que, para todo nó v da árvore, o *fator de balanceamento (f.b.)* de v, isto é, a diferença entre a altura da subárvore esquerda e a altura da subárvore direita de v não desvie muito de zero.

- Árvores AVL são um exemplo de árvores binárias balanceadas, em que o f.b. de cada nó é $-1, 0$ ou $+1$
- Veja um exemplo de uma árvore binária e os f.b.s de seus nós.

Exemplo 3 (cont.)

Problema:

Dada uma árvore binária A com n nós, calcular os fatores de balanceamento de cada nó de A.

- Vamos projetar o algoritmo indutivamente.

Hipótese de indução:

Para um valor de $n > 0$ qualquer, sabemos como calcular fatores de balanceamento de árvores com menos que n nós.

- **Caso base**: quando $n = 0$, convencionamos que o f.b. é igual a zero.
- Vamos mostrar agora como usar a hipótese para calcular f.b.s de uma árvore A com exatamente n nós.

Exemplo 3 - indução (cont.)

A idéia é aplicar a h.i. às subárvores esquerda e direita da raiz e, em seguida, calcular o f.b. da raiz.

Dificuldade: o f.b. da raiz depende das alturas das subárvores esquerda e direita e não dos seus f.b.s.

Conclusão: é necessário uma h.i. mais forte !

Nova hipótese de indução:

Para um valor de $n > 0$ qualquer, sabemos como calcular fatores de balanceamento e alturas de árvores com menos que n nós.

Exemplo 3 - indução (cont.)

- Novamente, o caso base $n = 0$ é fácil. O f.b. e a altura são ambos iguais a zero.
- Seja A uma árvore binária com n nós, para um $n > 0$ qualquer.

  Sejam (f_e, h_e) e (f_d, h_d) os f.b.s e alturas das subárvores esquerda (A_e) e direita (A_d) de A que, por h.i., sabemos como calcular.

  Então, o f.b. da raiz de A é $h_e - h_d$ e a altura da árvore é $\max(h_e, h_d) + 1$.

  Isto completa o cálculo dos f.b.s e da altura de A. ■

De novo, o fortalecimento da hipótese tornou a resolução do problema mais fácil.

Exemplo 3 - Algoritmo

FatorAltura (A)

$\triangleright$ **Entrada:** Uma árvore binária A com $n \geq 0$ nós
$\triangleright$ **Saída:** A árvore A os f.b.s nos seus nós e a altura de A
1. **se** $n = 0$ **então** $h \leftarrow 0$
2. **senão**
3. seja r a raiz de A
3. $h_e \leftarrow$ FatorAltura(A_e)
4. $h_d \leftarrow$ FatorAltura(A_d)
5. $\triangleright$ armazena o f.b. na raiz em $r.f$
6. $r.f \leftarrow h_e - h_d$
7. $h \leftarrow \max(h_e, h_d) + 1$
8. **retorne**(h)

Exemplo 3 - Complexidade

- Seja $T(n)$ o número de operações executadas pelo algoritmo para calcular os f.b.s e a altura de uma árvore A de n nós. Então

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(n_e) + T(n_d) + c_2, & n > 0, \end{cases}$$

onde:
  - n_e, n_d são os números de nós das subárvores esquerda e direita;
  - c_1 é uma constante que denota o tempo (constante) para a execução dos comandos de atribuição do caso base; e
  - c_2 é outra constante que denota o tempo para a execução das operações aritméticas e atribuições do caso geral.
- Em vez de escrever c_1 e c_2 poderíamos ter usado a notação $\Theta(1)$.

Exemplo 3 - Complexidade

O pior caso da recorrência parece ser quando, ao longo da recursão, um de n_e, n_d é sempre igual a zero (e o outro é sempre igual a $n - 1$).

Chega-se então a:

$$T(n) = \sum_{i=1}^{n} (c_1 + c_2) = n(c_1 + c_2) \in \Theta(n).$$

Exercício:

Há algum ganho em complexidade quando ambos n_e e n_d são aproximadamente $n/2$ ao longo da recursão ?

Projeto por Indução - Exemplo 4: o problema da celebridade

Num conjunto S de n pessoas, uma *celebridade* é alguém que é conhecido por todas as pessoas de S mas que não conhece ninguém. Isso implica que pode existir somente uma celebridade em S

Problema:

Dado um conjunto de n pessoas e uma matriz M $n \times n$ onde $M[i, j] = 1$ se a pessoa i conhece a pessoa j e $M[i, j] = 0$ caso contrário, encontrar (se existir) uma celebridade no conjunto. Isto é, determinar um k tal que todos os elementos da coluna k (exceto $M[k, k]$) de M são 1s e todos os elementos da linha k (exceto $M[k, k]$) são 0s.

Exemplo 4 - Indução

Existe uma solução simples mas laboriosa: Para cada i, verifique todos os outros elementos da linha i e da coluna i. O custo dessa solução é $2n(n-1)$.

Um argumento indutivo que parece ser mais eficiente é o seguinte:

- Se $n = 1$, podemos considerar que o único elemento é uma celebridade.
- Outra opção seria considerarmos o caso base como $n = 2$, o primeiro caso interessante. A solução é simples: existe uma celebridade se, e somente se, $M[1,2] \oplus M[2,1] = 1$. Mais uma comparação define a celebridade: se $M[1,2] = 0$, então a celebridade é a pessoa 1; senão, é a pessoa 2.

Hipótese de Indução:

Para um n qualquer, $n > 2$, sabemos encontrar uma celebridade num conjunto de $n-1$ pessoas.

Exemplo 4 - Indução (cont.)

Tome então um conjunto $S = \{1, 2, \ldots, n\}$, $n > 2$, de pessoas e a matriz M correspondente. Considere o conjunto $S' = S \setminus \{n\}$;

Há dois casos possíveis:

1. Existe uma celebridade em S', digamos a pessoa k; então, k é celebridade em S se, e somente se, $M[n,k] = 1$ e $M[k,n] = 0$.
2. Se k não for celebridade em S ou não existir celebridade em S', então a pessoa n é celebridade em S se $M[n,j] = 0$ e $M[j,n] = 1, \forall j < n$; caso contrário não há celebridade em S.

Essa primeira tentativa, infelizmente, também conduz a um algoritmo quadrático. (Por quê ?)

Exemplo 4 - Segunda tentativa

A segunda tentativa baseia-se em um fato muito simples:

Dadas duas pessoas i e j, é possível determinar se uma delas ***não é*** *uma celebridade com apenas uma comparação: se $M[i,j] = 1$, então i não é celebridade; caso contrário j não é celebridade.*

Vamos usar esse argumento aplicando a hipótese de indução sobre o conjunto de $n-1$ pessoas obtidas removendo dentre as n uma pessoa que sabemos não ser celebridade.

- O caso base e a hipótese de indução são os mesmos que anteriormente.

Exemplo 4 - Segunda tentativa

Tome então um conjunto arbitrário de $n > 2$ pessoas e a matriz M correspondente.

Sejam i e j quaisquer duas pessoas e suponha que, usando o argumento acima, determinemos que j não é celebridade.

Seja $S' = S \setminus \{j\}$ e considere os dois casos possíveis:

1. Existe uma celebridade em S', digamos a pessoa k. Se $M[j,k] = 1$ e $M[k,j] = 0$, então k é celebridade em S; caso contrário não há uma celebridade em S.
2. Não existe uma celebridade em S'; então não existe uma celebridade em S.

Exemplo 4 - Algoritmo

Celebridade(S, M)

▷ **Entrada:** conjunto de pessoas $S = \{1, 2, \ldots, n\}$;
M, a matriz que define quem conhece quem em S.
▷ **Saída:** Um inteiro $k \le n$ que é celebridade em S ou $k = 0$

1. **se** $|S| = 1$ **então** $k \leftarrow$ elemento em S
2. **senão**
3. sejam i, j quaisquer duas pessoas em S
4. **se** $M[i,j] = 1$ **então** $s \leftarrow i$ **senão** $s \leftarrow j$
5. $S' \leftarrow S \setminus \{s\}$
6. $k \leftarrow$ Celebridade(S', M)
7. **se** $k > 0$ **então**
8. **se** $(M[s,k] \neq 1)$ **ou** $(M[k,s] \neq 0)$ **então** $k \leftarrow 0$
9. **retorne**(k)

Exemplo 4 - Complexidade

O algoritmo resultante tem complexidade linear em n.

A recorrência $T(n)$ para o número de operações executadas pelo algoritmo é:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(n-1) + \Theta(1), & n > 1. \end{cases}$$

A solução desta recorrência é

$$\sum_{1}^{n} \Theta(1) = n\Theta(1) = \Theta(n).$$

Projeto por indução - Exemplo 5
Máxima subseqüência consecutiva (m.s.c)

Um dos problemas corriqueiros na área de processamento de imagens é a busca de trechos de pontos (pixels) contíguos cuja soma de pesos seja máximo. A função peso varia com a aplicação.

Problema: (versão simplificada)

Dada uma seqüência $X = x_1, x_2, \ldots, x_n$ de números inteiros (não necessariamente positivos), encontrar uma **subseqüência consecutiva** *$Y = x_i, x_{i+1}, \ldots, x_j$ de X, onde $1 \le i, j \le n$, cuja soma seja máxima dentre todas as subseqüências consecutivas.*

Exemplos:

$X = [3, 0, -1, 2, 4, -1, 2, -2]$.	Resp: $Y = [3, 0, -1, 2, 4, -1, 2]$
$X = [-1, -2, 0]$.	Resp: $Y = [0]$
$X = [-3, -1]$.	Resp: $Y = [\quad]$

Exemplo 5 - Indução

Como antes, vamos examinar o que podemos obter de uma **hipótese de indução** simples:

Hipótese de Indução simples:

Para um n qualquer, sabemos calcular a m.s.c. de seqüências cujo comprimento seja menor que n.

- Seja então $X = x_1, x_2, \ldots, x_n$ uma seqüência qualquer de comprimento $n > 1$.
- Considere a seqüência X' obtida removendo-se de X o número x_n.
- Seja $Y' = x_i, x_{i+1}, \ldots, x_j$ a m.s.c. de X', obtida aplicando-se a h.i.

Exemplo 5 - Indução

Há três casos a examinar:

1. $Y' = [\ \ ]$. Neste caso, $Y = x_n$ se $x_n \geq 0$ ou $Y = [\ \ ]$ se $x_n < 0$.
2. $j = n - 1$. Como no caso anterior, temos $Y = Y' || x_n$ se $x_n \geq 0$ ou $Y = Y'$ se $x_n < 0$.
3. $j < n - 1$. Aqui há dois subcasos a considerar:
   1. Y' também é m.s.c. de X; isto é, $Y = Y'$.
   2. Y' não é a m.s.c. de X. Isto significa que x_n é parte da m.s.c. Y de X, que é, portanto, da forma $x_k, x_{k+1}, \ldots, x_{n-1}, x_n$, para algum $k < n - 1$.

Exemplo 5 - Indução

- A hipótese de indução nos permite resolver todos os casos anteriores, exceto o último.
  Não há informação suficiente na h.i. para permitir a resolução deste caso.

  O que falta na h.i. ?
- É evidente que, quando

$$Y = x_k, x_{k+1}, \ldots, x_{n-1}, x_n,$$

então $x_k, x_{k+1}, \ldots, x_{n-1}$ é um *sufixo* de X' de máximo valor entre os *sufixos* de X'.
- Assim, se conhecermos o sufixo máximo (em valor) de X', além da m.s.c., teremos resolvido o problema completamente para X.

Exemplo 5 - Indução

Parece então natural enunciar a seguinte h.i. fortalecida:

Hipótese de indução reforçada:

Para um n qualquer, sabemos calcular a m.s.c. e o sufixo máximo de seqüências cujo comprimento seja menor que n.

É clara desta discussão também, a base da indução: para $n = 1$, a m.s.c. de $X = x_1$ é x_1 caso $x_1 \geq 0$, e a seqüência vazia caso contrário.

Exemplo 5 - Algoritmo

MSC(X, n): (cont.)

▷ **Entrada:** Uma seqüência $X = [x_1, x_2, \ldots, x_n]$ de números reais.
▷ **Saída:** Inteiros i, j, k e reais $MaxSeq, MaxSuf$ tais que:
– x_i, x_j são o primeiro e último elementos da m.s.c. de X, cujo valor é $MaxSeq$; e
– x_k é o primeiro elemento do sufixo máximo de X, cujo valor é $MaxSuf$.
– O valor $j = 0$ significa que X é composta de negativos somente. Neste caso, convencionamos $MaxSeq = 0$.
– O valor $k = 0$ significa que o máximo sufixo de X tem soma negativa. Neste caso, $MaxSuf = 0$.

Exemplo 5 - Algoritmo (cont.)

MSC(X, n): (cont.)

1. **se** $n = 1$
2. **então**
3. **se** $x_1 < 0$
4. **então** $i, j, k \leftarrow 0$; $MaxSeq, MaxSuf \leftarrow 0$
5. **senão** $i, j, k \leftarrow 1$; $MaxSeq, MaxSuf \leftarrow x_1$
6. **senão**
7. $i, j, k, MaxSeq, MaxSuf \leftarrow \text{MSC}(X - x_n)$
8. **se** $k = 0$ **então** $k \leftarrow n$
9. $MaxSuf \leftarrow MaxSuf + x_n$
10. **se** $MaxSuf > MaxSeq$
11. **então** $i \leftarrow k; j \leftarrow n$; $MaxSeq \leftarrow MaxSuf$
12. **senão se** $MaxSuf < 0$ **então** $MaxSuf \leftarrow 0; k \leftarrow 0$
13. **retorne**($i, j, k, MaxSeq, MaxSuf$)

Exemplo 5 - Complexidade

A complexidade $T(n)$ de MSC é simples de ser calculada.

Como no último exemplo,

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(n-1) + \Theta(1), & n > 1. \end{cases}$$

A solução desta recorrência é

$$\sum_{1}^{n} \Theta(1) = n\Theta(1) = \Theta(n).$$

Reforçar a hipótese de indução: preciso lembrar disso ...

Projeto por indução - Erros comuns

Os erros discutidos nas provas por indução, naturalmente, traduzem-se em erros no projeto de um algoritmo por indução.

Exemplo:

Problema: *Dado um grafo conexo G, verificar se G é bipartido ou não. Caso seja, retornar a partição dos vértices.*

Definição:

Um grafo é *bipartido* se seu conjunto de vértices pode ser particionado em dois conjuntos, de forma que toda aresta de G tenha extremos em conjuntos diferentes. Se G é conexo e bipartido, então a partição é única.

Projeto por indução - Erros comuns

O que há de **errado** com o (esboço de um) algoritmo recursivo mostrado abaixo ?

- Sejam G um grafo conexo, v um vértice de G e considere o grafo $G' = G - \{v\}$.
- Se G' não for bipartido, então G também não é. Caso contrário, sejam A e B os dois conjuntos da partição de G' obtidos recursivamente.
- Considere agora o vértice v e sua relação com os vértices de A e B.
- Se v tiver um vizinho em A e outro em B, então G não é bipartido (já que a partição, se existir, deve ser única).
- Caso contrário, adicione v a A ou B, o conjunto no qual v não tem vizinhos. A partição está completa.

Divisão e Conquista

Projeto de Algoritmos por Divisão e Conquista

- **Dividir para conquistar:** uma tática de guerra aplicada ao projeto de algoritmos.
- Um algoritmo de divisão e conquista é aquele que resolve o problema desejado combinando as soluções parciais de (um ou mais) subproblemas, obtidas recursivamente.
- É mais um paradigma de projeto de algoritmos baseado no princípio da indução.
- Informalmente, podemos dizer que o paradigma incremental representa o projeto de algoritmos por indução fraca, enquanto o paradigma de divisão e conquista representa o projeto por indução forte.
- É natural, portanto, demonstrar a corretude de algoritmos de divisão e conquista por indução.

Algoritmo Genérico

DivisaoConquista(x)

▷ **Entrada:** A instância x
▷ **Saída:** Solução y do problema em questão para x
1. **se** x é suficientemente pequeno **então**
   ▷ *Solucao*(x) algoritmo para pequenas instâncias
2. **retorne** *Solucao*(x)
3. **senão**
   ▷ divisão
4. decomponha x em instâncias menores $x_1, x_2, \cdots, x_k$
5. **para** i **de** 1 **até** k **faça** $y_i :=$ *DivisaoConquista*(x_i)
   ▷ conquista
6. combine as soluções y_i para obter a solução y de x.
7. **retorne**(y)

Projeto por Divisão e Conquista - Exemplo 1 Exponenciação

Problema:
Calcular a^n, para todo real a e inteiro $n \geq 0$.

Primeira solução, por indução fraca:

- **Caso base:** $n = 0$; $a^0 = 1$.
- **Hipótese de indução:** *Suponha que, para qualquer inteiro $n > 0$ e real a, sei calcular a^{n-1}.*
- **Passo da indução:** Queremos provar que conseguimos calcular a^n, para $n > 0$. Por hipótese de indução, sei calcular a^{n-1}. Então, calculo a^n multiplicando a^{n-1} por a.

Exemplo 1 - Solução 1 - Algoritmo

Exponenciacao(a, n)

▷ **Entrada:** A base a e o expoente n.
▷ **Saída:** O valor de a^n.
1. **se** $n = 0$ **então**
2. **retorne**(1) {caso base}
3. **senão**
4. $an' := Exponenciacao(a, n-1)$
5. $an := an' * a$
6. **retorne**(an)

Exemplo 1 - Solução 1 - Complexidade

Seja $T(n)$ o número de operações executadas pelo algoritmo para calcular a^n.

Então a relação de recorrência deste algoritmo é:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(n-1) + c_2, & n > 0, \end{cases}$$

onde c_1 e c_2 representam, respectivamente, o tempo (constante) executado na atribuição da base e multiplicação do passo.

Neste caso, não é difícil ver que

$$T(n) = c_1 + \sum_{i=1}^{n} c_2 = c_1 + nc_2 = \Theta(n).$$

Este algoritmo é linear no tamanho da entrada ?

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Exemplo 1 - Solução 2 - Divisão e Conquista

Vamos agora projetar um algoritmo para o problema usando indução forte de forma a obter um algoritmo de divisão e conquista.

Segunda solução, por indução forte:

- **Caso base:** $n = 0$; $a^0 = 1$.
- **Hipótese de indução:** *Suponha que, para qualquer inteiro $n > 0$ e real a, sei calcular a^k, para todo $k < n$.*
- **Passo da indução:** Queremos provar que conseguimos calcular a^n, para $n > 0$. Por hipótese de indução sei calcular $a^{\lfloor \frac{n}{2} \rfloor}$. Então, calculo a^n da seguinte forma:

$$a^n = \begin{cases} \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2, & \text{se } n \text{ par;} \\ a \cdot \left(a^{\lfloor \frac{n}{2} \rfloor}\right)^2, & \text{se } n \text{ ímpar.} \end{cases}$$

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Exemplo 1 - Solução 2 - Algoritmo

ExponenciacaoDC(a, n)

▷ **Entrada:** A base a e o expoente n.
▷ **Saída:** O valor de a^n.
1. **se** $n = 0$ **então**
2. **retorne**(1) {caso base}
3. **senão**
▷ divisão
4. $an' := ExponenciacaoDC(a, n \ div \ 2)$
▷ conquista
5. $an := an' * an'$
6. **se** $(n \ mod \ 2) = 1$
7. $an := an * a$
8. **retorne**(an)

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Exemplo 1 - Solução 2 - Complexidade

- Seja $T(n)$ o número de operações executadas pelo algoritmo de divisão e conquista para calcular a^n.
- Então a relação de recorrência deste algoritmo é:

$$T(n) = \begin{cases} c_1, & n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + c_2, & n > 0, \end{cases}$$

- Não é difícil ver que $T(n) \in \Theta(\log n)$. Por quê?

Projeto por Divisão e Conquista - Exemplo 2 Busca Binária

Problema:
Dado um vetor ordenado A com n números reais e um real x, determinar a posição $1 \leq i \leq n$ tal que $A[i] = x$, ou que não existe tal i.

- O projeto de um algoritmo para este problema usando indução simples, nos leva a um algoritmo incremental de complexidade de pior caso $\Theta(n)$. Pense em como seria a indução !
- Se utilizarmos indução forte para projetar o algoritmo, podemos obter um algoritmo de divisão e conquista que nos leva ao algoritmo de busca binária. Pense na indução !
- Como o vetor está ordenado, conseguimos determinar, com apenas uma comparação, que *metade* das posições do vetor não pode conter o valor x.

Exemplo 2 - Algoritmo

BuscaBinaria(A, e, d, x)

▷ **Entrada:** Vetor A, delimitadores e e d do subvetor e x.
▷ **Saída:** Índice $1 \leq i \leq n$ tal que $A[i] = x$ ou $i = 0$.
1. **se** $e = d$ **então se** $A[e] = x$ **então retorne**(e)
2. **senão retorne**(0)
3. **senão**
4. $i := (e + d)$ *div* 2
5. **se** $A[i] = x$ **retorne**(i)
6. **senão se** $A[i] > x$
7. $i := BuscaBinaria(A, e, i - 1, x)$
8. **senão** $\{A[i] < x\}$
9. $i := BuscaBinaria(A, i + 1, d, x)$
10. **retorne**(i)

Exemplo 2 - Complexidade

- O número de operações $T(n)$ executadas na busca binária no pior caso é:

$$T(n) = \begin{cases} c_1, & n = 1 \\ T(\lceil \frac{n}{2} \rceil) + c_2, & n > 1, \end{cases}$$

- Não é difícil ver que $T(n) \in \Theta(\log n)$. Por quê?
- O algoritmo de busca binária (divisão e conquista) tem complexidade de pior caso $\Theta(\log n)$, que é assintoticamente melhor que o algoritmo de busca linear (incremental).
- E se o vetor não estivesse ordenado, qual paradigma nos levaria a um algoritmo assintoticamente melhor ?

Projeto por Divisão e Conquista - Exemplo 3 - Máximo e Mínimo

Problema:
Dado um conjunto S de $n \geq 2$ números reais, determinar o maior e o menor elemento de S.

- Um algoritmo incremental para esse problema faz $2n - 3$ comparações: fazemos uma comparação no caso base e duas no passo.
- Será que um algoritmo de divisão e conquista seria melhor ?
- Um possível algoritmo de divisão e conquista seria:
  - Divida S em dois subconjuntos de mesmo tamanho S_1 e S_2 e solucione os subproblemas.
  - O máximo de S é o máximo dos máximos de S_1 e S_2 e o mínimo de S é o mínimo dos mínimos de S_1 e S_2.

Exemplo 3 - Complexidade

- Qual o número de comparações $T(n)$ efetuado por este algoritmo?

$$T(n) = \begin{cases} 1, & n = 2 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2, & n > 2, \end{cases}$$

- Supondo que n é uma potência de 2, temos:

$$T(n) = \begin{cases} 1, & n = 2 \\ 2T(\frac{n}{2}) + 2, & n > 2, \end{cases}$$

- Neste caso, podemos provar que $T(n) = \frac{3}{2}n - 2$ usando o método da substituição (indução !).

Exemplo 3 - Complexidade

- **Caso Base:** $T(2) = 1 = 3 - 2$.
- **Hipótese de Indução:** Suponha, para $n = 2^{k-1}$, $k \geq 2$, que $T(n) = \frac{3}{2}n - 2$.
- **Passo de Indução:** Queremos provar para $n = 2^k$, $k \geq 2$, que $T(n) = \frac{3}{2}n - 2$.

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + 2 \\ &= 2(\tfrac{3}{4}n - 2) + 2 \text{ (por h. i.)} \\ &= \tfrac{3}{2}n - 2. \end{aligned}$$

- É possível provar que $T(n) = \frac{3}{2}n - 2$ quando n não é potência de 2 ?

Exemplo 3 - Complexidade

- Assintoticamente, os dois algoritmos para este problema são equivalentes, ambos $\Theta(n)$.
- No entanto, o algoritmo de divisão e conquista permite que menos comparações sejam feitas. A estrutura hierárquica de comparações no retorno da recursão evita comparações desnecessárias.

Ordenação

O Problema da Ordenação

Problema:

Ordenar um conjunto de $n \geq 1$ inteiros.

- Podemos projetar por indução diversos algoritmos para o problema da ordenação.
- Na verdade, todos os algoritmos básicos de ordenação surgem de projetos por indução sutilmente diferentes.
- Começaremos usando indução simples no projeto do algoritmo de ordenação.

Projeto por Indução Simples

Hipótese de Indução Simples:

Sabemos ordenar um conjunto de $n - 1 \geq 1$ inteiros.

- **Caso base:** $n = 1$. Um conjunto de um único elemento está ordenado.
- **Passo da Indução (Primeira Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x um elemento qualquer de S. Por hipótese de indução, sabemos ordenar o conjunto $S - x$, basta então inserir x na posição correta para obtermos S ordenado.
- Esta indução dá origem ao algoritmo incremental *Insertion Sort*.

Insertion Sort - Pseudo-código - Versão Recursiva

OrdenacaoInsercao(A, n)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.
1. **se** $n \geq 2$ **faça**
2. OrdenacaoInsercao($A, n - 1$)
3. $v := A[n]$
4. $j := n - 1$
5. **enquanto** $(j > 0)$ e $(A[j] > v)$ **faça**
6. $A[j + 1] := A[j]$
7. $j := j - 1$
8. $A[j + 1] := v$

Insertion Sort - Pseudo-código - Versão Iterativa

- É fácil eliminar o uso de recursão simulando-a com um laço.

OrdenacaoInsercao(A)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.
1. **para** $i := 2$ **até** n **faça**
2. $v := A[i]$
3. $j := i - 1$
4. **enquanto** $(j > 0)$ e $(A[j] > v)$ **faça**
5. $A[j+1] := A[j]$
6. $j := j - 1$
7. $A[j+1] := v$

Insertion Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Insertion Sort* executa no pior caso ?
- Tanto o número de comparações quanto o de trocas é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n, & n > 1, \end{cases}$$

- Portanto, $\Theta(n^2)$ comparações e trocas são executadas no pior caso.

Projeto por Indução Simples - Segunda Alternativa

Hipótese de Indução Simples:
Sabemos ordenar um conjunto de $n - 1 \geq 1$ inteiros.

- **Caso base:** $n = 1$. Um conjunto de um único elemento está ordenado.
- **Passo da Indução (Segunda Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x o menor elemento de S. Então x certamente é o primeiro elemento da seqüência ordenada de S e basta ordenarmos os demais elementos de S. Por hipótese de indução, sabemos ordenar o conjunto $S - x$ e assim obtemos S ordenado.
- Esta indução dá origem ao algoritmo incremental *Selection Sort*.

Selection Sort - Pseudo-código - Recursiva

OrdenacaoSelecao(A, i, n)

▷ **Entrada:** Vetor A de n números inteiros e os índices de início e término da seqüência a ser ordenada.
▷ **Saída:** Vetor A ordenado.
1. **se** $i < n$ **faça**
2. $min := i$
3. **para** $j := i + 1$ **até** n **faça**
4. **se** $A[j] < A[min]$ **então** $min := j$
5. $t := A[min]$
6. $A[min] := A[i]$
7. $A[i] := t$
8. OrdenacaoSelecao($A, i + 1, n$)

Selection Sort - Pseudo-código - Versão Iterativa

- Novamente, é fácil eliminar o uso de recursão simulando-a com um laço.

OrdenacaoSelecao(A)

```
▷ Entrada: Vetor A de n números inteiros.
▷ Saída: Vetor A ordenado.
1.  para i := 1 até n − 1 faça
2.      min := i
3.      para j := i + 1 até n faça
4.          se A[j] < A[min] então min := j
5.      t := A[min]
6.      A[min] := A[i]
7.      A[i] := t
```


Selection Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Selection Sort* executa no pior caso ?
- O número de comparações é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n, & n > 1, \end{cases}$$

- Portanto, $\Theta(n^2)$ comparações são executadas no pior caso.

Selection Sort - Análise de Complexidade - Cont.

- Já o número de trocas é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + 1, & n > 1, \end{cases}$$

- Portanto, $\Theta(n)$ trocas são executadas no pior caso.
- Apesar dos algoritmos *Insertion Sort* e *Selection Sort* terem a mesma complexidade assintótica, em situações onde a operação de troca é muito custosa, é preferível utilizar *Selection Sort*.

Projeto por Indução Simples - Terceira Alternativa

- Ainda há uma terceira alternativa para o passo da indução.
- **Passo da Indução (Terceira Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x o maior elemento de S. Então x certamente é o último elemento da seqüência ordenada de S e basta ordenarmos os demais elementos de S. Por hipótese de indução, sabemos ordenar o conjunto $S - x$ e assim obtemos S ordenado.
- Em princípio, esta indução dá origem a uma variação do algoritmo *Selection Sort*.
- No entanto, se implementamos de uma forma diferente a seleção e o posicionamento do maior elemento, obteremos o algoritmo *Bubble Sort*.

Bubble Sort - Pseudo-código - Versão Iterativa

BubbleSort(A)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.

```
1.  para i := n decrescendo até 1 faça
2.      para j := 2 até i faça
3.          se A[j − 1] > A[j] então
4.              t := A[j − 1]
5.              A[j − 1] := A[j]
6.              A[j] := t
```


Bubble Sort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Bubble Sort* executa no pior caso ?
- O número de comparações e de trocas é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n, & n > 1, \end{cases}$$

- Portanto, $\Theta(n^2)$ comparações e trocas são executadas no pior caso.
- Ou seja, algoritmo *Bubble Sort* executa mais trocas que o algoritmo *Selection Sort* !

Projeto por Indução Forte

- Também podemos usar indução forte para projetar algoritmos para o problema da ordenação.

Hipótese de Indução Forte:
Sabemos ordenar um conjunto de $1 \le k < n$ inteiros.

- **Caso base:** $n = 1$. Um conjunto de um único elemento está ordenado.
- **Passo da Indução (Primeira Alternativa):** Seja S um conjunto de $n \ge 2$ inteiros. Podemos particionar S em dois conjuntos, S_1 e S_2, de tamanhos $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$. Como $n \ge 2$, ambos S_1 e S_2 possuem menos de n elementos. Por hipótese de indução, sabemos ordenar os conjuntos S_1 e S_2. Podemos então obter S ordenado intercalando os conjuntos ordenados S_1 e S_2.

Projeto por Indução Forte

- Esta indução dá origem ao algoritmo de divisão e conquista *Mergesort*.
- Na implementação do algoritmo, o conjunto S é um vetor de tamanho n.
- A operação de divisão é imediata, o vetor é dividido em dois vetores com metade do tamanho do original, que são ordenados recursivamente.
- O trabalho do algoritmo está concentrado na conquista: a intercalação dos dois subvetores ordenados.
- Para simplificar a implementação da operação de intercalação e garantir sua complexidade linear, usamos um vetor auxiliar.

Mergesort - Pseudo-código

OrdenacaoIntercalacao(A, e, d)

▷ **Entrada:** Vetor A de n números inteirosos índices e e d que delimitam início e fim do subvetor a ser ordenado.
▷ **Saída:** Subvetor de A de e a d ordenado.

```
01.  se d > e então
02.      m := (e + d) div 2
03.      OrdenacaoIntercalacao(A, e, m)
04.      OrdenacaoIntercalacao(A, m + 1, d)
```


Mergesort - Pseudo-código (cont.)

OrdenacaoIntercalacao(A, e, d)**:cont.**

```
     ▷ Copia o trecho de e a m para B.
05.     para i de e até m faça
06.         B[i] := A[i]
     ▷ Copia o trecho de m a d para B em ordem reversa.
07.     para j de m + 1 até d faça
08.         B[d + m + 1 − j] := A[j]
09.     i := e; j := d
10.     para k de e até d faça
11.         se B[i] < B[j] então
12.             A[k] := B[i]
13.             i := i + 1
14.         senão
15.             A[k] := B[j]
16.             j := j − 1
```


Mergesort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Mergesort* executa no pior caso ?
- A etapa de intercalação tem complexidade $\Theta(n)$, logo, o número de comparações e de trocas é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n, & n > 1, \end{cases}$$

- Portanto, $\Theta(n \log n)$ comparações e trocas são executadas no pior caso.
- Ou seja, algoritmo *Mergesort* é assintoticamente mais eficiente que todos os anteriores.
- Em contrapartida, o algoritmo *Mergesort* usa o dobro de memória. Ainda assim, assintoticamente o gasto de memória é equivalente ao dos demais algoritmos.

Mergesort - Análise de Complexidade

- É possível fazer a intercalação dos subvetores ordenados sem o uso de vetor auxiliar ?
- Sim! Basta deslocarmos os elementos de um dos subvetores, quando necessário, para dar lugar ao mínimo dos dois subvetores.
- No entanto, a etapa de intercalação passa a ter complexidade $\Theta(n^2)$, resultando na seguinte recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n^2, & n > 1, \end{cases}$$

- Ou seja, a complexidade do *Mergesort* passa a ser $\Theta(n^2)$!
- Como era de se esperar, a eficiência da etapa de intercalação é crucial para a eficiência do *Mergesort*.

Projeto por Indução Forte - Segunda Alternativa

Hipótese de Indução Forte:
Sabemos ordenar um conjunto de $1 \leq k < n$ inteiros.

- **Caso base:** $n = 1$. Um conjunto de um único elemento está ordenado.
- **Passo da Indução (Segunda Alternativa):** Seja S um conjunto de $n \geq 2$ inteiros e x um elemento qualquer de S. Sejam S_1 e S_2 os subconjuntos de $S - x$ dos elementos menores ou iguais a x e maiores que x, respectivamente. Ambos S_1 e S_2 possuem menos de n elementos. Por hipótese de indução, sabemos ordenar os conjuntos S_1 e S_2. Podemos obter S ordenado concatenando S_1 ordenado, x e S_2 ordenado.

Projeto por Indução Forte

- Esta indução dá origem ao algoritmo de divisão e conquista *Quicksort*.
- Na implementação do algoritmo, o conjunto S é um vetor de tamanho n novamente.
- Em contraste ao *Mergesort*, no *Quicksort* é a operação de divisão que é a operação mais custosa: depois de escolhemos o *pivot*, temos que separar os elementos do vetor maiores que o *pivot* dos menores que o *pivot*.
- Conseguimos fazer essa divisão com $\Theta(n)$ operações: basta varrer o vetor com dois apontadores, um varrendo da direita para a esquerda e outro da esquerda para a direita, em busca de elementos situados na parte errada do vetor, e trocar um par de elementos de lugar quando encontrado.
- Após essa etapa basta ordenarmos os dois trechos do vetor recursivamente para obtermos o vetor ordenado, ou seja, a conquista é imediata.

Quicksort - Pseudo-código

Quicksort(A, e, d)

▷ **Entrada:** Vetor A de números inteiros e os índices e e d que delimitam início e fim do subvetor a ser ordenado.
▷ **Saída:** Subvetor de A de e a d ordenado.

01. **se** $d > e$ **então**
    ▷ escolhe o pivot
02. $v := A[d]$
03. $i := e - 1$
04. $j := d$

Quicksort - Pseudo-código (cont.)

Quicksort(A, e, d): cont.

05. **repita**
06. **repita** $i := i + 1$ **até que** $A[i] \geq v$
07. **repita** $j := j - 1$ **até que** $A[j] \leq v$ **ou** $j = e$
08. $t := A[i]$
09. $A[i] := A[j]$
10. $A[j] := t$
11. **até que** $j \leq i$
12. $A[j] := A[i]$
13. $A[i] := A[d]$
14. $A[d] := t$
15. Quicksort($A, e, i - 1$)
16. Quicksort($A, i + 1, d$)

Quicksort - Análise de Complexidade

- Quantas comparações e quantas trocas o algoritmo *Quicksort* executa no pior caso ?
- Certamente a operação de divisão tem complexidade $\Theta(n)$, mas o tamanho dos dois subproblemas depende do *pivot* escolhido.
- No pior caso, cada divisão sucessiva do *Quicksort* separa um único elemento dos demais, recaindo na recorrência:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n, & n > 1, \end{cases}$$

- Portanto, $\Theta(n^2)$ comparações e trocas são executadas no pior caso.

Quicksort - Análise de Complexidade

- Então, o algoritmo *Quicksort* é assintoticamente menos eficiente que o *Mergesort* no pior caso.
- Veremos a seguir que, no caso médio, o *Quicksort* efetua $\Theta(n \log n)$ comparações e trocas.
- Assim, na prática, o *Quicksort* é bastante eficiente, com uma vantagem adicional em relação ao *Mergesort*: é *in place*, isto é, não utiliza um vetor auxiliar.

Quicksort - Análise de Caso Médio

- Considere que i é o índice da posição do *pivot* escolhido no vetor ordenado.
- Supondo que qualquer elemento do vetor tem igual probabilidade de ser escolhido como o *pivot*, então, na média, o tamanho dos subproblemas resolvidos em cada divisão sucessiva será:

$$\frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i)), \text{ para } n \geq 2$$

- Não é difícil ver que:

$$\sum_{i=1}^{n} T(i-1) = \sum_{i=1}^{n} T(n-i) = \sum_{i=1}^{n-1} T(i), \text{ supondo } T(0) = 0$$

Quicksort - Análise de Caso Médio

- Assim, no caso médio, o número de operações efetuadas pelo *Quicksort* é dado pela recorrência:

$$T(n) = \begin{cases} 0, & n = 0 \text{ ou } n = 1 \\ \frac{2}{n}\sum_{i=1}^{n-1} T(i) + n - 1, & n \geq 2, \end{cases}$$

- Esta é uma recorrência de história completa conhecida ! Sabemos que $T(n) \in \Theta(n \log n)$.
- Portanto, na média, o algoritmo *Quicksort* executa $\Theta(n \log n)$ trocas e comparações.

Heapsort

- O projeto por indução que leva ao *Heapsort* é essencialmente o mesmo do *Selection Sort*: selecionamos e posicionamos o maior (ou menor) elemento do conjunto e então aplicamos a hipótese de indução para ordenar os elementos restantes.
- A diferença importante é que no *Heapsort* utilizamos a estrutura de dados *heap* para selecionar o maior (ou menor) elemento eficientemente.
- Um *heap* é um vetor que simula uma árvore binária completa, a menos, talvez, do último nível, com estrutura de *heap*.

Heapsort - Estrutura do Heap

- Na simulação da árvore binária completa com um vetor, definimos que o nó i tem como filhos esquerdo e direito os nós $2i$ e $2i+1$ e como pai o nó $\lfloor i/2 \rfloor$.
- Uma árvore com estrutura de *heap* é aquela em que, para toda subárvore, o nó raiz é maior ou igual (ou menor ou igual) às raízes das subárvores direita e esquerda.
- Assim, o maior (ou menor) elemento do *heap* está sempre localizado no topo, na primeira posição do vetor.

Heapsort - O Algoritmo

- Então, o uso da estrutura *heap* permite que:
  - O elemento máximo do conjunto seja determinado e corretamente posicionado no vetor em tempo constante, trocando o primeiro elemento do *heap* com o último.
  - O trecho restante do vetor (do índice 1 ao $n-1$), que pode ter deixado de ter a estrutura de heap, volte a tê-la com número de trocas de elementos proporcional à altura da árvore.
- O algoritmo *Heapsort* consiste então da construção de um *heap* com os elementos a serem ordenados, seguida de sucessivas trocas do primeiro com o último elemento e rearranjos do *heap*.

Heapsort - Pseudo-código

Heapsort(A)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A ordenado.
1. *ConstroiHeap*(A, n)
2. **para** *tamanho* **de** n **decrescendo até** 2 **faça**
   ▷ troca elemento do topo do heap com o último
3. $t := A[tamanho]$; $A[tamanho] := A[1]$; $A[1] := t$
   ▷ rearranja A para ter estrutura de heap
4. *AjustaHeap*$(A, 1, tamanho)$

Heapsort - Rearranjo - Pseudo-código

AjustaHeap(A, i, n)

▷ **Entrada:** Vetor A de n números inteiros com estrutura de heap, exceto, talvez, pela subárvore de raiz i.
▷ **Saída:** Vetor A com estrutura de heap.
1. **se** $2i \leq n$ e $A[2i] \geq A[i]$
2. **então** $maximo := 2i$ **senão** $maximo := i$
3. **se** $2i + 1 \leq n$ e $A[2i + 1] \geq A[maximo]$
4. **então** $maximo := 2i + 1$
5. **se** $maximo \neq i$ **então**
6. $t := A[maximo]$; $A[maximo] := A[i]$; $A[i] := t$
7. $AjustaHeap(A, maximo, n)$

Heapsort - Análise de Complexidade

- Quantas comparações e quantas trocas são executadas no pior caso na etapa de ordenação do algoritmo *Heapsort* ?
- A seleção e posicionamento do elemento máximo é feita em tempo constante.
- No pior caso, a função **AjustaHeap** efetua $\Theta(h)$ comparações e trocas, onde h é a altura do *heap* que contém os elementos que resta ordenar.
- Como o *heap* representa uma árvore binária completa, então $h \in \Theta(\log i)$, onde i é o número de elementos do *heap*.

Heapsort - Análise de Complexidade

- Logo, a complexidade da etapa de ordenação do *Heapsort* é:

$$\sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = n \log n.$$

- Portanto, na etapa de ordenação do *Heapsort* são efetuadas $O(n \log n)$ comparações e trocas no pior caso.
- Na verdade $\sum_{i=1}^{n} \log i \in \Theta(n \log n)$, conforme visto em exercício de lista !
- No entanto, também temos que computar a complexidade de construção do *heap*.

Heapsort - Construção do *Heap* - *Top-down*

- Mas, como construímos o *heap* ?
- Se o trecho de 1 a i do vetor tem estrutura de *heap*, é fácil adicionar a folha $i + 1$ ao *heap* e em seguida rearranjá-lo, garantindo que o trecho de 1 a $i + 1$ tem estrutura de *heap*.
- Esta é a abordagem *top-down* para construção do *heap*.

Construção do *Heap* - *Top-down* - Pseudo-código

ConstroiHeap(A, n)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A com estrutura de *heap*.
1. **para** i **de** 2 **até** n **faça**
2. $v := A[i]$
3. $j := i$
4. **enquanto** $j > 1$ **e** $A[j \ div \ 2] < v$ **faça**
5. $A[j] := A[j \ div \ 2]$
6. $j := j \ div \ 2$
7. $A[j] := v$

Construção do *Heap* - *Top-down* - Complexidade

- Quantas comparações e quantas trocas são executadas no pior caso na construção do *heap* pela abordagem *top-down* ?
- O rearranjo do *heap* na iteração i efetua $\Theta(h)$ comparações e trocas no pior caso, onde h é a altura da árvore representada pelo trecho do *heap* de 1 a i. Logo, $h \in \Theta(\log i)$.
- Portanto, o número de comparações e trocas efetuadas construção do *heap* por esta abordagem é:

$$\sum_{i=1}^{n} \log i \in \Theta(n \log n).$$

- Então, o algoritmo *Heapsort* efetua ao todo $\Theta(n \log n)$ comparações e trocas no pior caso.

Heapsort - Construção do *Heap* - *Bottom-up*

- É possível construir o *heap* de forma mais eficiente.
- Suponha que o trecho de i a n do vetor é tal que, para todo j, $i \leq j \leq n$, a subárvore de raiz j representada por esse trecho do vetor tem estrutura de *heap*.
- Note que, em particular, o trecho de $\lfloor n/2 \rfloor + 1$ a n do vetor satisfaz a propriedade, pois inclui apenas folhas da árvore binária de n elementos.
- Podemos então executar **AjustaHeap($A, i-1, n$)**, garantindo assim que o trecho de $i-1$ a n satisfaz a propriedade.
- Esta é a abordagem *bottom-up* para construção do *heap*.

Construção do *Heap* - *Bottom-up* - Pseudo-código

ConstroiHeap(A, n)

▷ **Entrada:** Vetor A de n números inteiros.
▷ **Saída:** Vetor A com estrutura de *heap*.
1. **para** i **de** n div 2 **decrescendo até** 1 **faça**
2. *AjustaHeap*(A, i, n)

Construção do *Heap* - *Bottom-up* - Complexidade

- Quantas comparações e quantas trocas são executadas no pior caso na construção do *heap* pela abordagem *bottom-up* ?
- O rearranjo do *heap* na iteração i efetua $\Theta(h)$ comparações e trocas no pior caso, onde h é a altura da subárvore de raiz i.
- Seja $T(h)$ a soma das alturas de todos os nós de uma árvore binária completa de altura h.
- O número de operações efetuadas na construção do *heap* pela abordagem *bottom-up* é $T(\log n)$.

Construção do *Heap* - *Bottom-up* - Complexidade

- A expressão de $T(h)$ é dada pela recorrência:

$$T(h) = \begin{cases} 0, & h = 0 \\ 2T(h-1) + h, & h > 1, \end{cases}$$

- É possível provar (por indução) que $T(h) = 2^{h+1} - (h+2)$.
- Então, $T(\log n) \in \Theta(n)$ e a abordagem *bottom-up* para construção do *heap* apenas efetua $\Theta(n)$ comparações e trocas no pior caso.
- Ainda assim, a complexidade do *Heapsort* no pior caso é $\Theta(n \log n)$.

O problema da Ordenação - Cota Inferior

- Estudamos diversos algoritmos para o problema da ordenação.
- Todos eles têm algo em comum: usam somente comparações entre dois elementos do conjunto a ser ordenado para definir a posição relativa desses elementos.
- Isto é, o resultado da comparação de x_i com x_j, $i \neq j$, define se x_i será posicionado antes ou depois de x_j no conjunto ordenado.
- Todos os algoritmos dão uma cota superior para o número de comparações efetuadas por um algoritmo que resolva o problema da ordenação.
- A menor cota superior é dada pelos algoritmos *Mergesort* e o *Heapsort*, que efetuam $\Theta(n \log n)$ comparações no pior caso.

O problema da Ordenação - Cota Inferior

- Será que é possível projetar um algoritmo de ordenação baseado em comparações ainda mais eficiente ?
- Veremos a seguir que não...
- É possível provar que qualquer algoritmo que ordena n elementos baseado apenas em comparações de elementos efetua no mínimo $\Omega(n \log n)$ comparações no pior caso.
- Para demonstrar esse fato, vamos representar os algoritmos de ordenação em um modelo computacional abstrato, denominado árvore (binária) de decisão.

Árvores de Decisão - Modelo Abstrato

- Os nós internos de uma árvore de decisão representam comparações feitas pelo algoritmo.
- As subárvores de cada nó interno representam possibilidades de continuidade das ações do algoritmo após a comparação.
- No caso das árvores binárias de decisão, cada nó possui apenas duas subárvores. Tipicamente, as duas subárvores representam os caminhos a serem seguidos conforme o resultado (verdadeiro ou falso) da comparação efetuada.
- As folhas são as respostas possíveis do algoritmo após as decisões tomadas ao longo dos caminhos da raiz até as folhas.

Árvores de Decisão para o Problema da Ordenação

- Considere a seguinte definição alternativa do problema da ordenação:

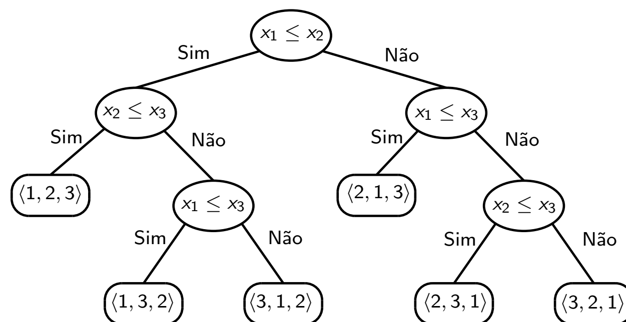
Problema da Ordenação:

Dado um conjunto de n inteiros $x_1, x_2, \ldots, x_n$, encontre uma permutação p dos índices $1 \leq i \leq n$ tal que $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.

- É possível representar um algoritmo para o problema da ordenação através de uma árvore de decisão da seguinte forma:
  - Os nós internos representam comparações entre dois elementos do conjunto, digamos $x_i \leq x_j$.
  - As ramificações representam os possíveis resultados da comparação: verdadeiro se $x_i \leq x_j$, ou falso se $x_i > x_j$.
  - As folhas representam possíveis soluções: as diferentes permutações dos n índices.

Árvores de Decisão para o Problema da Ordenação

Veja a árvore de decisão que representa o comportamento do *Insertion Sort* para um conjunto de 3 elementos:



Árvores de Decisão para o Problema da Ordenação

- Ao representarmos um algoritmo de ordenação qualquer baseado em comparações por uma árvore binária de decisão, todas as permutações de n elementos devem ser possíveis soluções.
- Assim, a árvore binária de decisão deve ter pelo menos $n!$ folhas, podendo ter mais (nada impede que duas seqüências distintas de decisões terminem no mesmo resultado).
- O caminho mais longo da raiz a uma folha representa o pior caso de execução do algoritmo.
- A altura mínima de uma árvore binária de decisão com pelo menos $n!$ folhas dá o número mínimo de comparações que o melhor algoritmo de ordenação baseado em comparações deve efetuar.

A Cota Inferior

- Qual a altura mínima, em função de n, de uma árvore binária de decisão com pelo menos $n!$ folhas ?
- Uma árvore binária de decisão T com altura h tem, no máximo, 2^h folhas.
- Portanto, se T tem pelo menos $n!$ folhas, $n! \leq 2^h$, ou seja, $h \geq \log_2 n!$.
- Mas,

$$\begin{aligned} \log_2 n! &= \sum_{i=1}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log i \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \log n/2 \\ &\geq (n/2 - 1) \log n/2 \\ &= n/2 \log n - n/2 - \log n + 1 \\ &\geq n/4 \log n, \text{ para } n \geq 16. \end{aligned}$$

- Então, $h \in \Omega(n \log n)$.

Observações finais

- Provamos então que $\Omega(n \log n)$ é uma cota inferior para o problema da ordenação.
- Portanto, os algoritmos *Mergesort* e *Heapsort* são algoritmos ótimos.

Ordenação em Tempo Linear

Algoritmos lineares para ordenação

Os seguintes algoritmos de ordenação têm complexidade $O(n)$, onde n é o tamanho do vetor A a ser ordenado:

- Counting Sort: Elementos são números inteiros "pequenos"; mais precisamente, inteiros x onde $x \in O(n)$.
- Radix Sort: Elementos são números inteiros de comprimento máximo constante, isto é, independente de n.
- Bucket Sort: Elementos são números reais uniformemente distribuídos no intervalo $[0..1)$.

Counting Sort

- Considere o problema de ordenar um vetor A de n inteiros quando é sabido que todos os inteiros estão no intervalo entre 1 e k, ou seja, $k \in O(n)$.
- Podemos ordenar o vetor simplesmente contando, para cada inteiro i no vetor, quantos elementos do vetor são menores que i.
- É exatamente o que faz o algoritmo *Counting Sort*, em tempo $O(n + k)$, isto é $O(n)$.
- O algoritmo usa dois vetores auxiliares:
  1. C de tamanho k que guarda em $C[i]$ o número de ocorrências de elementos $\leq i$ em A.
  2. B de tamanho n onde se constrói o vetor ordenado.

Counting Sort - Algoritmo

CountingSort(A, k)

$\triangleright$ **Entrada:** - Vetor A de tamanho ne um inteiro k, o valor do maior inteiro em A.
$\triangleright$ **Saída:** Os elementos do vetor A em ordem crescente.

1. **para** $i := 1$ **até** k **faça** $C[i] := 0$
2. **para** $j := 1$ **até** n **faça** $C[A[j]] := C[A[j]] + 1$
3. **para** $i := 2$ **até** k **faça** $C[i] := C[i] + C[i-1]$
4. **para** $j := n$ **até** 1 **faça**
5. $\quad B[C[A[j]]] := A[j]$
6. $\quad C[A[j]] := C[A[j]] - 1$
7. **retorne**(B)

Counting Sort - Complexidade

- Qual a complexidade do algoritmo *Counting Sort* ?
- O algoritmo não faz comparações entre elementos de A.
- Sua complexidade deve ser medida em função do número das outras operações, aritméticas, atribuições, etc.
- Claramente, o número de tais operações é uma função em $O(n + k)$, já que temos dois laços simples com n iterações e dois com k iterações.
- Assim, quando $k \in O(n)$, este algoritmo tem complexidade $O(n)$.

Algo de errado com o limite inferior de $\Omega(n \log n)$ para ordenação ?

Algoritmos *in-place* e *estáveis*

- Algoritmos de ordenação podem ser ou não *in-place* ou *estáveis*.
- Um algoritmo de ordenação é in-place se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.
- Exemplos: o *Quicksort* e o *Heapsort* são métodos de ordenação *in-place*, já o *Mergesort* e o *Counting Sort* não.
- Um método de ordenação é estável se elementos iguais ocorrem no vetor ordenado na mesma ordem em que são passados na entrada.
- Exemplos: o *Counting Sort* e o *Quicksort* são exemplos de métodos estáveis (desde que certos cuidados sejam tomados na implementação). O *Heapsort* não é.

O *Radix Sort*

- Considere agora o problema de ordenar um vetor A de n inteiros quando é sabido que todos os inteiros podem ser representados com apenas d dígitos, onde d é uma constante.
- Por exemplo, os elementos de A podem ser CEPs, ou seja, inteiros de 8 dígitos.
- Poderíamos ordenar os elementos do vetor dígito a dígito da seguinte forma:
    - Separamos os elementos do vetor em grupos que compartilham o mesmo dígito mais significativo.
    - Em seguida, ordenamos os elementos em cada grupo pelo mesmo método, levando em consideração apenas os $d-1$ dígitos menos significativos.
- Esse método funciona, mas requer o uso de bastante memória adicional para a organização dos grupos e subgrupos.

O *Radix Sort*

- Podemos evitar o uso excessivo de memória adicional ainda utilizando a idéia de ordenar os números dígito a dígito: basta começar pelo dígito menos significativo.
- É isso que faz o algoritmo *Radix Sort*.
- Para que o algoritmo *Radix Sort* funcione corretamente, é necessário utilizar um método de ordenação *estável* para a ordenação segundo cada dígito individualmente.
- Para isso podemos usar, por exemplo, o *Counting Sort*.

O *Radix Sort*

Suponha que os elementos do vetor A a ser ordenado sejam números inteiros de até d dígitos. O *Radix Sort* é simplesmente:

RadixSort(A, d)

1. **para** $i := 1$ **até** d **faça**
2. ordene os elementos de A pelo i-ésimo dígito usando um método **estável**

O *Radix Sort* - Exemplo

329		720		720		329
457		355		329		355
657		436		436		436
839		457		839		457
436	→	657	→	355	→	657
720		329		457		720
355		839		657		839
		↑ (último dígito)		↑ (dígito do meio)		↑ (primeiro dígito)

O *Radix Sort* - Corretude

O seguinte argumento indutivo garante a corretude do algoritmo:

- Sabemos, por hipótese de indução, que os números estão ordenados com relação aos $i-1$ dígitos menos significativos.
- Agora vamos argumentar que ao ordenarmos os números com relação ao i-ésimo dígito menos significativo via um algoritmo estável, obtemos os números ordenados com relação aos i dígitos menos significativos.
- Para dois números com o i-ésimo dígito distintos, o de menor valor no dígito i certamente estará antes do de maior valor no dígito i.
- Se ambos possuem o i-ésimo dígito igual, então a ordem dos dois também estará correta pois utilizamos um método de ordenação **estável** e, por hipótese de indução, os dois elementos já estavam ordenados segundo os $i-1$ dígitos menos significativos.

O *Radix Sort* - Complexidade

- Qual é a complexidade do *Radix Sort* ?
- Depende da complexidade do algoritmo estável usado para ordenar cada dígito dos elementos.
- Se essa complexidade estiver em $\Theta(f(n))$, obtemos uma complexidade total de $\Theta(d\ f(n))$ para o *Radix Sort*.
- Como supomos d constante, a complexidade reduz-se para $\Theta(f(n))$.
- Se o algoritmo estável for, por exemplo, o *Counting Sort*, obtemos a complexidade $\Theta(n+k)$.
- Supondo $k \in O(n)$, resulta numa complexidade linear em n.

E o limite inferior de $\Omega(n \log n)$ para ordenação ?

O *Radix Sort* - Complexidade

- Em contraste, um algoritmo por comparação como o *MergeSort* teria complexidade $n \log n$.
- Assim, o *Radix Sort* é mais vantajoso que o *MergeSort* quando $d < \log n$, ou seja, o número de dígitos for menor que $\log n$.
- Se considerarmos que n também é um limite superior para o maior valor a ser ordenado, então $O(\log n)$ é uma estimativa para o tamanho, em dígitos, desse número.
- Isso significa que não há diferença significativa entre o desempenho do *MergeSort* e do *Radix Sort*?

O *Radix Sort* - Complexidade

- A vantagem de se usar o *Radix Sort* fica evidente quando interpretamos os dígitos de forma mais geral que simplesmente 0..9, embora seja esse o significado literal de "dígitos".
- Tomemos o seguinte exemplo: suponha que desejemos ordenar um conjunto de 2^{20} números de 64 bits. Então, o *MergeSort* faria cerca de 20×2^{20} comparações e usaria um vetor auxiliar de tamanho 2^{20}.
- Se interpretarmos cada número do conjunto como tendo 4 dígitos em base 2^{16}, e usarmos o *Radix Sort* com o *Counting Sort* como método estável, a complexidade de tempo seria da ordem de $4(2^{20}+2^{16})$ operações, uma redução substancial. Mas, note que utilizamos dois vetores auxiliares, um de tamanho 2^{16} e outro de tamanho 2^{20}.

O *Radix Sort* - Complexidade

- O nome *Radix Sort* vem da **base** (em inglês *radix*) em que intepretamos os dígitos.
- Veja que se o uso de memória auxiliar for muito limitado, então o melhor mesmo é usar um algoritmo de ordenação por comparação *in-place*.
- Note que é possível usar o *Radix Sort* para ordenar outros tipos de elementos, como datas, palavras em ordem lexicográfica e qualquer outro tipo que possa ser visto como uma d-upla ordenada de itens comparáveis.

O *Bucket Sort*

- Assim como o *Counting Sort*, supõe que os n elementos a serem ordenados têm uma natureza peculiar. Neste caso, os elementos estão distribuídos uniformemente no intervalo semi-aberto $[0, 1)$.
- A idéia é dividir o intervalo $[0, 1)$ em n segmentos de mesmo tamanho (*buckets*) e distribuir os n elementos nos seus respectivos segmentos. Como os elementos estão distribuídos uniformemente, espera-se que o número de elementos seja aproximadamente o mesmo em todos os segmentos.
- Em seguida, os elementos de cada segmento são ordenados por um método qualquer. Finalmente, os segmentos ordenados são concatenados em ordem crescente.

O *Bucket Sort* - Código

BucketSort(A)

1. $n :=$ comprimento de A
2. **para** $i := 1$ **até** n **faça**
3. insira $A[i]$ na lista ligada $B[\lfloor nA[i] \rfloor]$
4. **para** $i := 0$ **até** $n - 1$ **faça**
5. ordene a lista $B[i]$ com *Insertion Sort*
6. Concatene as listas $B[0], B[1], \ldots, B[n-1]$ nessa ordem.

Bucket Sort - Exemplo

$A =$

1	.78
2	.17
3	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68

$B =$

0	
1	.12, .17
2	.21, .23, .26
3	.39
4	
5	
6	.68
7	.72, .78
8	
9	.94

O *Bucket Sort* - Corretude

- A corretude do algoritmo depende do fato de que elementos x e y de A, $x < y$, ou terminem no mesmo segmento ou sejam destinados a segmentos diferentes $B[i]$ e $B[j]$, respectivamente, onde $i < j$.
- A primeira possibilidade implica que x aparecerá antes de y na concatenação final, já que cada segmento é ordenado.
- Na segunda hipótese, temos que, se $x < y$, então $i = \lfloor nx \rfloor \leq \lfloor ny \rfloor = j$. Como $i \neq j$, temos $i < j$.

O *Bucket Sort* - Complexidade

- É claro que o pior caso do *Bucket Sort* é quadrático, supondo-se que as ordenações dos segmentos seja feita com ordenação por inserção.
- Entretanto, o tempo esperado é linear.
  Intuitivamente, a idéia da demonstração é que, como os n elementos estão distribuídos uniformemente no intervalo $[0, 1)$, então o tamanho esperado das listas também é uniforme, em $\Theta(1)$.
- Portanto as ordenações das n listas $B[i]$ leva tempo total esperado em $\Theta(n)$.
- Os detalhes podem ser vistos no livro de Cormen, Leiserson e Rivest.

Estatísticas de Ordem

Estatísticas de Ordem (Problema da Seleção)

- Estamos interessados em resolver o **Problema da Seleção**:
  Dado um conjunto A de n números reais e um inteiro k, determinar qual é o k-ésimo menor elemento de A.
- A determinação do elemento mínimo, elemento máximo e mediana de um conjunto são casos particulares do problema da seleção, onde $k = 1$, $k = n$ e $k = \left\lceil \frac{n}{2} \right\rceil$, respectivamente.
- Um algoritmo para o problema da seleção deve de alguma forma obter informação, ainda que parcial, sobre a ordem dos elementos do conjunto.

Problema da Seleção - Primeira Solução

- Um primeiro algoritmo para o problema da seleção pode ser ordenar os n elementos do conjunto A e retornar o elemento situado na posição k após a ordenação.
- A complexidade de pior caso desse algoritmo é dada pela complexidade de pior caso do algoritmo de ordenação utilizado, portanto $\Omega(n \log n)$.
- Se utilizarmos o *Mergesort* ou *Heapsort* para a ordenação, teremos um algoritmo $O(n \log n)$ para o problema.
- Será que é necessário ordenar todos os elementos ?
- Para determinarmos o máximo e o mínimo não é necessário ordenar os elementos, pois existe um algoritmo $O(n)$ no pior caso.

Será que existe algoritmo $O(n)$ para k qualquer ?

Problema da Seleção - Segunda Solução

- Podemos utilizar a idéia do *Quicksort* de particionar os elementos em dois conjuntos com relação a um elemento *pivot*.
- Assim, determinamos a posição i do *pivot* no conjunto ordenado e o elemento procurado (k-ésimo menor) poderá estar em apenas um dos dois subconjuntos.
- Seguindo essa idéia podemos projetar um algoritmo de divisão e conquista para o problema da seleção:
  - Se $i = k$, o *pivot* é o elemento procurado.
  - Se $i > k$, então o elemento procurado está no subconjunto dos elementos menores ou iguais ao *pivot*;
  - Se $i < k$, então o elemento procurado está no subconjunto dos elementos maiores ou iguais ao *pivot*;

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Segunda Solução - Pseudo-código

Seleção(A, e, d, k)

▷ **Entrada:** vetor A de números reais, os índices e e d que delimitam início e fim do subvetor onde será feita a seleção e k, o índice do elemento procurado no vetor ordenado.
▷ **Saída:** o k-ésimo menor elemento do vetor A.

1. **se** $d > e$ **então**
   ▷ O índice i dá a posição do pivot no vetor ordenado
2. $i \leftarrow$ PARTIÇÃO(A, e, d);
3. **se** $i = k$ **então retorne**($A[i]$)
4. **senão se** $i > k$
5. **então faça** Seleção($A, e, i-1, k$)
6. **senão faça** Seleção($A, i+1, d, k-i$)

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Partição(A, e, d)

▷ **Entrada:** vetor A de números reais, os índices e e d que delimitam início e fim do subvetor onde será feita a partição.
▷ **Saída:** índice i da posição do elemento $v := A[d]$ do vetor de entrada quando o subvetor $A[e, d]$ estiver ordenado. Elementos em $A[e, i-1]$ serão menores que v e elementos em $A[i+1, d]$ serão maiores que v.

1. $v := A[d]$ ▷ escolhe o pivot
2. $i := e - 1$; $j := d$
3. **repita**
4. **repita** $i := i + 1$ **até que** $A[i] \geq v$
5. **repita** $j := j - 1$ **até que** $A[j] \leq v$ ou $(j = 1)$
6. $t := A[i]$; $A[i] := A[j]$; $A[j] := t$ ▷ Troca $A[i]$ com $A[j]$
7. **até que** $j \leq i$
   ▷ Destroca $A[i]$ com $A[j]$ e troca $A[i]$ com o pivot
8. $A[j] := A[i]$; $A[i] := A[d]$; $A[d] := t$
9. **retorne**(i).

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Segunda Solução – Complexidade

- Assim como no *Quicksort*, no pior caso, cada partição separa apenas um elemento dos demais nas sucessivas chamadas recursivas; ou seja:

$$T(n) = T(n-1) + n;$$

- Portanto, este algoritmo tem complexidade $O(n^2)$ no pior caso, que é assintoticamente pior que o primeiro algoritmo para este problema, que usa ordenação.
- No entanto, este algoritmo tem complexidade $O(n)$ no caso médio, portanto, mais eficiente na prática.

Será que existe um algoritmo $O(n)$ no pior caso para o problema da seleção ?

Problema da Seleção - Terceira Solução

1. Divida os n elementos em $\lfloor \frac{n}{5} \rfloor$ subconjuntos de 5 elementos e um subconjunto de $n \bmod 5$ elementos.

1	•	•	...	•	•	•	...	•	•	
2	•	•	...	•	•	•	...	•	•	
	•	•	...	•	•	•	...	•	•	n
	•	•	...	•	•	•	...	•		
	•	•	...	•	•	•	...	•		

2. Encontre a mediana de cada um dos $\lceil \frac{n}{5} \rceil$ subconjuntos usando um método simples de ordenação como o *Insertion Sort*.

1	•	•	...	•	•	•	...	•		
2	•	•	...	•	•	•	...	•	•	
	∘	∘	...	∘	∘	∘	...	∘	∘	
	•	•	...	•	•	•	...	•	•	n
	•	•	...	•	•	•	...	•		

Na figura acima, cada subconjunto está ordenado crescentemente, de cima para baixo.

Problema da Seleção - Terceira Solução (cont.)

3. Determine, recursivamente, a mediana x das medianas dos subconjuntos de 5 elementos.

1	•	•	...	•	•	•	...	•		
2	•	•	...	•	•	•	...	•	•	
	∘	∘	...	∘	∘	∘	...	∘	∘	
	•	•	...	•	•	•	...	•	•	n
	•	•	...	•	•	•	...	•		

A figura acima é a mesma que a anterior, com algumas colunas trocadas de lugar. A ordem dos elementos em cada coluna permanece inalterada. Por simplicidade de exposição, supomos que a última coluna permanece no mesmo lugar.

Problema da Seleção - Terceira Solução (cont.)

4. Como na solução anterior, usando x como *pivot*, particione o conjunto original criando dois subconjuntos $A_{\leq}$ e $A_{>}$, onde
   - $A_{\leq}$ contém os elementos $\leq x$; e
   - $A_{>}$ contém os elementos $> x$.

   Se a posição final de x após o particionamento é i, então $|A_{\leq}| = i - 1$ e $|A_{>}| = n - i$.
5. Finalmente, para encontrar o k-ésimo menor elemento do conjunto, compare k com a posição i de x após o particionamento:
   - Se $i = k$, x é o elemento procurado;
   - Se $i < k$, então determine, recursivamente, o k-ésimo menor elemento do subconjunto $A_{\leq}$;
   - caso contrário, determine, recursivamente, o $(k - i)$-ésimo menor elemento do subconjunto $A_{>}$.

Terceira Solução - Complexidade

Para calcular a complexidade $T(n)$, eis um resumo dos passos do algoritmo e suas complexidades:

1. Divisão em subconjuntos de 5 elementos. $O(n)$
2. Encontrar as medianas dos subconjuntos. $O(n)$
3. Recursão para encontrar x, a mediana das medianas. $T(\lceil n/5 \rceil)$
4. Particionamento com pivot x. $O(n)$
5. Recursão para encontrar o k-ésimo menor elemento no conjunto $A_{\leq}$ ou $A_{>}$. $T(i-1)$ ou $T(n-i)$.

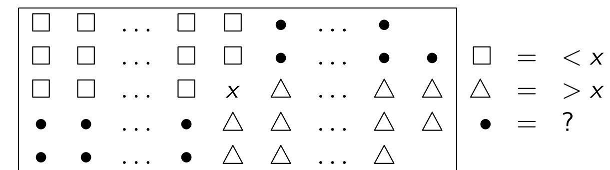
Portanto, precisamos estimar o valor máximo de i para determinar a complexidade $T(n)$.

Terceira Solução - Complexidade

O diagrama abaixo classifica os elementos da última figura.

Veja que o número de elementos comprovadamente $> x$, isto é $\triangle$s, é, no mínimo, $\frac{3n}{10} - 6$.

Isto porque existem, no mínimo, $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$ grupos que contribuem com 3 elementos maiores que x, excetuando-se o último e aquele que contém x. Portanto, $3\left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2\right) \geq \frac{3n}{10} - 6$.



Terceira Solução - Complexidade

Da mesma forma, o número de elementos comprovadamente $< x$, isto é $\square$s, é, no mínimo, $\frac{3n}{10} - 6$. Assim, no passo 5 do algoritmo, o valor de i ou $n - i$ não é maior que

$$n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6.$$

A recorrência $T(n)$ está agora completa:

$$T(n) \leq \begin{cases} \Theta(1), & n \leq 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n), & n > 140, \end{cases}$$

Uma demonstração por indução mostra que existe uma constante c positiva para a qual $T(n) \leq cn$, para todo $n > 140$. Portanto, o algoritmo tem comportamento linear no pior caso.

Programação Dinâmica

Programação Dinâmica: Conceitos Básicos

- Tipicamente o paradigma de programação dinâmica aplica-se a problemas de **otimização**.
- Podemos utilizar programação dinâmica em problemas onde há:
  - **Subestrutura Ótima:** As soluções ótimas do problema incluem soluções ótimas de subproblemas.
  - **Sobreposição de Subproblemas:** O cálculo da solução através de recursão implica no recálculo de subproblemas.

Programação Dinâmica: Conceitos Básicos (Cont.)

- A técnica de programação dinâmica visa evitar o recálculo desnecessário das soluções dos subproblemas.
- Para isso, soluções de subproblemas são armazenadas em tabelas.
- Logo, para que o algoritmo de programação dinâmica seja eficiente, é preciso que o número total de subproblemas que devem ser resolvidos seja pequeno (polinomial no tamanho da entrada).

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Multiplicação de Cadeias de Matrizes

Problema: Multiplicação de Matrizes

Calcular o número mínimo de operações de multiplicação (escalar) necessários para computar a matriz M dada por:

$$M = M_1 \times M_2 \times \dots M_i \dots \times M_n$$

onde M_i é uma matriz de b_{i-1} linhas e b_i colunas, para todo $i \in \{1, \dots, n\}$.

- Matrizes são multiplicadas aos pares sempre. Então, é preciso encontrar uma parentização (agrupamento) ótimo para a cadeia de matrizes.
- Para calcular a matriz M' dada por $M_i \times M_{i+1}$ são necessárias $b_{i-1} * b_i * b_{i+1}$ multiplicações entre os elementos de M_i e M_{i+1}.

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Multiplicação de Cadeias de Matrizes (Cont.)

- **Exemplo:** Qual é o mínimo de multiplicações escalares necessárias para computar $M = M_1 \times M_2 \times M_3 \times M_4$ com $b = \{200, 2, 30, 20, 5\}$?
- As possibilidades de parentização são:

$$
\begin{array}{lcl}
M = (M_1 \times (M_2 \times (M_3 \times M_4))) & \rightarrow & 5.300 \text{ multiplicações} \\
M = (M_1 \times ((M_2 \times M_3) \times M_4)) & \rightarrow & 3.400 \text{ multiplicações} \\
M = ((M_1 \times M_2) \times (M_3 \times M_4)) & \rightarrow & 4.500 \text{ multiplicações} \\
M = ((M_1 \times (M_2 \times M_3)) \times M_4)) & \rightarrow & 29.200 \text{ multiplicações} \\
M = (((M_1 \times M_2) \times M_3) \times M_4) & \rightarrow & 152.000 \text{ multiplicações}
\end{array}
$$

- A ordem das multiplicações faz **muita** diferença !

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Multiplicação de Cadeias de Matrizes (Cont.)

- Poderíamos calcular o número de multiplicações para todas as possíveis parentizações.
- O número de possíveis parentizações é dado pela recorrência:

$$P(n) = \begin{cases} 1, & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1, \end{cases}$$

- Mas $P(n) \in \Omega(4^n/n^{\frac{3}{2}})$, a estratégia de força bruta é impraticável !

Multiplicação de Cadeias de Matrizes (Cont.)

- Dada uma parentização ótima, existem dois pares de parênteses que identificam o último par de matrizes que serão multiplicadas. Ou seja, existe k tal que $M = A \times B$ onde $A = M_1 \times \ldots M_k$ e $B = M_{k+1} \times \ldots M_n$.
- Como a parentização de M é ótima, as parentizações no cálculo de A e B devem ser ótimas também, caso contrário, seria possível obter uma parentização de M ainda melhor !
- Eis a subestrututra ótima do problema: a parentização ótima de M inclui a parentização ótima de A e B.

Multiplicação de Cadeias de Matrizes (Cont.)

- De forma geral, se $m[i,j]$ é número mínimo de multiplicações que deve ser efetuado para computar $M_i \times M_{i+1} \times \ldots \times M_j$, então $m[i,j]$ é dado por:

$$m[i,j] := \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + b_{i-1} * b_k * b_j\}.$$

- Podemos então projetar um algoritmo recursivo (indutivo) para resolver o problema.

Multiplicação de Matrizes - Algoritmo Recursivo

MinimoMultiplicacoesRecursivo(b, i, j)

▷ **Entrada:** Vetor b com as dimensões das matrizes e os índices i e j que delimitam o início e término da subcadeia.
▷ **Saída:** O número mínimo de multiplicações escalares necessário para computar a multiplicação da subcadeia. Esse valor é registrado em uma tabela ($m[i,j]$), bem como o índice da divisão em subcadeias ótimas ($s[i,j]$).

1. **se** $i = j$ **então retorne**(0)
2. $m[i,j] := \infty$
3. **para** $k := i$ **até** $j - 1$ **faça**
4. $\quad q :=$ *MinimoMultiplicacoesRecursivo*$(b, i, k)+$ *MinimoMultiplicacoesRecursivo*$(b, k+1, j) + b[i-1] * b[k] * b[j]$
5. $\quad$ **se** $m[i,j] > q$ **então**
6. $\quad\quad m[i,j] := q$; $s[i,j] := k$
7. **retorne**$(m[i,j])$.

Efetuando a Multiplicação Ótima

- É muito fácil efetuar a multiplicação da cadeia de matrizes com o número mínimo de multiplicações escalares usando a tabela s, que registra os índices ótimos de divisão em subcadeias.

MultiplicaMatrizes(M, s, i, j)

▷ **Entrada:** Cadeia de matrizes M, a tabela s e os índices i e j que delimitam a subcadeia a ser multiplicada.
▷ **Saída:** A matriz resultante da multiplicação da subcadeia entre i e j, efetuando o mínimo de multiplicações escalares.

1. **se** $i < j$ **então**
2. $\quad X := MultiplicaMatrizes(M, s, i, s[i, j])$
3. $\quad Y := MultiplicaMatrizes(M, s, s[i, j] + 1, j)$
4. $\quad$ **retorne**$(Multiplica(X, Y, b[i-1], b[s[i, j]], b[j]))$
5. **senão retorne**(M_i);

Algoritmo Recursivo - Complexidade

- O número mínimo de operações feita pelo algoritmo recursivo é dada pela recorrência:

$$T(n) \geq \begin{cases} 1, & n = 1 \\ 1 + \sum_{k=1}^{n-1}[T(k) + T(n-k) + 1] & n > 1, \end{cases}$$

- Portanto, $T(n) \geq 2\sum_{k=1}^{n-1} T(i) + n$, para $n > 1$.
- É possível provar (por substituição) que $T(n) \geq 2^{n-1}$, ou seja, o algoritmo recursivo tem complexidade $\Omega(2^n)$, ainda impraticável !

Algoritmo Recursivo - Complexidade

- A ineficiência do algoritmo recursivo deve-se à sobreposição de subproblemas: o cálculo do mesmo $m[i, j]$ pode ser requerido em vários subproblemas.
- Por exemplo, para $n = 4$, $m[1, 2]$, $m[2, 3]$ e $m[3, 4]$ são computados duas vezes.
- O número de total de $m[i, j]$'s calculados é $O(n^2)$ apenas !
- Portanto, podemos obter um algoritmo mais eficiente se evitarmos recálculos de subproblemas.

Memorização x Programação Dinâmica

- Existem duas técnicas para evitar o recálculo de subproblemas:
  - **Memorização:** Consiste em manter a estrutura recursiva do algoritmo, registrando em uma tabela o valor ótimo para subproblemas já computados e verificando, antes de cada chamada recursiva, se o subproblema a ser resolvido já foi computado.
  - **Programação Dinâmica:** Consiste em preencher uma tabela que registra o valor ótimo para cada subproblema de forma apropriada, isto é, a computação do valor ótimo de cada subproblema depende somente de subproblemas já previamente computados. Elimina completamente a recursão.

Algoritmo de Memorização

MinimoMultiplicacoesMemorizado(b, n)

1. **para** $i := 1$ **até** n **faça**
2. **para** $j := 1$ **até** n **faça**
3. $m[i, j] := \infty$
4. **retorne**$(Memorizacao(b, 1, n))$

Memorizacao(b, i, j)

1. **se** $m[i, j] < \infty$ **então retorne**$(m[i, j])$
2. **se** $i = j$ **então** $m[i, j] := 0$
3. **senão**
4. **para** $k := i$ **até** $j - 1$ **faça**
5. $q := Memorizacao(b, i, k) +$ $Memorizacao(b, k + 1, j) + b[i - 1] * b[k] * b[j]$;
6. **se** $m[i, j] > q$ **então** $m[i, j] := q$; $s[i, j] := k$
7. **retorne**$(m[i, j])$

Algoritmo de Programação Dinâmica

- O uso de programação dinâmica é preferível pois elimina completamente o uso de recursão.
- O algoritmo de programação dinâmica para o problema da multiplicação de matrizes torna-se trivial se computarmos, para valores crescentes de u, o valor ótimo de todas as subcadeias de tamanho u.

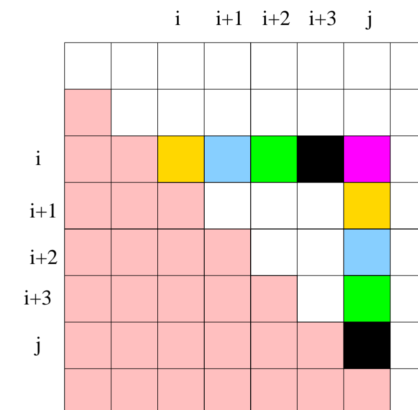
Algoritmo de Programação Dinâmica

MinimoMultiplicacoes(b)

▷ **Entrada:** Vetor b com as dimensões das matrizes.
▷ **Saída:** As tabelas m e s preenchidas.

1. **para** $i = 1$ **até** n **faça** $m[i, i] := 0$
   ▷ calcula o mínimo de todas sub-cadeias de tamanho $u + 1$
2. **para** $u = 1$ **até** $n - 1$ **faça**
3. **para** $i = 1$ **até** $n - u$ **faça**
4. $j := i + u$; $m[i, j] := \infty$
5. **para** $k = 1$ **até** $j - 1$ **faça**
6. $q := m[i, k] + m[k + 1, j] + b[i - 1] * b[k] * b[j]$
7. **se** $q < m[i, j]$ **então**
8. $m[i, j] := q$; $s[i, j] := k$
9. **retorne**(m,s)

Algoritmo de Programação Dinâmica - Exemplo



Algoritmo de Programação Dinâmica - Exemplo

m

	1	2	3	4
1	0			
2		0		
3			0	
4				0

s

	1	2	3	4
1	_			
2		_		
3			_	
4				_

Algoritmo de Programação Dinâmica - Exemplo

m

	1	2	3	4
1	0	12000		
2		0	1200	
3			0	3000
4				0

s

	1	2	3	4
1	_	1		
2		_	2	
3			_	3
4				_

{ 200, 2, 30, 20, 5 }

Algoritmo de Programação Dinâmica - Exemplo

m

	1	2	3	4
1	0	12000	9200	
2		0	1200	
3			0	3000
4				0

s

	1	2	3	4
1	_	1	1	
2		_	2	
3			_	3
4				_

{ 200, 2, 30, 20, 5 }

b0*b1*b3=200*2*20=8000

b0*b2*b3=200*30*20=120000

Algoritmo de Programação Dinâmica - Exemplo

m

	1	2	3	4
1	0	12000	9200	
2		0	1200	1400
3			0	3000
4				0

s

	1	2	3	4
1	_	1	1	
2		_	2	3
3			_	3
4				_

{ 200, 2, 30, 20, 5 }

b1*b2*b4=2*30*5=300

b1*b3*b4=2*20*5=200

Algoritmo de Programação Dinâmica - Exemplo

m:

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3400
4				0

s:

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

b0*b1*b4=200*2*5=2000

b0*b2*b4=200*30*5=30000

b0*b3*b4=200*20*5=20000

{ 200, 2, 30, 20, 5 }

Algoritmo de Programação Dinâmica - Exemplo

m:

	1	2	3	4
1	0	12000	9200	3400
2		0	1200	1400
3			0	3000
4				0

s:

	1	2	3	4
1	_	1	1	1
2		_	2	3
3			_	3
4				_

M1 ((M2 . M3) . M4)

Algoritmo de Programação Dinâmica - Complexidade

- A complexidade de tempo do algoritmo é dada por:

$$\begin{aligned} T(n) &= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} \sum_{k=i}^{i+u-1} \Theta(1) \\ &= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} u \, \Theta(1) \\ &= \sum_{u=1}^{n-1} u(n-u) \, \Theta(1) \\ &= \sum_{u=1}^{n-1} (nu - u^2) \, \Theta(1). \end{aligned}$$

Algoritmo de Programação Dinâmica - Complexidade

- Como
$$\sum_{u=1}^{n-1} nu = n^3/2 - n^2/2$$
e
$$\sum_{u=1}^{n-1} u^2 = n^3/3 - n^2/2 + n/6.$$
Então,
$$T(n) = (n^3/6 - n/6)\, \Theta(1).$$
- A complexidade de tempo do algoritmo é $\Theta(n^3)$.
- A complexidade de espaço é $\Theta(n^2)$, já que é necessário armazenar a matriz com os valores ótimos dos subproblemas.

O Problema Binário da Mochila

O Problema da Mochila

Dada uma mochila de capacidade W (inteiro) e um conjunto de n itens com tamanho w_i (inteiro) e valor c_i associado a cada item i, queremos determinar quais itens devem ser colocados na mochila de modo a **maximizar** o valor total transportado, respeitando sua capacidade.

- Podemos fazer as seguintes suposições:
  - $\sum_{i=1}^{n} w_i > W$;
  - $0 < w_i \leq W$, para todo $i = 1, \ldots, n$.

O Problema Binário da Mochila

- Podemos resolver o problema da mochila com Programação Linear Inteira:
  - Criamos uma variável x_i para cada item: $x_i = 1$ se o item i estiver na solução ótima e $x_i = 0$ caso contrário.
  - A modelagem do problema é simples:

$$\max \sum_{i=1}^{n} c_i x_i \quad (1)$$

$$\sum_{i=1}^{n} w_i x_i \leq W \quad (2)$$

- (1) é a função objetivo e (2) o conjunto de restrições.

O Problema Binário da Mochila

- Como podemos projetar um algoritmo para resolver o problema ?
- Existem 2^n possíveis subconjuntos de itens: um algoritmo de força bruta é impraticável !
- É um problema de otimização. Será que tem subestrutura ótima ?
- Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n - 1$ primeiros itens.
- Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n - 1$ primeiros itens.

O Problema Binário da Mochila

- Seja $z[k, d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original.
- A fórmula de recorrência para computar $z[k, d]$ para todo valor de d e k é:

$$z[0, d] = 0$$

$$z[k, 0] = 0$$

$$z[k, d] = \begin{cases} z[k-1, d], & \text{se } w_k > d \\ \max\{z[k-1, d], z[k-1, d-w_k] + c_k\}, & \text{se } w_k \leq d \end{cases}$$

O Problema Binário da Mochila - Complexidade Recursão

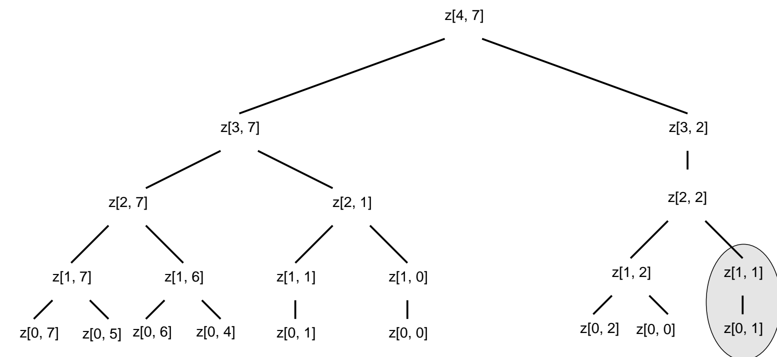
- A complexidade do algoritmo recursivo para este problema no pior caso é dada pela recorrência:

$$T(k,d) = \begin{cases} 1, & k = 0 \text{ ou } d = 0 \\ T(k-1,d) + T(k-1,d-w_k) + 1 & k > 0 \text{ e } d > 0. \end{cases}$$

- Portanto, no pior caso, o algoritmo recursivo tem complexidade $\Omega(2^n)$. É impraticável !
- Mas há sobreposição de subproblemas: o recálculo de subproblemas pode ser evitado !

Mochila - Sobreposição de Subproblemas

- Considere vetor de tamanhos $w = \{2,1,6,5\}$ e capacidade da mochila $W = 7$. A árvore de recursão seria:

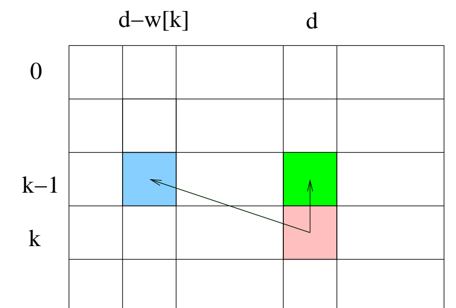


- O subproblema $z[1,1]$ é computado duas vezes.

Mochila - Programação Dinâmica

- O número total máximo de subproblemas a serem computados é nW.
- Isso porque tanto o tamanho dos itens quanto a capacidade da mochila são inteiros !
- Podemos então usar programação dinâmica para evitar o recálculo de subproblemas.
- Como o cálculo de $z[k,d]$ depende de $z[k-1,d]$ e $z[k-1,d-w_k]$, preenchemos a tabela linha a linha.

Mochila



z[k,d]=max { z[k–1,d] , z[k–1,d–w[k]] + c[k] }

O Problema Binário da Mochila - Algoritmo

MochilaMaximo(c, w, W, n)

▷ **Entrada:** Vetores c e w com valor e tamanho de cada item, capacidade W da mochila e número de itens n.
▷ **Saída:** O valor máximo do total de itens colocados na mochila.

1. **para** $d := 0$ **até** W **faça** $z[0, d] := 0$
2. **para** $k := 1$ **até** n **faça** $z[k, 0] := 0$
3. **para** $k := 1$ **até** n **faça**
4.   **para** $d := 1$ **até** W **faça**
5. $z[k, d] := z[k-1, d]$
6.   **se** $w_k \leq d$ **e** $c_k + z[k-1, d-w_k] > z[k, d]$ **então**
7. $z[k, d] := c_k + z[k-1, d-w_k]$
8. **retorne**($z[n, W]$)

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Complexidade

- Claramente, a complexidade do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É um algoritmo pseudo-polinomial: sua complexidade depende do valor de W, parte da entrada do problema.
- O algoritmo não dá o subconjunto de valor total máximo, apenas o valor máximo.
- É fácil recuperar o subconjunto a partir da tabela z preenchida.

Mochila - Recuperação da Solução

MochilaSolucao(z, n, W)

▷ **Entrada:** Tabela z preenchida, capacidade W da mochila e número de itens n.
▷ **Saída:** O vetor x que indica os itens colocados na mochila.

para $i := 1$ **até** n **faça** $x[i] := 0$
MochilaSolucaoAux(x, z, n, W)
retorne(x)

MochilaSolucaoAux(x, z, k, d)

se $k \neq 0$ **então**
  se $z[k, d] = z[k-1, d]$ **então**
    $x[k] := 0$; *MochilaSolucaoAux*$(x, z, k-1, d)$
  senão
    $x[k] := 1$; *MochilaSolucaoAux*$(x, z, k-1, d-w_k)$

Mochila - Exemplo

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	**34**

Mochila - Exemplo

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	**10**	17	17	17	25	32
4	0	7	10	17	17	24	31	**34**

Mochila - Exemplo

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	**10**	17	17	17	17	17
3	0	7	**10**	17	17	17	25	32
4	0	7	10	17	17	24	31	**34**

Mochila - Exemplo

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Exemplo

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

x[1] = x[4] = 1 , x[2] = x[3] = 0

Mochila - Complexidade

- O algoritmo de recuperação da solução tem complexidade $O(n)$.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É possível economizar memória, registrando duas linhas: a que está sendo preenchida e a anterior. Mas isso inviabiliza a recuperação da solução.

Máxima subcadeia comum

Definição: Subcadeia

Dada uma cadeia $S = \{a_1, \ldots, a_n\}$, $S' = \{b_1, \ldots, b_p\}$ é uma *subcadeia* de S se existem p índices $i(j)$ satisfazendo:

(a) $i(j) \in \{1, \ldots, n\}$ para todo $j \in \{1, \ldots, p\}$;

(b) $i(j) < i(j+1)$ para todo $j \in \{1, \ldots, p-1\}$;

(c) $b_j = a_{i(j)}$ para todo $j \in \{1, \ldots, p\}$.

- **Exemplo:** $S = \{ABCDEFG\}$ e $S' = \{ADFG\}$.

Problema da Máxima Subcadeia Comum

Dadas duas cadeias de caracteres X e Y de um alfabeto Σ, determinar a maior subcadeia comum de X e Y

Máxima subcadeia comum (cont.)

- É um problema de otimização. Será que tem subestrutura ótima ?
- **Notação:** Seja S uma cadeia de tamanho n. Para todo $i = 1, \dots, n$, o prefixo de tamanho i de S será denotado por S_i.
- **Exemplo:** Para $S = \{ABCDEFG\}$, $S_2 = \{AB\}$ e $S_4 = \{ABCD\}$.
- **Definição:** $c[i,j]$ é o tamanho da maior subcadeia comum entre os prefixos X_i e Y_j. Logo, se $|X| = m$ e $|Y| = n$, $c[m,n]$ é o valor ótimo.

Máxima subcadeia comum (cont.)

- **Teorema (subestrutura ótima):** Seja $Z = \{z_1, \dots, z_k\}$ a maior subcadeia comum de $X = \{x_1, \dots, x_m\}$ e $Y = \{y_1, \dots, y_n\}$, denotado por $Z = \texttt{MSC}(X, Y)$.
  1. Se $x_m = y_n$ então $z_k = x_m = y_n$ e $Z_{k-1} = \texttt{MSC}(X_{m-1}, Y_{n-1})$.
  2. Se $x_m \neq y_n$ então $z_k \neq x_m$ implica que $Z = \texttt{MSC}(X_{m-1}, Y)$.
  3. Se $x_m \neq y_n$ então $z_k \neq y_n$ implica que $Z = \texttt{MSC}(X, Y_{n-1})$.
- **Fórmula de Recorrência:**

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Máxima subcadeia comum (cont.)

```
MSC(X, m, Y, n, c, b)
01. para i = 1 até m faça c[i,0] := 0
02. para j = 1 até n faça c[0,j] := 0
03. para i = 1 até m faça
04.     para j = 1 até n faça
05.         se x_i = y_j então
06.             c[i,j] := c[i−1, j−1] + 1 ; b[i,j] := "↖"
07.         senão
08.             se c[i, j−1] > c[i−1, j] então
09.                 c[i,j] := c[i, j−1] ; b[i,j] := "←"
10.             senão
11.                 c[i,j] := c[i−1, j]; b[i,j] := "↑";
12. retorne(c[m,n], b).
```


Máxima subcadeia comum - Exemplo

- **Exemplo:** $X = \{a, b, c, b\}$ e $Y = \{b, d, c, a, b\}$, $m = 4$ e $n = 5$.

X \ Y		b	d	c	a	b
	0	1	2	3	4	5
0	0	0	0	0	0	0
a 1	0	0	0	0	1	1
b 2	0	1	1	1	1	2
c 3	0	1	1	2	2	2
b 4	0	1	1	2	2	3

X \ Y		(b)	d	(c)	a	(b)
	0	1	2	3	4	5
0						
a 1		↑	↑	↑	↖	←
(b) 2		↖	←	←	↑	↖
(c) 3		↑	↑	↖	←	↑
(b) 4		↖	↑	↑	↑	↖

Máxima subcadeia comum - Complexidade

- Claramente, a complexidade do algoritmo é $O(mn)$.
- O algoritmo não encontra a subcadeia comum de tamanho máximo, apenas seu tamanho.
- Com a tabela b preenchida, é fácil encontrar a subcadeia comum máxima.

Máxima subcadeia comum (cont.)

- Para recuperar a solução, basta chamar $Recupera_MSC(b, X, m, n)$.

Recupera_MSC(b, X, i, j)

1. **se** $i = 0$ e $j = 0$ **então retorne**
2. **se** $b[i, j] =$ "↖" **então**
3. $Recupera_MSC(b, X, i-1, j-1)$; **imprima** x_i
4. **senão**
5. **se** $b[i, j] =$ "↑" **então**
6. $Recupera_MSC(b, X, i-1, j)$
7. **senão**
8. $Recupera_MSC(b, X, i, j-1)$

Máxima subcadeia comum - Complexidade

- A determinação da subcadeia comum máxima é feita em tempo $O(m + n)$ no pior caso.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da subcadeia comum máxima é $O(mn)$.
- Note que a tabela b é dispensável, podemos economizar memória recuperando a solução a partir da tabela c. Ainda assim, o gasto de memória seria $O(mn)$.
- Caso não haja interesse em determinar a subcadeia comum máxima, mas apenas seu tamanho, é possível reduzir o gasto de memória para $O(\min\{n + m\})$: basta registrar apenas a linha da tabela sendo preenchida e a anterior.

Algoritmos gulosos

Algoritmos Gulosos: Conceitos Básicos

- Tipicamente algoritmos gulosos são utilizados para resolver problemas de **otimização**.
- Uma característica comum dos problemas onde se aplicam algoritmos gulosos é a existência **subestrutura ótima**, semelhante à programação dinâmica !
- Programação dinâmica: tipicamente os subproblemas são resolvidos à otimalidade **antes** de se proceder à escolha de um elemento que irá compor a solução ótima
- Algoritmo Guloso: primeiramente é feita a escolha de um elemento que irá compor a solução ótima e só **depois** um subproblema é resolvido.

Algoritmos Gulosos: Conceitos Básicos

- Um algoritmo guloso sempre faz a escolha que parece ser a *"melhor"* a cada iteração.
- Ou seja, a escolha é feita de acordo com um **critério guloso** ! É uma decisão **localmente** ótima !
- Propriedade da **escolha gulosa**: garante que a cada iteração é tomada uma decisão que irá levar a um ótimo global.
- Em um algoritmo guloso uma escolha que foi feita **nunca é revista**, ou seja, não há qualquer tipo de *backtracking*.

Seleção de Atividades

- $S = \{a_1, \ldots, a_n\}$: conjunto de n atividades que podem ser executadas em um mesmo local. Exemplo: palestras em um auditório.
- Para todo $i = 1, \ldots, n$, a atividade a_i começa no instante s_i e termina no instante f_i, com $0 \leq s_i < f_i < \infty$.
  Ou seja, supõe-se que a atividade a_i será executada no intervalo de tempo (semi-aberto) $[s_i, f_i)$.

Definição

As atividades a_i e a_j são ditas **compatíveis** se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ são disjuntos.

Problema de Seleção de Atividades

Encontre em S um subconjunto de atividades mutuamente compatíveis que tenha tamanho **máximo**.

Seleção de Atividades

- Exemplo:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- Pares de atividades incompatíveis: (a_1, a_2), (a_1, a_3)
  Pares de atividades compatíveis: (a_1, a_4), (a_4, a_8)
- Conjunto **maximal** de atividades compatíveis: (a_3, a_9, a_{11}).
- Conjuntos **máximos** de atividades compatíveis: (a_1, a_4, a_8, a_{11}) e (a_2, a_4, a_9, a_{11})

As atividades estão ordenadas em ordem monotonamente crescente de tempos de término ! Isso será importante mais adiante ...

Seleção de Atividades

- Vimos que tanto os algoritmos gulosos quanto aqueles que usam programação dinâmica valem-se da existência da propriedade de subestrutura ótima.
- Inicialmente verificaremos que o problema da seleção de atividades tem esta propriedade e, então, projetaremos um algoritmo por programação dinâmica.
- Em seguida, mostraremos que há uma forma de resolver uma quantidade consideravelmente menor de subproblemas do que é feito na programação dinâmica.
- Isto será garantido por uma **propriedade de escolha gulosa**, a qual dará origem a um algoritmo guloso.
- Este processo auxiliará no entendimento da diferença entre estas duas técnicas de projeto de algoritmos.

Seleção de Atividades

Definição

$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$, i.e., o conjunto de tarefas que começam depois do término de a_i e terminam antes do início de a_j.

- Tarefas artificiais: a_0 com $f_0 = 0$ e a_{n+1} com $s_{n+1} = \infty$
- Tem-se que $S = S_{0,n+1}$ e, com isso, S_{ij} está bem definido para qualquer par (i, j) tal que $0 \leq i, j \leq n + 1$.
- Supondo que $f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_n < f_{n+1}$, ou seja, que as tarefas estão ordenadas em ordem crescente de tempos de término, pode-se concluir que $S_{ij} = \emptyset$ para todo $i \geq j$.
  Por quê ?

Seleção de Atividades

- **Subestrutura ótima**: considere o *subproblema* da seleção de atividades definido sobre S_{ij}. Suponha que a_k pertence a uma solução ótima de S_{ij}. Como $f_i \leq s_k < f_k \leq s_j$, uma solução ótima para S_{ij} que contenha a_k será composta pelas atividades de uma solução ótima de S_{ik}, pelas atividades de uma solução ótima de S_{kj} e por a_k. **Por quê ?**
- **Definição**: para todo $0 \leq i, j \leq n + 1$, seja $c[i, j]$ o valor ótimo do problema de seleção de atividades para a instância S_{ij}.
- Deste modo, o valor ótimo do problema de seleção de atividades para instância $S = S_{0,n+1}$ é $c[0, n + 1]$.
- **Fórmula de recorrência**:

$$c[i,j] = \begin{cases} 0 & \text{se } S_{ij} = \emptyset \\ \max\limits_{i<k<j : a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{se } S_{ij} \neq \emptyset \end{cases}$$

Agora é fácil escrever o algoritmo de programação dinâmica ...

Seleção de Atividades

Podemos "converter" o algoritmo de programação dinâmica em um algoritmo guloso se notarmos que o primeiro resolve subproblemas desnecessariamente.

Teorema: (escolha gulosa)

Considere o subproblema definido para uma instância não-vazia S_{ij}, e seja a_m a atividade de S_{ij} com o menor tempo de término, i.e.:

$$f_m = \min\{f_k : a_k \in S_{ij}\}.$$

Então **(a)** existe uma solução ótima para S_{ij} que contém a_m e **(b)** S_{im} é vazio e o subproblema definido para esta instância é trivial, portanto, a escolha de a_m deixa apenas um dos subproblemas com solução possivelmente não-trivial, já que S_{mj} pode não ser vazio.

Prova: ... $\Box$.

Seleção de Atividades

SelecionaAtivGulosoRec(s, f, i, j)

▷ **Entrada:** vetores s e f com instantes de início e término das atividades $i, i+1, \ldots, j$, sendo $f_i \leq \ldots \leq f_j$.
▷ **Saída:** conjunto de tamanho máximo de índices de atividades mutuamente compatíveis.

1. $m \leftarrow i + 1$;
 ▷ Busca atividade com menor tempo de término que pode estar em S_{ij}
2. **enquanto** $m < j$ **e** $s_m < f_i$ **faça** $m \leftarrow m + 1$;
3. **se** $m \geq j$ **então retorne** $\emptyset$;
4. **senão**
5. **se** $f_m > s_j$ **então retorne** $\emptyset$; ▷ $a_m \notin S_{ij}$
6. **senão retorne** $\{a_m\} \cup$ SelecionaAtivGulosoRec(s, f, m, j).

Seleção de Atividades

- A chamada inicial será *SelecionaAtivGulosoRec*$(s, f, 0, n+1)$.
- **Complexidade**: $\Theta(n)$.
  Ao longo de todas as chamadas recursivas,cada atividade é examinada exatamente uma vez no laço da linha **2**. Em particular, a atividade a_k é examinada na última chamada com $i < k$.
- Como o algoritmo anterior é um caso simples de recursão caudal, é trivial escrever uma versão iterativa do mesmo ...

Seleção de Atividades

SelecionaAtivGulosoIter(s, f, n)

▷ **Entrada:** vetores s e f com instantes de início e término das n atividades com os instantes de término em ordem monotonamente crescente.
▷ **Saída:** o conjunto A de tamanho máximo contendo atividades mutuamente compatíveis.

1. $A \leftarrow \{a_1\}$;
2. $i \leftarrow 1$;
3. **para** $m \leftarrow 2$ **até** n **faça**
4. **se** $s_m \geq f_i$ **então**
5. $A \leftarrow A \cup \{a_m\}$;
6. $i \leftarrow m$;
7. **retorne** A.

Seleção de Atividades

- **Invariante**: i é sempre o índice da última atividade colocada em A. Como as atividades estão em ordenadas pelo instante de término, tem-se que:

$$f_i = \max\{f_k : a_k \in A\},$$

  ou seja, f_i é sempre o maior instante de término de uma atividade em A.
- **Complexidade**: $\Theta(n)$.

Códigos de Huffman

- Códigos de Huffman: técnica de compressão de dados.
- Reduções no tamanho dos arquivos dependem das características dos dados contidos nos mesmos. Valores típicos oscilam entre 20 e 90%.
- Exemplo: arquivo texto contendo 100.000 caracteres no alfabeto $\Sigma = \{a, b, c, d, e, f\}$. As freqüências de cada caracter no arquivo são indicadas na tabela abaixo.
- Codificação do arquivo: representar cada caracter por uma seqüência de *bits*
- Alternativas:
  1. seqüências de tamanho fixo.
  2. seqüências de tamanho variável.

	a	b	c	d	e	f
Freqüência	45	13	12	16	9	5
Código de tamanho fixo	000	001	010	011	100	101
Código de tamanho variável	0	101	100	111	1101	1100

Códigos de Huffman

- Qual o tamanho (em *bits*) do arquivo comprimido usando os códigos acima ?
- Códigos de tamanho fixo: $3 \times 100.000 = 300.000$
  Códigos de tamanho variável:

$$(\underbrace{45 \times 1}_{a} + \underbrace{13 \times 3}_{b} + \underbrace{12 \times 3}_{c} + \underbrace{16 \times 3}_{d} + \underbrace{9 \times 4}_{e} + \underbrace{5 \times 4}_{f}) \times 1.000 = 224.000$$

Ganho de $\approx 25\%$ em relação à solução anterior !

Problema da Codificação:
Dadas as freqüências de ocorrência dos caracteres de um arquivo, encontrar as seqüências de *bits* (códigos) para representá-los de modo que o arquivo comprimido tenha tamanho mínimo.

Códigos de Huffman

Definição:
Códigos livres de prefixo são aqueles onde, dados dois caracteres quaisquer i e j representados pela codificação, a seqüência de *bits* associada a i **não** é um *prefixo* da seqüência associada a j.

Importante:
Pode-se provar que sempre **existe** uma solução ótima do problema da codificação que é dado por um código *livre de prefixo*.

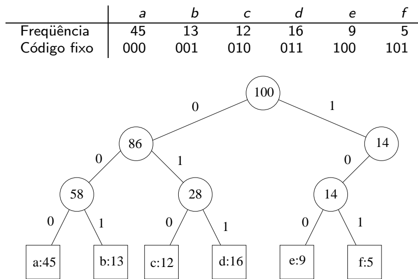
- O processo de codificação, i.e, de geração do arquivo comprimido é sempre fácil pois reduz-se a concatenar os códigos dos caracteres presentes no arquivo original em seqüência.
  Exemplo: usando a codificação de tamanho variável do exemplo anterior, o arquivo original dado por abc seria codificado por 0101100.

Códigos de Huffman

- A vantagem dos códigos livres de prefixo se torna evidente quando vamos decodificar o arquivo comprimido.
  Como nenhum código é prefixo de outro código, o código que se encontra no início do arquivo comprimido não apresenta ambigüidade. Pode-se simplesmente identificar este código inicial, traduzí-lo de volta ao caracter original e repetir o processo no restante do arquivo comprimido.
  Exemplo: usando a codificação de tamanho variável do exemplo anterior, o arquivo comprimido contendo os *bits* 001011101 divide-se de **forma unívoca** em 0 0 101 1101, ou seja, corresponde ao arquivo original dado por *aabe*.
- Como representar de maneira conveniente uma codificação livre de prefixo de modo a facilitar o processo de decodificação ?

Códigos de Huffman

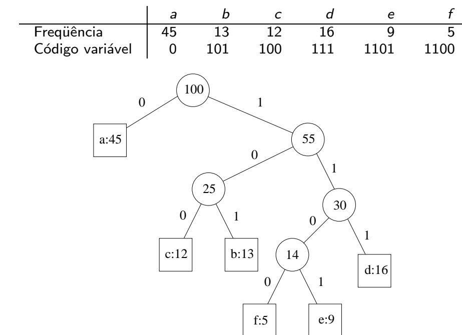
- Solução: usar uma árvore binária. O *filho esquerdo* está associado ao *bit* ZERO enquanto o *filho direito* está associado ao *bit* UM. Nas folhas encontram-se os caracteres presentes no arquivo original.
- Vejamos como ficam as árvores que representam os códigos do exemplo anterior.



	a	*b*	*c*	*d*	*e*	*f*
Freqüência	45	13	12	16	9	5
Código fixo	000	001	010	011	100	101

Códigos de Huffman

- Solução: usar uma árvore binária. O *filho esquerdo* está associado ao *bit* ZERO enquanto o *filho direito* está associado ao *bit* UM. Nas folhas encontram-se os caracteres presentes no arquivo original.
- Vejamos como ficam as árvores que representam os códigos do exemplo anterior.



	a	*b*	*c*	*d*	*e*	*f*
Freqüência	45	13	12	16	9	5
Código variável	0	101	100	111	1101	1100

Códigos de Huffman

- Pode-se mostrar (exercício !) que uma codificação ótima sempre é representada por uma árvore binária *cheia*, na qual cada vértice interno tem exatamente dois filhos.
  A codificação com tamanho fixo do exemplo anterior não é ótima ...
- Então podemos restringir nossa atenção às árvores binárias cheias com $|C|$ folhas e $|C|-1$ vértices internos (exercício !), onde C é o conjunto de caracteres do alfabeto no qual está escrito o arquivo original.
- Computando o tamanho do arquivo comprimido:
  Se T é a árvore que representa a codificação, $d_T(c)$ é a profundidade da folha representado o caracter c e $f(c)$ é a sua freqüência, o tamanho do arquivo comprimido será dado por:

$$B(T) = \sum_{c \in C} f(c) d_T(c).$$

  Diremos que $B(T)$ é o **custo** da árvore T.

Códigos de Huffman

- **Idéia do algoritmo de Huffman**: Começar com $|C|$ folhas e realizar sequencialmente $|C|-1$ operações de "intercalação" de dois vértices da árvore.
  Cada uma destas intercalações dá origem a um novo vértice interno, que será o pai dos vértices que participaram da intercalação.
- A escolha do par de vértices que dará origem a intercalação em cada passo depende da soma das freqüências das folhas das subárvores com raízes nos vértices que ainda não participaram de intercalações.

Algoritmo de Huffman

Huffman(C)

▷ **Entrada:** Conjunto de caracteres C e as freqüências f dos caracteres em C.
▷ **Saída:** raiz de uma árvore binária representando uma codificação ótima livre de prefixos.

1. $n \leftarrow |C|$;
   ▷ Q é fila de prioridades dada pelas freqüências dos vértices ainda não intercalados
2. $Q \leftarrow C$;
3. **para** $i \leftarrow 1$ **até** $n-1$ **faça**
4. **alocar novo registro** z; ▷ vértice de T
5. $z.esq \leftarrow x \leftarrow$ EXTRAI_MIN(Q);
6. $z.dir \leftarrow x \leftarrow$ EXTRAI_MIN(Q);
7. $z.f \leftarrow x.f + y.f$;
8. INSERE(Q, z);
9. **retorne** EXTRAI_MIN(Q).

Corretude do algoritmo de Huffman

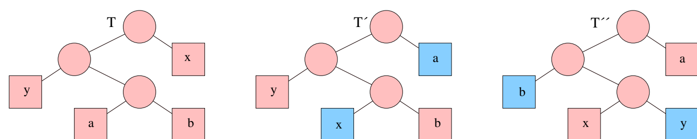
Lema 1: (escolha gulosa)

Seja C um alfabeto onde cada caracter $c \in C$ tem freqüência $f[c]$. Sejam x e y dois caracteres em C com as **menores** freqüências. Então, existe **um** código ótimo livre de prefixo para C no qual os códigos para x e y tem o mesmo comprimento e diferem apenas no último bit.

Prova do Lema 1:

- Seja uma árvore **ótima** T.
- Sejam a e b duas folhas "irmãs" (i.e. usadas em uma intercalação) mais profundas de T e x e y as folhas de T de menor freqüência.
- **Idéia:** a partir de T, obter uma outra árvore **ótima** T' com x e y sendo duas folhas "irmãs".

Corretude do algoritmo de Huffman



$$
\begin{aligned}
B(T) - B(T') &= \textstyle\sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\
&= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\
&= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0
\end{aligned}
$$

T' tem custo não superior ao de T.

Argumento análogo comparando os custos de T' e T'' mostram que $B(T) \geq B(T') \geq B(T'')$.

Como T é ótima, o resultado vale. □

Corretude do algoritmo de Huffman

Lema 2: (subestrutura ótima)

Seja C um alfabeto com freqüência $f[c]$ definida para cada caracter $c \in C$. Sejam x e y dois caracteres de C com freqüência mínima. Seja C' o alfabeto obtido pela remoção de x e y e pela inclusão de um **novo** caracter z, ou seja, $C' = C \cup \{z\} - \{x, y\}$. As freqüências dos caracteres em $C' \cap C$ são as mesmas que em C e $f[z]$ é definida como sendo $f[z] = f[x] + f[y]$.
Seja T' uma árvore binária representado um código ótimo livre de prefixo para C'. Então a árvore binária T obtida de T' substituindo-se o vértice (folha) z pela por um vértice interno tendo x e y como fihos, representa uma código ótimo livre de prefixo para C.

Corretude do algoritmo de Huffman

Prova do Lema 2:

- Comparando os custos de T e T':
  - Se $c \in C - \{x, y\}$, $f[c]d_T(c) = f[c]d_{T'}(c)$.
  - $f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = f[z]d_{T'}(z) + (f[x] + f[y])$.
- Logo, $B(T') = B(T) - f[x] - f[y]$.
- Provando o lema por contradição: supor que existe T'' tal que $B(T'') < B(T)$. Pelo lema anterior, podemos supor que x e y são folhas "irmãs" em T''. Seja T''' a árvore obtida de T'' pela substituição de x e y por uma folha z com freqüência $f[z] = f[x] + f[y]$. O custo de T''' é tal que

$$B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T'),$$

contradizendo a hipótese de que T' é uma árvore ótima para C'. □

Corretude do algoritmo de Huffman

Teorema:

O algoritmo de Huffman constrói um código ótimo (livre de prefixo).

Prova: imediata a partir dos Lemas 1 e 2. □

Passos do projeto de algoritmos gulosos: resumo

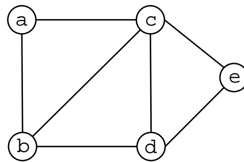
1. Formule o problema como um **problema de otimização** no qual uma escolha é feita, restando-nos então resolver um único subproblema a resolver.
2. Provar que existe sempre uma solução ótima do problema que atende à **escolha gulosa**, ou seja, a escolha feita pelo algoritmo guloso é segura.
3. Demonstrar que, uma vez feita a escolha gulosa, o que resta a resolver é um subproblema tal que se combinarmos a resposta ótima deste subproblema com o(s) elemento(s) da escolha gulosa, chega-se à solução ótima do problema original.
   Esta é a parte que requer mais engenhosidade !
   Normalmente a prova começa com uma solução ótima *genérica* e mostra que ela pode ser modificada (possivelmente após vários passos) até que ela inclua o(s) elemento(s) identificados pela escolha gulosa.

Grafos: Noções Básicas e Representação

Definição de Grafo

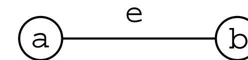
- Um *grafo* $G = (V, E)$ é dado por dois conjuntos finitos: o conjunto de *vértices* V e o conjunto de *arestas* E.
- Cada aresta de E é um par não ordenado de vértices de V.
- **Exemplo:**

$$V = \{a, b, c, d, e\}$$
$$E = \{(a, b), (a, c), (b, c), (b, d), (c, d), (c, e), (d, e)\}$$



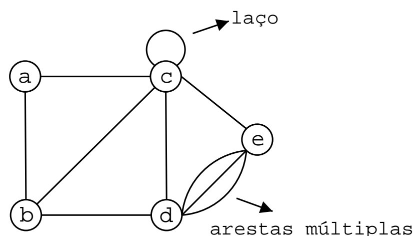
Definição de Grafo

- Dada uma aresta $e = (a, b)$, dizemos que os vértices a e b são os *extremos* da aresta e e que a e b são vértices *adjacentes*.
- Podemos dizer também que a aresta e é *incidente* aos vértices a e b, e que os vértices a e b são incidentes à aresta e.



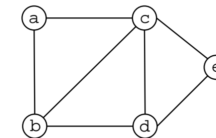
Grafo Simples

- Dizemos que um grafo é *simples* quando não possui laços ou arestas múltiplas.
- Um *laço* é uma aresta com ambos os extremos em um mesmo vértice e *arestas múltiplas* são duas ou mais arestas com o mesmo par de vértices como extremos.
- **Exemplo:**



Tamanho do Grafo

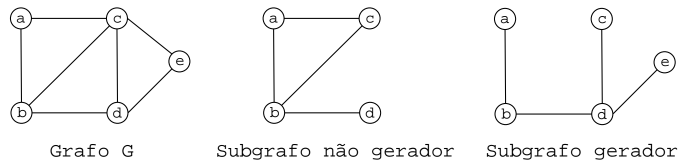
- Denotamos por $|V|$ e $|E|$ a cardinalidade dos conjuntos de vértices e arestas de um grafo G, respectivamente.
- No exemplo abaixo temos $|V| = 5$ e $|E| = 7$.



O *tamanho* do grafo G é dado por $|V| + |E|$.

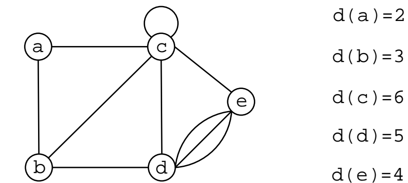
Subgrafo e Subgrafo Gerador

- Um *subgrafo* $H = (V', E')$ de um grafo $G = (V, E)$ é um grafo tal que $V' \subseteq V$, $E' \subseteq E$.
- Um *subgrafo gerador* de G é um subgrafo H com $V' = V$.
- **Exemplo:**



Grau de um vértice

- O *grau* de um vértice v, denotado por $d(v)$ é o número de arestas incidentes a v, com laços contados duas vezes.
- **Exemplo:**

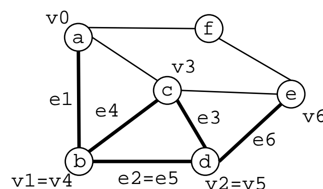


- **Teorema** *Para todo grafo* $G = (V, E)$ *temos:*

$$\sum_{v \in V} d(v) = 2|E|.$$

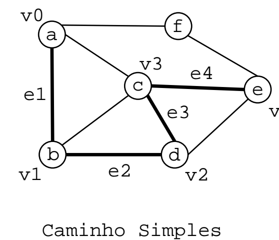
Caminhos em Grafos

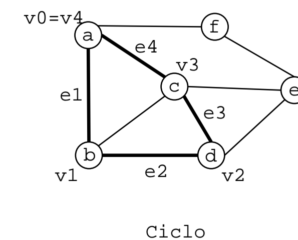
- Um *caminho* P de v_0 a v_n no grafo G é uma seqüência finita e não vazia $(v_0, e_1, v_1, \ldots, e_n, v_n)$ cujos elementos são alternadamente vértices e arestas e tal que, para todo $1 \leq i \leq n$, v_{i-1} e v_i são os extremos de e_i.
- O *comprimento* do caminho P é dado pelo seu número de arestas, ou seja, n.
- **Exemplo:**



Caminhos Simples e Ciclos

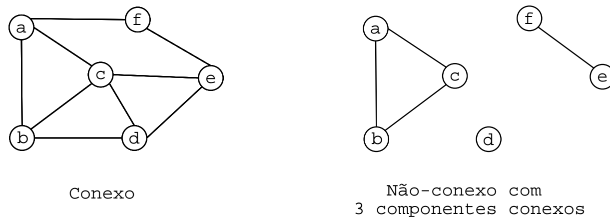
- Um *caminho simples* é um caminho em que não há repetição de vértices e nem de arestas na sequência.
- Um *ciclo* ou *caminho fechado* é uma caminho em que $v_0 = v_n$.
- **Exemplo:**





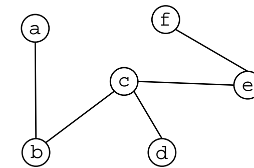
Grafo Conexo

- Dizemos que um grafo é *conexo* se, para qualquer par de vértices u e v de G, existe um caminho de u a v em G.
- Quando o grafo G não é conexo, podemos particionar em *componentes conexos*. Dois vértices u e v de G estão no mesmo componente conexo de G se há caminho de u a v em G.
- **Exemplo:**



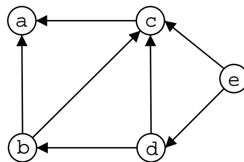
Árvore

- Um grafo G é uma *árvore* se é conexo e não possui ciclos (acíclico). Ou equivalentemente, G é árvore se:
  - É conexo com $|V| - 1$ arestas.
  - É conexo e a remoção de qualquer aresta desconecta o grafo (minimal conexo).
  - Para todo par de vértices u, v de G, existe exatamente um caminho de u a v em G.
- **Exemplo:**



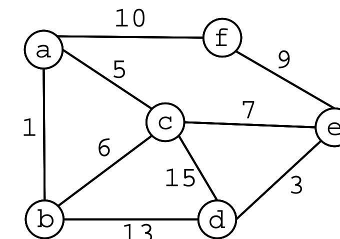
Grafo Orientado

- As definições que vimos até agora são para grafos *não orientados*.
- Um *grafo orientado* é definido de forma semelhante, com a diferença que as arestas (às vezes chamadas de arcos) são dadas por pares ordenados.
- **Exemplo:**



Grafo Ponderado

- Um grafo (orientado ou não) é *ponderado* se a cada aresta e do grafo está associado um valor real $c(e)$, o qual denominamos *custo (ou peso)* da aresta.
- **Exemplo:**



Algoritmos em Grafos - Motivação

- Grafos são estruturas abstratas que podem modelar diversos problemas do mundo real.
- Por exemplo, um grafo pode representar conexões entre cidades por estradas ou uma rede de computadores.
- O interesse em estudar algoritmos para problemas em grafos é que conhecer algoritmo para um único problema em grafos pode significar conhecer algoritmos para diversos problemas reais.

Representação Interna de Grafos

- A complexidade dos algoritmos para solução de problemas modelados por grafos depende fortemente da sua representação interna.
- Existem duas representações canônicas: matriz de adjacências e lista de adjacências.
- O uso de uma ou outra num determinado algoritmo depende da natureza das operações que ditam a complexidade do algoritmo.
- Outras representações podem ser utilizadas mas essas duas são as mais utilizadas por sua simplicidade.
- Para alguns problemas em grafos o uso de estruturas de dados adicionais são fundamentais para o projeto de algoritmos eficientes.

Matriz de adjacências

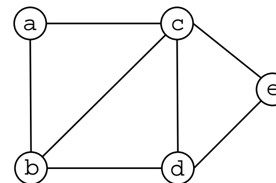
- Um grafo simples $G = (V, E)$, orientado ou não, pode ser representado internamente por uma estrutura de dados chamada *matriz de adjacências*.
- A matriz de ajacências é uma matriz quadrada A de ordem $|V|$, cujas linhas e colunas são indexadas pelos vértices em V, e tal que:

$$A[i,j] = 1 \text{ se } (i,j) \in E \text{ e } 0 \text{ caso contrário.}$$

- Note que se G é não-orientado, então a matriz A correspondente é simétrica.
- Esta definição, para grafos simples, pode ser generalizada: basta fazer $A[i,j]$ ser um registro que contém informações sobre a adjacência dos vértices i e j como multiplicidade ou custo da aresta (i,j).

Matriz de adjacências

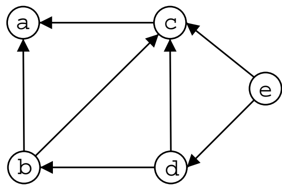
- Veja um exemplo de um grafo não-orientado simples e a matriz de adjacências correspondente.



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	0
c	1	1	0	1	1
d	0	1	1	0	1
e	0	0	1	1	0

Matriz de adjacências

- Veja um exemplo de um grafo orientado simples e a matriz de adjacências correspondente.



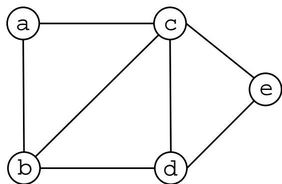
	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	0	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

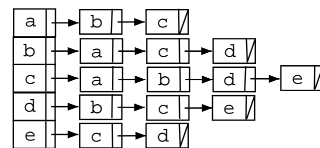
Lista de adjacências

- Uma *lista de adjacências* que representa um grafo simples G é um vetor L indexado pelos vértices em V de forma que, para cada $v \in V$, $L[v]$ é um apontador para o início de uma lista ligada dos vértices que são adjacentes a v em G.
- O tamanho da lista $L[v]$ é precisamente o grau $d(v)$.
- Se G for um grafo não orientado, cada aresta (i, j) é representada duas vezes: uma na lista de adjacências de i e outra na de j.
- Se G for um grafo orientado, a lista ligada $L(v)$ contém apenas os elementos pós-adjacentes (ou pré-adjacentes) a v em G. Assim, cada aresta (ou arco) é representada exatamente uma vez.
- Também podemos usar uma lista ligada para representar grafos não simples ou ponderados: basta que cada nó da lista seja um registro que contenha as demais informações sobre a adjacência de i e j como multiplicidade e custo (ou peso).

Lista de adjacências

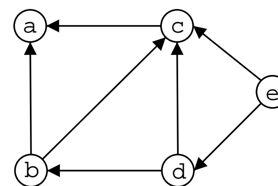
- Veja um exemplo de um grafo não-orientado e a lista de adjacências correspondente.

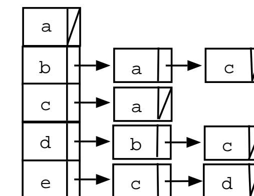




Lista de adjacências

Veja um exemplo de um grafo orientado e a lista de adjacências correspondente.





Matriz × Lista de adjacências

- Cada uma dessas estruturas tem suas vantagens:
  - Matrizes de adjacências são mais adequadas quando queremos descobrir rapidamente se uma dada aresta está ou não presente num grafo, ou os atributos dela.
  - Por outro lado, lista de adjacências são mais adequadas quando queremos determinar todos os vértices adjacentes a um dado vértice.
  - Matrizes de adjacências ocupam espaço proporcional a $|V|^2$ e são, portanto, mais adequadas a grafos densos ($|E| = \Theta(|V|^2)$).
  - Listas de adjacências ocupam espaço proporcional a $|E|$, sendo mais adequadas a grafos esparsos ($|E| = \Theta(|V|)$).
- Em alguns casos pode ser útil ter as duas estruturas simultaneamente.

Buscas em grafos

Buscas em grafos

- Em muitas aplicações em grafos é necessário percorrer rapidamente o grafo visitando-se todos os seus vértices.
- Para que isso seja feito de modo sistemático e organizado são utilizados algoritmos de busca, semelhantes àqueles que já foram vistos para percursos em árvores na disciplina de Estruturas de Dados.
- As buscas são usadas em diversas aplicações para determinar informações relevantes sobre a estrutura do grafo de entrada.
- Além disso, alterações, muitas vezes pequenas, nos algoritmos de busca permitem resolver de forma eficiente problemas mais complexos definidos sobre grafos.
- São dois os algoritmos básicos de busca em grafos: a busca em largura e a busca em profundidade.

Busca em largura

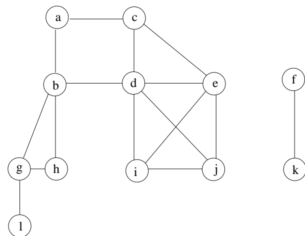
- Diremos que um vértice u é **alcançável** a partir de um vértice v de um grafo G se existe um caminho de v para u em G.
- Definição: um vértice u está a uma **distância** k de um vértice v se k é o comprimento do menor caminho que começa em v e termina em u.

  Se u **não é alcançável** a partir de v, então arbitra-se que a distância entre eles é **infinita**.

Uma busca em largura ou BFS (do inglês *breadth first search*) em um grafo $G = (V, E)$ é um método em que, partindo-se de um vértice especial u denominado *raiz da busca*, percorre-se G visitando-se todos os vértices alcançáveis a partir de u em **ordem crescente de distância**.
Nota: a ordem em que os vértices equidistantes de u são percorridos é irrelevante e depende da forma como as adjacências dos vértices são armazenadas e percorridas.

Busca em largura

No grafo da figura abaixo, assumindo-se o vértice a como sendo a raiz da busca e que as listas de adjacências estão ordenadas alfabeticamente, a ordem de visitação dos vértices seria $\{a, b, c, g, h, d, e, l, i, j\}$. Os vértices f e k não seriam visitados porque não são alcançáveis a partir do vértice a.



Busca em largura

- A busca em largura é um exemplo interessante de aplicação de **filas**.
- O algoritmo descrito a seguir usa um critério de coloração para ir controlando os vértices do grafo que já foram visitados e/ou cujas listas de adjacências já foram exploradas durante a busca.
- Um vértice é identificado como **visitado** no momento em que a busca o atinge pela primeira vez.
- Uma vez visitado, este vértice poderá permitir que sejam atingidos outros vértices ainda não alcançados pela busca, o que é feito no momento em que se explora a sua vizinhança.
- Concluída esta operação, o vértice é dito estar **explorado**.

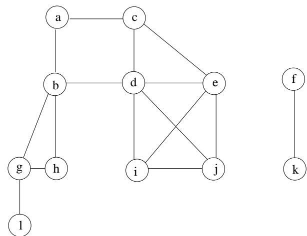
Busca em largura

- São listados abaixo os principais elementos do pseudo-código que descreve a busca em largura.
- o vetor `cor` contendo $|V|$ posições. O elemento `cor`$[i]$ será `branco` se o vértice i ainda não foi visitado, `cinza` se ele foi visitado mas a sua vizinhança não foi explorada e `preto` se o vértice i foi visitado e sua vizinhança já foi explorada.
- o vetor `dist` contendo $|V|$ posições. O elemento `dist`$[i]$ é inicializado com o valor infinito e armazenará a distância entre o vértice i e a raiz da busca.
- o vetor `pred` contendo $|V|$ posições. O elemento `pred`$[i]$ armazenará a informação sobre qual era o vértice cuja adjacência estava sendo explorada quando o vértice i foi visitado pela primeira vez.
- a fila Q armazena a lista dos vértices visitados cuja vizinhança ainda deve ser explorada, ou seja, aqueles de cor cinza.
- As listas de adjacências são acessadas através do vetor `Adj`.

Busca em largura

```
BFS(G,raiz)
▷ Q é o conjunto de vértices a serem explorados (cinzas)
1.  InicializaFila(Q);        ▷ inicializa fila como sendo vazia
2.  Para todo v ∈ V − {raiz} faça
3.      dist[v] ← ∞;    cor[v] ← branco;    pred[v] ← NULO;
4.  dist[raiz] ← 0;   cor[raiz] ← cinza;   pred[raiz] ← NULO;
5.  InsereFila(Q, raiz);
6.  Enquanto (!FilaVazia(Q)) faça
7.      RemoveFila(Q, u)        ▷ tirar u do conjunto Q
8.      Para todo v ∈ Adj[u] faça
9.          Se cor[v] = branco então
10.             dist[v] ← dist[u] + 1;    pred[v] ← u;
11.             cor[v] ← cinza;    InsereFila(Q, v);
12.     cor[u] ← preto;
13. Retorne(dist, pred).
```

Busca em largura: exemplo



Nota: o subgrafo formado por $(V, E_{\texttt{pred}})$, onde $E_{\texttt{pred}} = \{(\texttt{pred}[v], v) : \forall v \in V\}$, é uma árvore.

Busca em largura: complexidade

- Faz-se uma análise agregada onde é contado o *total* de operações efetuadas no laço das linhas 6 a 12 no *pior caso*.
- Supõe-se que o grafo é armazenado por listas de adjacências.
- Depois da inicialização da linha 3, nenhum vértice é colorido como branco. Logo todo vértice é inserido e removido no máximo uma vez e, portanto, o tempo total gasto nas operações de manutenção da fila Q é $O(|V|)$.
- A lista de adjacências de um vértice só é percorrida quando ele é removida da fila. Logo, ela é varrida no máximo uma vez. Como o comprimento total de todas as listas de adjacências é $\Theta(E)$, o tempo total gasto explorando estas listas é $\Theta(E)$.
- O gasto de tempo com as inicializações é claramente $\Theta(V)$. Assim, o tempo total de execução do algoritmo é $O(|V| + |E|)$.
  Ou seja, BFS roda em tempo linear no tamanho da entrada de G quando este é representado por suas listas de adjacências.

Busca em profundidade

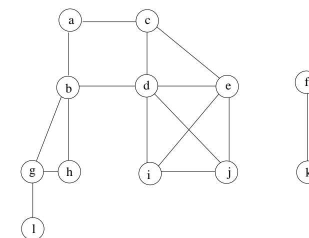
Intuitivamente, pode-se dizer que a estratégia de percorrer um grafo usando uma busca em profundidade ou DFS (do inglês *depth first search*) difere daquela adotada na busca em largura na medida em que, a partir do vértice raiz, ela procura ir o mais "longe" possível no grafo sempre passando por vértices ainda não visitados.

Se ao chegar em um vértice v a busca não consegue avançar, ela retorna ao vértice $\texttt{pred}[v]$ cuja adjacência estava sendo explorada no momento em que v foi visitado pela primeira vez e volta a explorá-la.

É uma generalização do percurso em **pré-ordem** para árvores.

Busca em profundidade

No grafo da figura abaixo, novamente supondo o vértice a como sendo raiz da busca e que as listas de adjacência estão ordenadas em ordem alfabética, a ordem de visitação dos vértices seria: $\{a, b, d, c, e, i, j, g, h, l\}$.



Busca em profundidade

- O algoritmo a seguir implementa uma busca em profundidade em um grafo. Os principais elementos do algoritmo são semelhantes aos do algoritmo da busca em largura.
  As diferenças fundamentais entre os dois algoritmos assim como alguns elementos específicos da DFS são destacados abaixo.
- para que a estratégia de busca tenha o comportamento desejado, deve-se usar uma **pilha** no lugar da fila. Por razões de eficiência, a pilha deve armazenar não apenas uma informação sobre o vértice cuja adjacência deve ser explorada mas também um apontador para o próximo elemento da lista a ser percorrido.
- o vetor `dist` conterá não mais a distância dos vértices alcançáveis a partir da raiz e sim o número de arestas percorridas pelo algoritmo até visitar cada vértice pela primeira vez.

Busca em profundidade

- p é uma variável apontadora de um registro da lista de adjacências, para o qual é assumida a existência de dois campos: `vert`, contendo o rótulo que identifica o vértice, e `prox` que aponta para o próximo registo da lista.
- Excluídas estas pequenas diferenças, as implementações dos algoritmos BFS e DFS são bastante parecidas. Assim, é fácil ver que **a complexidade da busca em profundidade também é $O(|V| + |E|)$**, isto é, linear no tamanho da representação do grafo por listas de adjacências.

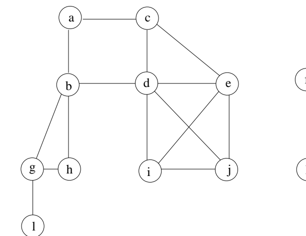
Busca em profundidade: algoritmo iterativo

DFS(G,raiz);
▷ Q é o conjunto de vértices a serem explorados (cinzas)
1. InicializaPilha(Q); ▷ inicializa pilha como sendo vazia
2. dist[raiz] $\leftarrow$ 0; cor[raiz] $\leftarrow$ cinza; pred[raiz] $\leftarrow$ NULO;
3. **Para** todo $v \in V - \{\text{raiz}\}$ **faça**
4. { dist[v] $\leftarrow \infty$; cor[v] $\leftarrow$ branco; pred[v] $\leftarrow$ NULO; }
5. Empilha(Q, raiz, Adj[*raiz*]);

6. **Enquanto** (!PilhaVazia(Q)) **faça**
7. Desempilha(Q, u, p); ▷ tirar u do conjunto Q
8. **Enquanto** ($p \neq$ NULO) **e** (cor[$p \neg$ vert] $\neq$ branco)**,** $p \leftarrow p \neg$ prox;
9. **Se** $p \neq$ NULO **então**
10. Empilha($Q, u, p \neg$ prox); $v \leftarrow p \neg$ vert; dist[v] $\leftarrow$ dist[u] + 1;
11. pred[v] $\leftarrow u$; cor[v] $\leftarrow$ cinza; Empilha(Q, v, Adj[v]);
12. **Se não** cor[u] $\leftarrow$ preto;

13. **Retorne** (dist, pred).

Busca em profundidade: exemplo



Nota: o subgrafo formado por (V, E_{pred}), onde $E_{\text{pred}} = \{(\text{pred}[v], v) : \forall v \in V\}$, é uma árvore.

Busca em profundidade: versão recursiva

- Apresenta-se a seguir uma versão recursiva da busca em profundidade.
- Nesta versão, **todos** vértices do grafo serão visitados. Ou seja, ao final, teremos uma **floresta** de árvores DFS.
- O mesmo pode ser feito para BFS mas, é mais natural nas aplicações de busca em profundidade.
- Uma variável tempo será usada para marcar os instantes onde um vértice é descoberto (d) pela busca e onde sua vizinhança termina de ser explorada (f).
- A exemplo do que ocorre na busca em largura, algumas das variáveis do algoritmo são desnecessárias para efetuar a busca em profundidade pura e simples.

  Contudo, elas são fundamentais para aplicações desta busca e, por isso, são mantidas aqui.

Busca em profundidade: versão recursiva

DFS(G)

```
1.  para cada vértice u ∈ V faça
2.     cor[u] ← branco;      pred[u] ← NULO;
3.  tempo ← 0;
4.  para cada vértice u ∈ V faça
5.     se cor[u] = branco então  DFS-AUX(u)
```

DFS-AUX(u) ▷ u acaba de ser descoberto

```
1.  cor[u] ← cinza;
2.  d[u] ← tempo;      tempo ← tempo + 1;
3.  para v ∈ Adj[u] faça        ▷ explora aresta (u, v)
4.     se cor[v] = branco então
5.        pred[v] ← u;      DFS-AUX(v);
6.  cor[u] ← preto;      ▷ u foi explorado
7.  f[u] ← tempo;      tempo ← tempo + 1;
```


Propriedades das buscas: BFS

Notação: $\delta(s, v)$ é a distância de s a v, ou seja, menor comprimento de um caminho ligando estes dois vértices.

Lema 22.1: (Cormen, 2ed)

Seja $G = (V, E)$ um grafo (direcionado ou não) e s um vértice qualquer de V. Então, para toda aresta $(u, v) \in E$, tem-se que $\delta(s, v) \leq \delta(s, u) + 1$.

Prova: u é alcançável a partir de s ... Se não for ... □

Lema 22.2: (Cormen, 2ed)

Suponha que uma BFS é executada em G tendo s como raiz. Ao término da execução, para cada vértice $v \in V$, o valor de $\mathtt{dist}[v]$ computado pelo algoritmo satisfaz $\mathtt{dist}[v] \geq \delta(s, v)$.

Propriedades das buscas: BFS

Prova do Lema 22.2: indução no número de operações de inserção na fila.

Hipótese indutiva (HI): $\mathtt{dist}[v] \geq \delta(s, v)$ para todo $v \in V$.

Base da indução: inclusão inicial de s na fila ...

Passo da indução: v é um vértice branco descoberto durante a exploração da adjacência do vértice u. Como a HI implica que $\mathtt{dist}[u] \geq \delta(s, u)$, as operações das linhas 10 e 11 e o Lema 22.1 implicam que:

$$\mathtt{dist}[v] = \mathtt{dist}[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v).$$

O valor de $\mathtt{dist}[v]$ nunca mais é alterado pelo algoritmo ... □

Propriedades das buscas: BFS

Para provar que $\texttt{dist}[v] = \delta(s, v)$, é necessário entender melhor como opera a fila Q.

Lema 22.3: (Cormen, 2ed)

Suponha que numa iteração qualquer do algoritmo BFS, os vértices na fila Q sejam dados por $\{v_1, v_2, \dots, v_r\}$, sendo v_1 a cabeça e v_r a cauda da fila. Então, $\texttt{dist}[v_r] \leq \texttt{dist}[v_1] + 1$ e $\texttt{dist}[v_i] \leq \texttt{dist}[v_{i+1}]$, para todo $i = 1, 2, \dots, r-1$.

Prova: indução no número de operações na fila (insere/remove).

Base da indução: a afirmação é evidente quando $Q = \{s\}$.

Passo da indução: dividido em duas partes.
A primeira trata da remoção de v_1, o que torna v_2 a nova *cabeça* de Q ...

Propriedades das buscas: BFS

Prova do Lema 22.3: (continuação)
A segunda trata de inclusão de um novo vértice v, tornando-o a nova *cauda* (v_{r+1}) de Q. Supõe-se que o vértice v foi descoberto na exploração da vizinhança de um vértice u.
Pela HI e pelas operações das linhas 10 e 11 chega-se a:

$$\texttt{dist}[v_{r+1}] = \texttt{dist}[v] = \texttt{dist}[u] + 1 \leq \texttt{dist}[v_1] + 1.$$

A HI também implica que

$$\texttt{dist}[v_r] \leq \texttt{dist}[u] + 1 = \texttt{dist}[v] = \texttt{dist}[v_{r+1}],$$

e, além disso, as demais desigualdades ficam inalteradas. □

Propriedades das buscas: BFS

Corolário 22.4: (Cormen, 2ed)

Suponha que os vértices v_i e v_j são inseridos na fila Q durante a execução do algoritmo BFS <u>nesta ordem</u>. Então, $\texttt{dist}[v_i] \leq \texttt{dist}[v_j]$ a partir do momento em que v_j for inserido em Q.

Prova: Imediato. □

Teorema 22.5: corretude da BFS (Cormen, 2ed)

Durante a execução do algoritmo BFS, todos os vértices $v \in V$ alcançáveis a partir de s são descobertos e, ao término da execução, $\texttt{dist}[v] = \delta(s, v)$ para todo $v \in V$.

Além disso, para todo vértice $v \neq s$ alcançável a partir de s, um dos menores caminhos de s para v é composto por um menor caminho de s para $\texttt{pred}[v]$ seguido da aresta $(\texttt{pred}[v], v)$.

Propriedades das buscas: DFS

- Verifica-se agora algumas propriedades da busca em profundidade. Para tanto, usa-se a versão recursiva do algoritmo e para todo vértice u, denota-se por I_u o intervalo $[d[u], f[u]]$.

Teorema 22.7 (Parentização): (Cormen, 2ed)

Em qualquer busca em profundidade em um grafo $G = (V, E)$ (direcionado ou não), para quaisquer vértices u e v de V, **<u>exatamente</u>** uma das situações abaixo ocorre:

- $I_u \cap I_v = \{\ \}$ e u e v não são descendentes um do outro na floresta construída pela DFS, *<u>ou</u>*
- $I_u \subset I_v$ e o u é descendente de v em uma árvore da floresta construída pela DFS,
- $I_v \subset I_u$ e o v é descendente de u em uma árvore da floresta construída pela DFS.

Propriedades das buscas: DFS

Corolário 22.8 (aninhamento dos intervalos dos descententes): (Cormen, 2ed)

O vértice v é um descendente *próprio* do vértice u em uma árvore da floresta construída pela DFS *se e somente se* $d[u] < d[v] < f[v] < f[u]$.

Uma outra caracterização de descendência é dada pelo **Teorema do Caminho Branco** enunciado a seguir.

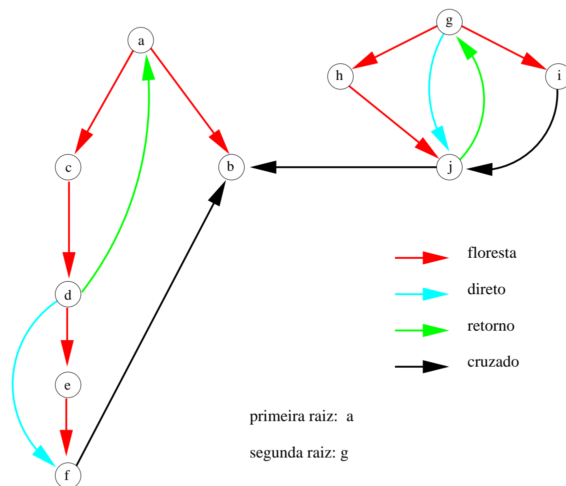
Teorema 22.9: (Cormen, 2ed)

Em uma floresta construída pela DFS, um vértice v é descendente de um vértice u *se e somente se* no *tempo* $d[u]$ em que a busca **descobre** u, o vértice v é alcançável a partir de u por um caminho *inteiramente* composto por vértices de **cor branca**.

Propriedades das buscas: Classificação de arestas

- O algoritmo DFS permite classificar arestas do grafo e esta classificação dá acesso a importantes informações sobre o mesmo (p.ex., a existência de ciclos em grafos direcionados).
- Considere a floresta $G_{\texttt{pred}}$ construída pela DFS. As arestas podem se classificar como sendo:
  - **Arestas da floresta:** aquelas em $G_{\texttt{pred}}$.
  - **Arestas de retorno (back edges):** arestas (u, v) conectando o vértice u a um vértice v que é seu **ancestral** em $G_{\texttt{pred}}$. Em grafos direcionados, os auto-laços são considerados arcos de retorno.
  - **Arestas diretas (forward edges):** arestas (u, v) que não estão em $G_{\texttt{pred}}$ onde v é descendente de u em uma árvore construída pela DFS.
  - **Arestas cruzadas (cross edges):** todas as arestas restantes. Elas podem ligar vértices de uma mesma árvore de $G_{\texttt{pred}}$, desde que nenhuma das extremidades seja ancestral da outra, ou vértices de diferentes árvores.

Classificação de arestas: exemplo



Classificação de arestas: modificações no algoritmo

- O algoritmo DFS pode ser modificado para ir classificando as arestas a medida que elas são encontradas.
- A idéia é que a aresta (u, v) pode ser classificada de acordo co a cor do vértice v no momento que a aresta é "explorada" pela primeira vez:
  1. `branco`: indica que (u, v) é um aresta da floresta (está em $G_{\texttt{pred}}$).
  2. `cinza`: indica que (u, v) é um aresta de retorno.
  3. `preto`: indica que (u, v) é um aresta direta ou uma aresta cruzada.
     Pode-se mostrar (exercício) que ela será direta se $d[u] < d[v]$, caso contrário será cruzada.

Classificação de arestas: modificações no algoritmo

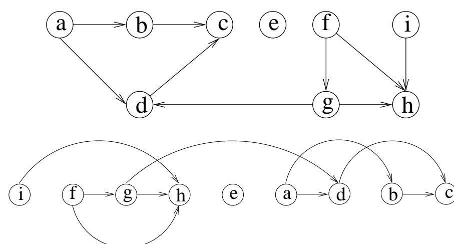
- Para grafos não direcionados, a classificação pode apresentar ambigüidades já que (u, v) e (v, u) são de fato a mesma aresta.
- O algoritmo pode ser ajustado para manter a primeira classificação em que a aresta puder ser categorizada.
- Arestas diretas e cruzadas *nunca* ocorrem em grafos não-direcionados.

Teorema 22.10: (Cormen, 2ed)
Em uma DFS de um grafo **não direcionado** $G = (V, E)$, toda aresta é da floresta ou é de retorno.

Ordenação topológica

Ordenação topológica

- A **ordenação topológica** de um grafo direcionado acíclico (**DAG**) $G = (V, E)$ é uma ordem linear dos vértices tal que, para todo arco (u, v) em E, u *antecede* v na ordem.
- *Todo DAG admite uma ordenação topológica !*
- Graficamente, isto significa que G pode ser desenhado de modo que todos seus vértices fiquem dispostos em uma linha horizontal e os seus arcos fiquem todos orientados *da esquerda para direita*.



Ordenação topológica

- Grafos direcionados acíclicos são muito usados para representar situações onde haja relações de precedência entre eventos. Exemplo: tarefas necessárias para construir uma casa.
- Em muitas aplicações a ordenação topológica é usada como um pré-processamento que conduz a um algoritmo *eficiente* para resolver um problema definido sobre grafos.
- A ordenação topológica pode ser obtida através de uma simples adaptação do algoritmo de busca em profundidade mostrada a seguir.

Ordenação topológica: algoritmo

TOPOLOGICAL-SORT(G)

▷ **Entrada:** grafo direcionado acíclico G.
▷ **Saída:** lista ligada L com ordem topológica dos vértices de G.

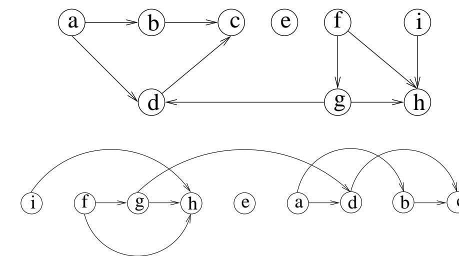
1. IncializaLista(L); $\quad$ `tempo` $\leftarrow 0$;
2. **para** cada vértice $u \in V$ **faça** `cor`$[u] \leftarrow$ `branco`;
3. **para** cada vértice $u \in V$ **faça**
4. $\quad$ **se** `cor`$[u] =$ `branco` **então** DFS-AUX(u)
5. **retorne** L.

DFS-AUX(u) $\quad$ ▷ *u acaba de ser descoberto*

1. `cor`$[u] \leftarrow$ `cinza`; $\quad d[u] \leftarrow$ `tempo`; $\quad$ `tempo` $\leftarrow$ `tempo` $+ 1$;
2. **para** $v \in$ `Adj`$[u]$ **faça** $\quad$ ▷ explora aresta (u, v)
3. $\quad$ **se** `cor`$[v] =$ `branco` **então** DFS-AUX(v);
4. `cor`$[u] \leftarrow$ `preto`; $\quad$ ▷ *u foi explorado*
5. $f[u] \leftarrow$ `tempo`; $\quad$ `tempo` $\leftarrow$ `tempo` $+ 1$;
6. InsereInicioLista(L, u);

Ordenação topológica

- Vê-se que os vértices são armazenados na lista ligada L na *ordem inversa* (decrescente) dos valores de $f[.]$ (a inserção se dá sempre no inicio da lista).
- A *complexidade* deste algoritmo é obviamente igual àquela da DFS, ou seja, $O(|V| + |E|)$.
- **Exemplo:** raízes a, e, f e i.



Ordenação topológica: corretude

Lema 22.11 (Cormen 2ed):
Um grafo direcionado G é acíclico *se e somente se* uma busca em profundidade em G não classifica nenhum arco como sendo de retorno.

Prova: ($\Longrightarrow$) por contradição. Supor que (u, v) é um arco de retorno. O único caminho em $G_{\mathtt{pred}}$ de v para u juntamente com o arco(u, v) forma um ciclo.

($\Longleftarrow$) por contradição. Supor que existe um ciclo C em G. Seja v o primeiro vértice de C descoberto pela busca e (u, v) o arco precedendo v em C. No instante $d[v]$, os arcos de $C - \{(u, v)\}$ formam um **caminho branco** ligando v a u. Logo, u será um descendente de v e, então, (u, v) é um arco de retorno. $\square$

Ordenação topológica: corretude

Teorema 22.12 (Cormen 2ed):
O algoritmo `TOPOLOGICAL-SORT`(G) produz uma ordenação topológica correta de um grafo direcionado acíclico.

Prova: suponha que DFS tenha sido executado sobre o DAG G. Seja (u, v) um arco de G e considere o instante onde este arco está sendo explorado. A cor de v não pode ser `cinza`, pois (u, v) seria arco de retorno, contrariando o Lema 22.11. Se `cor`$[v] =$ `branco`, v é descendente de u e, assim, $f[v] < f[u]$. Se `cor`$[v] =$ `preto`, $f[v]$ já foi fixado. Como (u, v) está sendo explorado, `cor`$[u] =$ `cinza` e $f[u]$ ainda não foi fixado. Portanto, $f[v] < f[u]$.

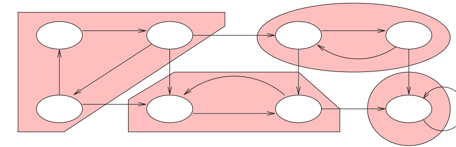
Conclusão: quando (u, v) está em G, $f[u] > f[v]$, ou seja, u aparecerá antes de v na ordenação dada pelo algoritmo, como requerido. $\square$

Componentes fortemente conexas

Componentes Fortemente Conexas (CFC)

Definição:

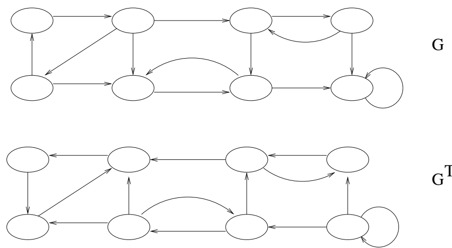
Uma CFC H de um grafo orientado $G = (V, E)$ é um subconjunto de vértices tal que, para todo par de vértices u e v de H, existe um caminho de u para v e vice-versa. Além disso, H é **maximal** com respeito à inclusão de vértices.



Componentes Fortemente Conexas (CFC)

Definição:

Dado um grafo direcionado $G = (V, E)$, o **grafo transposto** de G é o grafo $G^T = (V^T, E^T)$ onde $V^T = V$ e $E^T = \{(u, v) \in V^T \times V^T : (v, u) \in E\}$. *Ou seja, G^T é o grafo obtido de G invertendo-se a orientação de seus arcos*.



Componentes Fortemente Conexas (CFC)

Propriedades do grafo transposto:

- Se G é dado pelas suas listas de adjacências, as listas de adjacências de G^T podem ser obtidas em $\Theta(|V| + |E|)$.
- G e G^T têm as mesmas componentes fortemente conexas.

Algoritmo CFC:

1. Execute DFS(G) para obter o valor de $f[u]$ para todo $u \in V$.
2. Construir G^T.
3. Execute DFS(G^T), escolhendo como raiz da próxima árvore DFS o vértice não visitado (branco) com maior valor $f[\,]$.
4. Retornar os conjuntos de vértices de cada uma das árvores DFS como sendo aqueles que formam as diferentes CFCs de G.

Componentes Fortemente Conexas: corretude

Lema 22.13 (Cormen 2ed):

Seja C e C' duas CFCs distintas do grafo direcionado G. Sejam os vértices u e v em C e u' e v' em C'. Suponha que existe um caminho $u \rightsquigarrow u'$ em G. Então não pode existir um caminho $v \rightsquigarrow v'$ em G.

Definição:

Para todo subconjunto U de vértices define-se:

$$f(U) := \max_{u \in U}\{f[u]\} \quad \text{e} \quad d(U) := \min_{u \in U}\{d[u]\}$$

Lema 22.14 (Cormen 2ed):

Seja C e C' duas CFCs distintas do grafo direcionado $G = (V, E)$. Suponha que o arco (u, v) está em E, sendo que $u \in C$ e $v \in C'$. Então $f(C) > f(C')$.

Componentes Fortemente Conexas: corretude

Corolário 22.15 (Cormen 2ed):

Seja C e C' duas CFCs distintas do grafo direcionado $G = (V, E)$. Suponha que o arco (u, v) está em E^T **(grafo transposto)**, sendo que $u \in C$ e $v \in C'$. Então $f(C) < f(C')$.

Teorema 22.16 (Cormen 2ed):

O algoritmo CFC calcula corretamente as componentes fortemente conexas de um grafo direcionado G.

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Árvore Geradora Mínima

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Árvore Geradora Mínima: definição do problema

- Suponha que queiramos construir estradas para interligar n cidades, sendo que, a cada estrada entre duas cidades i e j que pode ser construída, há um custo de construção associado.
- *Como determinar eficientemente quais estradas devem ser construídas de forma a minimizar o custo total de interligação das cidades ?*
- Este problema pode ser modelado por um problema em grafos não orientados ponderados onde os vértices representam as cidades, as arestas representam as estradas que podem ser construídas e o peso de uma aresta representa o custo de construção da estrada.

Cid Carvalho de Souza e Cândida Nunes da Silva MC448 — Análise de Algoritmos I

Árvore Geradora Mínima: definição do problema

Nessa modelagem, o problema que queremos resolver é encontrar um **subgrafo gerador** (que contém todos os vértices do grafo original), **conexo** (para garantir a interligação de todas as cidades) e cuja soma dos custos de suas arestas seja a menor possível.

Mais formalmente, definimos o problema da seguinte forma:

Problema da Árvore Geradora Mínima:

Dado um grafo não orientado ponderado $G = (V, E)$, onde $w : E \rightarrow \mathbb{R}^+$ define os custos das arestas, encontrar um subgrafo gerador conexo T de G tal que, para todo subgrafo gerador conexo T' de G

$$\sum_{e \in T} w_e \leq \sum_{e \in T'} w_e.$$

Árvore Geradora Mínima

- Claramente, o problema só tem solução se G é conexo.
- **Portanto, a partir de agora, vamos supor que G é conexo**.
- Além disso, não é difícil ver que a solução para esse problema será sempre uma árvore: basta notar que T não conterá ciclos pois, caso contrário, poderíamos obter um outro subgrafo T', ainda conexo e com custo menor ou igual ao de T, removendo uma aresta do ciclo.
- Portanto, dizemos que este problema de otimização é o problema de encontrar a *Árvore Geradora Mínima* (AGM) em G.
- *Como projetamos um algoritmo para resolver esse problema ?*

Algoritmo para AGM

- Uma primeira abordagem exaustiva para solucionar o problema poderia ser enumerar todas as árvores geradoras do grafo, computar seus custos e retornar uma árvore de custo mínimo.
- No entanto, esse não é um bom algoritmo pois um grafo completo de n vértices possui um número **exponencial** (n^{n-2}) de árvores geradoras.
- Para obter um algoritmo eficiente (polinomial !) devemos então procurar alguma propriedade do problema que nos permita restringir o espaço de possíveis soluções a ser analisado.

Algoritmo genérico para AGMs

- Os dois algoritmos que serão vistos usam uma **estratégia gulosa**, diferindo apenas no modo em que esta é aplicada.
- A estratégia gulosa é resumida no *algoritmo genérico* mostrado a seguir, onde a AGM é construída aresta a aresta.
- A cada iteração do algoritmo é mantido um conjunto A de arestas que satisfaz à seguinte **invariante**:

> Antes de cada iteração, A é um subconjunto de arestas de *uma* AGM.

- Em cada iteração, determina-se ainda uma aresta (u, v) tal que $A \cup \{(u, v)\}$ que também satisfaz à invariante, i.e., está contido em alguma AGM. Tal aresta é dita ser **segura**.

Algoritmo genérico para AGMs

AGM-GENERICO(G, w)

1. $A \leftarrow \emptyset$;
2. **enquanto** A não forma uma árvore geradora **faça**
3. encontre uma aresta **segura** (u, v);
4. $A \leftarrow A \cup \{(u, v)\}$;
5. **retorne** A.

- A parte mais *engenhosa* do algoritmo é encontrar a aresta **segura** na linha 3.
- Esta aresta deve existir pois, por hipótese, no laço das linhas 2–4, A é um subconjunto **próprio** de uma árvore geradora T. Logo existe uma aresta segura $(u, v) \in T - A$.

Árvores geradoras: reconhecendo arestas seguras

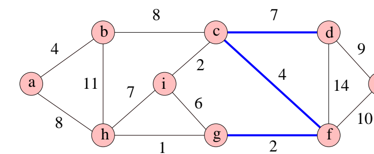
Notação:

Considere um grafo não orientado $G = (V, E)$ e tome $S \subseteq V$. O conjunto $V - S$ é denotado por $\overline{S}$.

Definição:

O *corte de arestas* de S, denotado por $\delta(S)$, é o conjunto de arestas de G com um extremo em S e outro em $\overline{S}$.

Exemplo: $S = \{a, b, c, g, h, i\}$.



Árvores geradoras: reconhecendo arestas seguras

Diz-se que um corte $\delta(S)$ **respeita** um conjunto de arestas A se $A \cap \delta(S)$ é vazio.

Diz-se que (u, v) é uma **aresta leve** de um corte $\delta(S)$ se $w_{uv} := \min_{(x,y)\in\delta(S)}\{w_{xy}\}$, i.e., (u, v) é a aresta de menor peso no corte $\delta(S)$.

Teorema 23.1: (Cormen 2ed)

Seja $G = (V, E)$ um grafo conexo não orientado ponderado nas arestas por uma função $w : E \rightarrow \mathbb{R}$. Seja A um subconjunto de E que está contido em alguma AGM de G, seja ainda $\delta(S)$ um corte de G que *respeita* A e (u, v) uma *aresta leve* de $\delta(S)$. Então, (u, v) é uma *aresta segura* para A.

Árvores geradoras: reconhecendo arestas seguras

O Teorema anterior deixa claro o funcionamento do algoritmo genérico para AGM. À medida que o algoritmo avança, o grafo induzido por A é acíclico (pois pertence à uma árvore). Ou seja $G_A = (V, A)$ é uma floresta, sendo que algumas árvores podem ter um único vértice. Cada aresta segura (u, v) conecta duas componentes distintas de G_A.

Inicialmente a floresta tem $|V|$ árvores formadas por vértices isolados. Cada iteração reduz o número de componentes da floresta de uma unidade. Portanto, após a inclusão de $|V| - 1$ arestas seguras, o algoritmo termina com uma única árvore.

Árvores geradoras: reconhecendo arestas seguras

Corolário 23.2: (Cormen 2ed)

Seja $G = (V, E)$ um grafo conexo não orientado ponderado nas arestas por uma função $w : E \rightarrow \mathbb{R}$. Seja A um subconjunto de E que está contido em alguma AGM de G, e seja $C = (V_C, E_C)$ uma componente conexa (árvore) da floresta $G_A = (V, A)$. Se (u, v) é uma *aresta leve* de $\delta(C)$, então (u, v) é **segura** para A.

Os algoritmos de Prim e de Kruskal

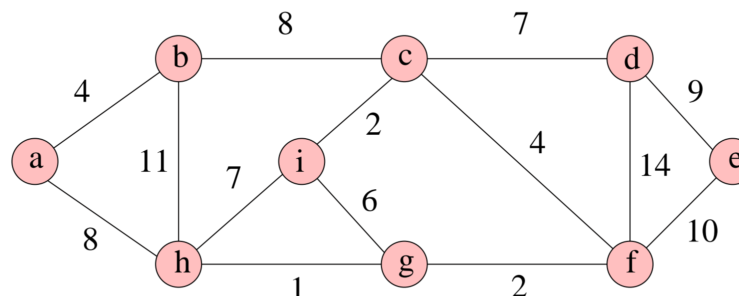
- Os algoritmos de Prim e de Kruskal especializam o algoritmo genérico para AGM visto anteriormente, fazendo uso Corolário 23.3.
- No *algoritmo de Prim*, o conjunto de arestas A é sempre uma **árvore**. A *aresta segura* adicionada a A é sempre uma *aresta leve* do corte $\delta(C)$, onde C é o conjunto de vértices que são extremidades de arestas em A.
- No *algoritmo de Kruskal*, o conjunto de arestas A é uma **floresta**. A *aresta segura* adicionada a A é sempre a aresta de menor peso dentre todas as arestas ligando dois vértices de componentes distintas de G_A.

O algoritmo de Prim

PRIM(G, w, r)

```
▷ r é um vértice escolhido para ser raiz da AGM
  Q é o conjunto de vértices a incluir na árvore
  dist[x]: peso da aresta mais leve ligando x a componente de r
1.  dist[r] ← 0;        Q ← V;
2.  para todo v ∈ V − {r} faça dist[v] ← ∞;
3.  pred[r] ← nulo;
4.  A ← {};        ▷ inicializa o conjunto de arestas da AGM
5.  W ← 0;         ▷ inicializa o peso da AGM
6.  enquanto (Q não for vazio) faça
7.     Remover de Q o vértice u com o menor valor em dist;
8.     W ← W + dist[u];
9.     se pred[u] ≠ nulo, A ← A ∪ {(pred[u], u)};
       ▷ atualiza o valor de dist[.] para vértices adjacentes a u
10.    para todo v ∈ Adj[u] faça
11.       se (v ∈ Q) e (dist[v] > w[u, v]) então
12           { dist[v] ← w[u, v];    pred[v] ← u; }
13. retorne (A, W).
```


O algoritmo de Prim: exemplo



O algoritmo de Prim: corretude

O algoritmo de Prim mantém a seguinte invariante, a qual é válida antes de cada execução do laço das linhas 6 a 12:

1. $A = \{(v, \texttt{pred}[v]) : v \in V - \{r\} - Q\}$.
2. Os vértices já colocados na AGM são aqueles em $V - Q$.
3. Para todos os vértices em Q, se $\texttt{pred}[v] \neq$ nulo, então $\texttt{dist}[v] < \infty$ e $\texttt{dist}[v]$ é o valor da aresta leve $(v, \texttt{pred}[v])$ que conecta v a algum vértice já pertencente a AGM.

O algoritmo de Prim: complexidade

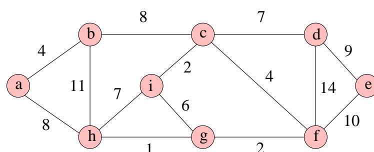
- A complexidade depende de como o conjunto Q é *implementado*.
- Se Q for implementado como um vetor simples (Q poderia estar representado pelo próprio vetor `dist`), a complexidade do algoritmo será $O(|V|^2 + |E|) \equiv O(|V|^2)$.
- Se Q for implementado como um heap (Q poderia estar representado pelo próprio vetor `dist`), a complexidade do algoritmo será $O((|V| + |E|) \log |V|) \equiv O(|E| \log |V|)$.
- Portanto, para grafos *densos* ($|E| \in O(|V|^2)$), a melhor alternativa é implementar Q como um vetor simples, enquanto que para grafos *esparsos* ($|E| \in O(|V|)$), deve-se optar pela implementação de Q como uma fila de prioridades.
- Note que, com esta última opção, o algoritmo de Prim assemelha-se muito a uma busca em largura. A diferença entre os algoritmos fica praticamente restrita à troca de uma fila simples por uma fila de prioridades !

O algoritmo de Kruskal para AGM

Principais passos do algoritmo de Kruskal:

1. Ordenar as arestas em ordem **não** decrescente de peso e inicializar A como estando vazio.
2. Seja (u, v) a próxima aresta na ordem **não** decrescente de peso. Se ela formar um ciclo com as arestas de A, então ela é **rejeitada**. Caso contrário, ela é **aceita** e faz-se $A = A \cup \{(u, v)\}$.
3. Repetir o passo anterior até que $|V| - 1$ arestas tenham sido **aceitas**.

Exemplo:



O algoritmo de Kruskal para AGM

- O algoritmo de Kruskal é baseado diretamente no *algoritmo genérico* discutido anteriormente.
- Note que quando uma aresta (u, v) é aceita, as componentes C_1 e C_2 contendo u e v, respectivamente, são *distintas*. É fácil ver que (u, v) é uma *aresta leve* para $\delta(C_1)$ (ou $\delta(C_2)$). Portanto, (u, v) satisfaz às condições do Corolário 23.3, o que garante a corretude do algoritmo.
- Em uma iteração qualquer, a solução parcial corrente é a floresta composta por todos vértices do grafo e as arestas em A.
- Para implementar o algoritmo de Kruskal, precisamos encontrar uma maneira eficiente de *manter as componentes* da floresta $G_A = (V, A)$.

O algoritmo de Kruskal para AGM

- Em particular, as componentes devem ser armazenadas de modo que duas operações sejam feitas muito rapidamente:
  - dado um vértice u, **encontrar** a componente contendo u;
  - dados dois vértices u e v em componentes distintas C e C', **unir** C e C' em uma única componente.
- para prosseguir com esta discussão, vamos apresentar um pseudo-código do algoritmo de Kruskal.
- Neste pseudo-código, denota-se por $a[u]$ a componente (árvore) de $G_A = (V, A)$ que contém o vértice u na iteração corrente.

Algoritmo de Kruskal: pseudo-código

KRUSKAL(G, w)

1. $W \leftarrow 0$; $A \leftarrow \emptyset$; ▷ inicializações
   ▷ Iniciar G_A com $|V|$ árvores com um vértice cada
2. **para** todo $v \in V$ **faça** $a[v] \leftarrow \{v\}$; ▷ $a[v]$ é identificado com v
   ▷ lista de arestas em ordem não decrescente de peso
3. $L \leftarrow$ **ordene**(E, w);
4. $k \leftarrow 0$; ▷ conta arestas **aceitas**
5. **enquanto** $k \neq |V| - 1$ **faça**
6. **remove**$(L, (u, v))$; ▷ tomar primeira aresta em L
   ▷ acha componentes de u e v
7. $a[u] \leftarrow$ **encontrar**(u); $a[v] \leftarrow$ **encontrar**(v);
8. **se** $a[u] \neq a[v]$ **então** ▷ aceita (u, v) se não forma ciclo com A
9. $A \leftarrow A \cup \{(u, v)\}$;
10. $W \leftarrow W + w(u, v)$;
11. $k \leftarrow k + 1$;
12. **unir**$(a[u], a[v])$; ▷ unir componentes
13. **retorne** (A, W).

Algoritmo de Kruskal: complexidade

- Supor que as componentes de G_A são mantidas de modo que as operações de **encontrar** na linha 7 e **unir** na linha 12 sejam feitas com complexidade $O(f(|V|))$ e $O(g(|V|))$, respectivamente.
- A ordenação da linha 3 tem complexidade $O(|E| \log |E|)$ e domina a complexidade das demais operações das inicializações das linhas de 1 a 4.
- O laço das linhas 5–12 será executado $O(|E|)$ no pior caso. Logo, a complexidade total das linhas 6 e 7 será $O(|E|.f(|V|)$.
- As linhas de 9 a 12 serão executados $|V| - 1$ vezes no total (por quê ?). Assim, a complexidade total de execução destas linhas será $O(|V|.g(|V|)$.
- A complexidade do algoritmo de Kruskal será então

$$O(|E| \log |E| + |E|.f(|V|) + |V|.g(|V|))$$

Fica claro que necessitamos de uma estrutura de dados que permita *manipular* eficientemente as componentes de G_A.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

- Note que os conjuntos de vértices das componentes de G_A formam uma *coleção de conjuntos disjuntos* de V. Sobre estes conjuntos é executada uma seqüência de operações **encontrar** e **unir**.
- A necessidade de representação e manipulação eficiente de coleção de conjuntos disjuntos sob estas operações ocorre em áreas diversas como construção de compiladores e problemas combinatórios, como é o caso da AGM.
- Estruturas de dados simples como vetores contendo informação sobre qual a componente contendo cada elemento (vértice) realizam a operação **encontrar** em $O(1)$ mas, no pior caso, consomem um tempo $O(|V|)$ para realizar uma operação de **união**.
- A alternativa é o uso de estruturas ligadas do tipo **árvore**

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

Cuidado ! Não confunda a estrutura de dados árvore que estaremos usando para armazenar as componentes de G_A com as árvores da floresta G_A.

Ou seja, nesta estrutura os registros estão ligados numa estrutura tipo árvore **cujos apontadores ligando registros de vértices distintos não necessariamente correspondem a uma aresta de** A (pode ser inclusive que nem exista uma aresta com estas mesmas extremidades no grafo).

Para evitar confusão o termo *componente* será usado na discussão que se segue para designar vértices em uma mesma árvore de G_A. O termo *árvore* denotará uma parte da estrutura de dados que representa uma componente de G_A.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

Na discussão que se segue, supomos que, no início, seja criado um **registro** para cada vértice u de V e que o endereço deste registro esteja armazenado em alguma variável de modo que possa ser acessado a qualquer momento do algoritmo.

Além disso, cada registro r da estrutura terá dois campos obrigatórios (`M`) e um terceiro campo opcional (`O`):

- `rot (M)`: contendo o rótulo do vértice que o originou;
- `prx (M)`: apontador ligando registros de uma mesma árvore;
- `num (O)`: número de registros da estrutura de dados tais que, seguindo o campo `prx` até que este seja nulo, chega-se ao registro r.

Algoritmo de Kruskal: complexidade e conjuntos disjuntos

encontrar(u):

▷ retorna o apontador para o registro que representa a componente onde se encontra o vértice u

1. $p \leftarrow$ endereço do registro correspondente a u;
2. **enquanto** (pˆ.prx $\neq$ nulo) **faça**
3. $\quad p \leftarrow p$ˆ.prx;
4. **retorne** p.

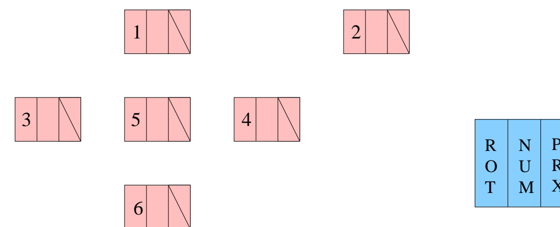
unir(p,q): **versão inicial**

▷ **Entrada**: dois apontadores para registros correspondentes a componentes distintas de G_A

▷ **Saída**: apontador para o registro correspondente ao primeiro parâmetro da entrada

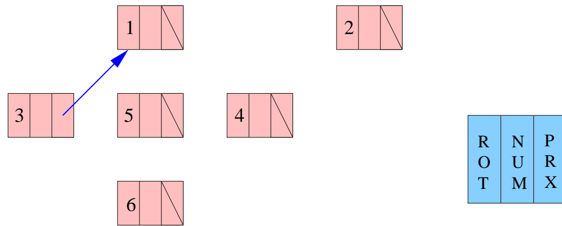
1. qˆ.prx $\leftarrow p$
2. **retorne** p.

Conjuntos disjuntos: exemplo



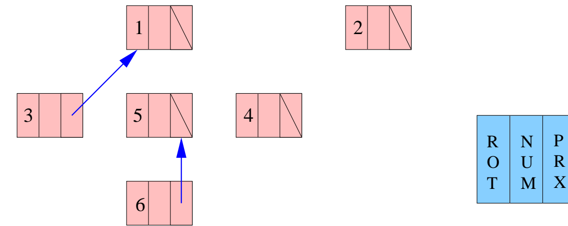
unir(a[1], a[3])

Conjuntos disjuntos: exemplo



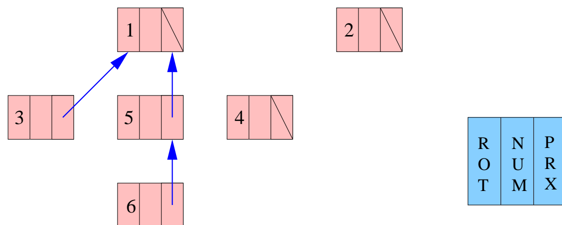
unir$(a[5], a[6])$

Conjuntos disjuntos: exemplo



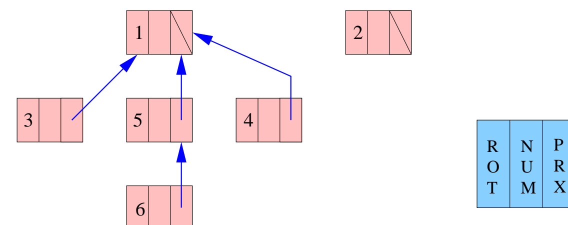
unir$(a[1], a[6])$

Conjuntos disjuntos: exemplo



unir$(a[5], a[4])$

Conjuntos disjuntos: exemplo



Conjuntos disjuntos: complexidade das operações

Complexidades:

Denote-se por T_x a árvore contendo o registro do vértice x e h_x e n_x respectivamente sua altura e seu número de elementos. No pior caso teríamos as complexidades $O(h_u)$ para **encontrar**(u) e $O(1)$ para **unir**(p,q).

Dificuldade:

Uma das árvores pode se transformar em um longo *caminho*. Neste caso, uma operação envolvendo esta árvore teria uma complexidade de pior caso muito alta ($O(|V|)$).

Alternativa:

Construir *árvores balanceadas* colocando no campo `num` do registro r, o número de registros na sub-árvore "abaixo" de r.

Conjuntos disjuntos: complexidade das operações

unir(p,q): **versão melhorada**

▷ **Entrada**: dois apontadores para registros correspondentes a componentes distintas de G_A

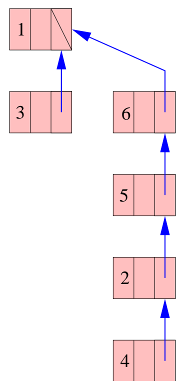
▷ **Saída**: apontador para o registro com maior valor no campo `num`

1. **se** $q\hat{}.\texttt{num} \leq p\hat{}.\texttt{num}$ **então**
2. $\quad q\hat{}.\texttt{prx} \leftarrow p$;
3. $\quad p\hat{}.\texttt{num} \leftarrow p\hat{}.\texttt{num} + q\hat{}.\texttt{num}$;
4. $\quad$ **retorne** p.
5. **senão**
6. $\quad p\hat{}.\texttt{prx} \leftarrow q$;
7. $\quad q\hat{}.\texttt{num} \leftarrow q\hat{}.\texttt{num} + p\hat{}.\texttt{num}$;
8. $\quad$ **retorne** q.

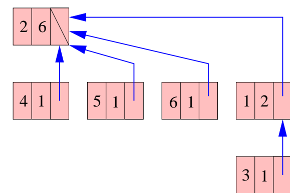
Conjuntos disjuntos: complexidade das operações

Seqüência de operações: unir($a[1], a[3]$), unir($a[2], a[4]$), unir($a[5], a[2]$), unir($a[6], a[5]$) e unir($a[1], a[6]$).

união não-balanceda:



união balanceda:



Conjuntos disjuntos: complexidade das operações

Teorema:

Considere uma seqüência $s_0, s_1, \ldots, s_m$ de operações sobre uma estrutura de dados representando conjuntos disjuntos. A operação s_0 simplesmente cria os registros representando os conjuntos unitários formados por cada um dos elementos. Em $s_1, \ldots, s_m$ todas as operações são do tipo **encontrar** ou **unir**, sendo esta última feita de modo *balanceada*.

Seja x um registro qualquer da estrutura e T_x a árvore que armazena x. Após a operação s_m, T_x é uma **árvore balanceada**.

Conjuntos disjuntos: complexidade das operações

Prova: seja n_x o número de registros em T_x e h_x a sua altura. Deve se mostrar que $2^{h_x-1} \leq n_x$ e, consequentemente, fica provado que $h_x \in O(\log n_x)$.

A prova é feita por indução no número de operações. Na base, o resultado é trivialmente verdadeiro para s_0 já que toda árvore tem um único elemento.

Suponha que o resultado é verdadeiro até o término de s_{m-1}. Deve-se analisar dois casos: *(i)* s_m é uma operação **encontrar**(u) e *(ii)* s_m é uma operação **unir**(v, w).

O caso *(i)* é trivial pois nenhuma árvore é alterada. No caso *(ii)*, sem perda de generalidade, vamos supor que $n_v \leq n_w$.

Conjuntos disjuntos: complexidade das operações

Prova (cont.):
Sejam T_v e T_w as árvores contendo os registros dos vértices v e w, respectivamente. Seja T' a árvore resultante da operação **unir**(v, w) com altura e número de registros dados por h' e n', respectivamente. Note que $n' = n_v + n_w$ e que deve ser provado que $2^{h'-1} \leq n'$.

Claramente $h' = \max\{1 + h_v, h_w\}$. Tem-se que

$$\begin{aligned} 2^{h'-1} &= 2^{\max\{1+h_v,h_w\}-1} = \max\{2^{h_v-1+1}, 2^{h_w-1}\} \\ &\leq \max\{2n_v, n_w\} \qquad \text{(pela H.I.)}. \end{aligned}$$

Como $n' = n_v + n_w$ e $n_v \leq n_w$, então $2n_v \leq n'$ e $n_w \leq n'$.
Logo, $2^{h'-1} \leq n'$. □

Algoritmo de Kruskal: complexidade

- Vimos anteriormente que se as operações de **encontrar** e **unir** fossem feitas com complexidades $O(f(|V|))$ e $O(g(|V|))$, respectivamente, a complexidade do algoritmo de Kruskal seria dada por $O(|E| \log |E| + |E|.f(|V|) + |V|.g(|V|))$.
- O resultado do Teorema garante que usando a **união por tamanho** as árvore de registros na estrutura de dados tem altura limitada a $O(\log |V|)$.
- Portanto, neste caso, a complexidade do algoritmo de Kruskal é dada por $O(|E| \log |E| + |E| \log |V| + |V|)$ ou, como estamos supondo que o grafo é conexo ($|E| \in \Omega(|V|)$), a complexidade é $O(|E| \log |E|) = O(|E| \log |V|)$.
- Melhorias na complexidade ainda podem ser alcançadas usando a técnica de **compressão de caminhos** ao se executar uma operação **encontrar**.

Algoritmo de Kruskal: compressão de caminhos

Na técnica de compressão de caminhos, todo registro visitado durante uma operação **encontrar** que termina em um registro q tem seu campo `prx` alterado de modo a apontar para q.

encontrar(u): adaptada para compressão de caminhos

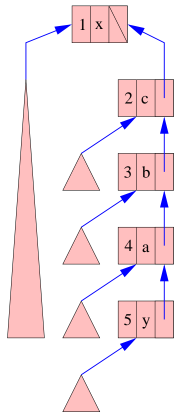
1. $p \leftarrow$ endereço do registro correspondente a u;
2. $q \leftarrow p$; ▷ guardará o último registro da busca
3. **enquanto** q^.prx $\neq$ nulo **faça** $q \leftarrow q$^.prx;
   ▷ refaz a busca fazendo todos registros apontarem para q
4. **enquanto** p^.prx $\neq q$ **faça**
5. $\quad r \leftarrow p$^.prx;
6. $\quad p$^.prx $\leftarrow q$;
7. $\quad p \leftarrow r$;
8. **retorne** q.

Nota: no algoritmo acima, a informação do campo `num` só é correta para o registro "raiz" da árvore.

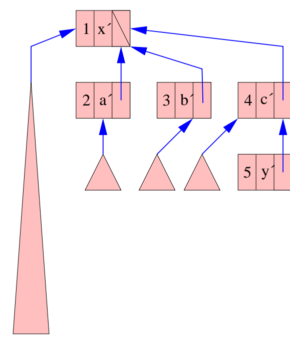
Compressão de caminhos: exemplo

encontrar(4):

Antes:



Depois:



Ajustando o valor de `num`: $x' = x$, $a' = a - y$, $b' = b - a$, $c' = c - b$, $y' = y$

Compressão de caminhos: complexidade

- Pode ser provado que, usando compressão de caminhos, a complexidade de se realizar $|V| - 1$ operações **unir** e $|E|$ operações **encontrar** tem complexidade dada por $O(|V| + |E|\alpha(|V| + |E|, |V|))$.
- Para compreender este resultado, é necessário conhecer a função de Ackerman A e o seu inverso α definidas por:

$$A(p, q) = \begin{cases} 2^q & p = 1,\ q \geq 1 \\ A(p-1, 2) & p \geq 2,\ q = 1 \\ A(p-1, A(p, q-1)) & p \geq 2,\ q \geq 2 \end{cases}$$

$$\alpha(m, n) = \min\{p \geq 1 : A(p, \lfloor m/n \rfloor) > \log n\}, m \geq n.$$

- É fácil ver que a função A tem crescimento muito rápido. Por exemplo:

$$A(2, 1) = A(1, 2) = 2^2$$

$$A(2, x) = A(1, A(2, x-1)) = 2^{A(2,x-1)} = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{\text{(x+1) vezes}}$$

Compressão de caminhos: complexidade

- A função α satisfaz $\alpha(m, n) \leq 4$ para *todos efeitos práticos*.
- Ou seja, a complexidade $O(|V| + |E|\alpha(|V| + |E|, |V|))$, embora não seja linear, se comporta como tal para valores práticos de $|V|$ e $|E|$.
- Note que $A(4, 1) = A(2, 16)$, ou seja, 17 potências sucessivas de 2. Então, se $m \approx n$, na expressão $\alpha(m, n)$, pode-se aproximar o valor de p para 4 em qualquer aplicação prática, já que

$$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{\text{17 vezes}} > \log n,$$

para qualquer valor razoável de n.

Caminhos Mínimos

Caminhos Mínimos a partir de um vértice
Definição do Problema

Dados um grafo conexo ponderado $G = (V, E)$, orientado ou não, onde $d : E \longrightarrow \mathbb{R}^+$ define as distâncias entre os extremos das arestas, e um vértice r de G, queremos encontrar, para todo vértice u de G um caminho C de distância mínima de r a u, ou seja, o caminho C deve ser tal que, para todo caminho C' de r a u em G:

$$\sum_{e \in C} d_e \leq \sum_{e \in C'} d_e.$$

Algoritmo de Dijkstra: projeto por indução

- **Base:** Para $k = 1$, o próprio vértice origem r é o vértice mais próximo, com distância nula.
  Para $k = 2$, o segundo vértice mais próximo de r é um vértice w adjacente a r por uma aresta $e = (r, w)$ com d_e mínimo, que é o valor da distância mínima.
- **Hipótese:** Dado um grafo G não orientado ponderado e conexo, sabemos encontrar os k vértices mais próximos do vértice origem r, $1 \leq k \leq |V| - 1$, e as respectivas distâncias mínimas.
- **Passo:** Por hipótese de indução, sabemos encontrar os k vértices mais próximos do vértice origem r e as distâncias mínimas. Seja S o conjunto dos k vértices mais próximos de r e $\texttt{dist}[u]$ a distância mínima de r a u, para todo $u \in S$.

Algoritmo de Dijkstra: projeto por indução

- **Passo (cont.):** O $(k+1)$-ésimo vértice mais próximo de r é um vértice de $\overline{S}$ e o caminho mínimo de r a ele necessariamente passa por uma aresta de $\delta(S)$.
  Defina, para todo vértice w de $\overline{S}$ que é extremo de alguma aresta $e = (u, w)$ de $\delta(S)$,

  $$d(w) = \min_{e=(u,w) \in \delta(S)} \{\texttt{dist}[u] + d_e\}.$$

  O vértice w com $d(w)$ mínimo será o $(k+1)$-ésimo vértice mais próximo de r, pois qualquer caminho de r a um vértice $y \in \overline{S}$, $y \neq w$ necessariamente passaria por um vértice x extremo de alguma aresta de $\delta(S)$.
  Como $d(w) \leq d(x)$ e os pesos das arestas são não negativos tal caminho de r a y certamente não terá comprimento menor que $d(w)$. $\square$

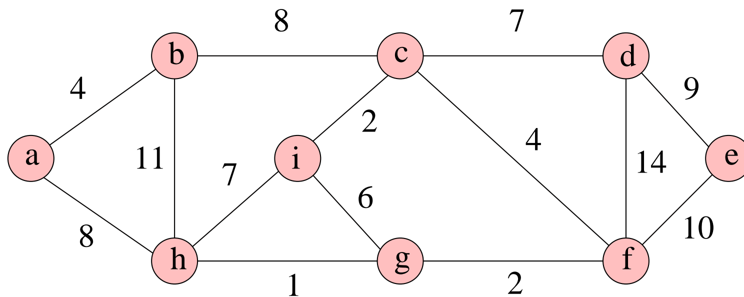
Algoritmo de Dijkstra

Da demonstração indutiva obtemos um algoritmo para determinar não apenas as distâncias mínimas de r aos demais vértices, mas também os caminhos mínimos:

Principais passos do algoritmo

1. Inicialmente, tome $S = \{r\}$.
2. Para k de 2 a $|V|$ repita
   - 2.1. Calcule $d(w)$ para todos os vértices $w \in \overline{S}$ que são extremos de arestas de $\delta(S)$.
   - 2.2. Tome w com $d(w)$ mínimo, onde $d(w) = \texttt{dist}[u] + d_e$ para alguma aresta $e = (u, w)$ de $\delta(S)$.
   - 2.3. Acrescente w a S, tomando como caminho mínimo de r a w a concatenação do caminho mínimo de r a u e da aresta e, com comprimento total $d(w)$.

O algoritmo de Dijkstra: exemplo



Árvore de Caminhos Mínimos

- É interessante notar que a união dos caminhos mínimos de r aos demais vértices é uma **árvore geradora com raiz** em r.
- Esta árvore é chamada de árvore de caminhos mínimos para r (ACM(r)). Ela é formada exatamente pelas arestas que conectam o vértice a ser inserido em S no passo 2.3 a um vértice de S.
- O algoritmo de Dijsktra é muito parecido com o algoritmo de Prim visto anteriormente e, da mesma forma, sua complexidade depende fortemente da implementação.
- Dada a semelhança entre os dois algoritmos, podemos usar as mesmas estruturas de dados analisadas no algoritmo de Prim para o algoritmo de Dijkstra, obtendo implementações de mesma complexidade.

Algoritmo de Dijkstra

DIJKSTRA(G, d, r)

▷ r é o vértice origem
$\overline{S}$ é o conjunto de vértices com caminho mais curto ainda a descobrir
dist[x]: comprimento do menor caminho de r até x, dado S

1. dist[r] $\leftarrow 0$; $\overline{S} \leftarrow V$;
2. **para** todo $v \in V - \{r\}$ **faça** dist[v] $\leftarrow \infty$;
3. pred[r] $\leftarrow$ nulo;
4. $A \leftarrow \{\}$; ▷ inicializa o conjunto de arestas da ACM(r)
5. **enquanto** ($\overline{S}$ não for vazio) **faça**
6. Remover de $\overline{S}$ o vértice u com o menor valor em dist;
7. **se** pred[u] $\neq$ nulo, $A \leftarrow A \cup \{(\text{pred}[u], u)\}$;
   ▷ atualiza o valor de dist[.] para vértices adjacentes a u
8. **para** todo $v \in$ Adj[u] **faça**
9. **se** ($v \in \overline{S}$) **e** (dist[v] $>$ dist[u] $+ w[u, v]$) **então**
10. { dist[v] $\leftarrow$ dist[u] $+ w[u, v]$; pred[v] $\leftarrow u$; }
11. **retorne** (A,dist).

Caminhos mínimos a partir de um vértice
Observações Finais

- O algoritmo de Dijkstra funciona tanto para grafos orientados quanto para grafos não orientados, desde que não haja arestas de peso negativo. (Por quê ?)
- Para grafos não orientados, a existência de uma aresta de peso negativo torna ambígua a definição de caminho de comprimento mínimo, já que o caminho pode *ziguezaguear* pela aresta de peso negativo produzindo um caminho indefinidamente curto.
- Em grafos orientados, a presença de arestas de peso negativo não necessariamente invalida a definição do problema, mas a existência de um *ciclo negativo*, isto é, um ciclo cuja soma dos pesos das arestas é negativo, sim.

Caminhos mínimos a partir de um vértice
Observações Finais

- Para **grafos orientados acíclicos** existe algoritmo de complexidade $O(|V|+|E|)$ para determinar os caminhos mínimos a partir de um vértice: basta calcular os caminhos mínimos analisando os vértices segundo a **ordenação topológica**.
- Para grafos orientados com arestas de peso negativo, mas sem ciclos negativos, é possível determinar os caminhos mínimos a partir de um vértice utilizando o **algoritmo de Bellman-Ford**, de complexidade $O(|V| \cdot |E|)$.
- **O algoritmo de Bellman-Ford também detecta a existência de ciclos negativos, um aspecto importante da solução do problema.**

Caminhos mínimos entre todos os pares

- Considere um grafo orientado ponderado $G = (V, E)$, com pesos não negativos nas arestas.
- **Como podemos determinar o caminho mínimo entre qualquer par de vértices do grafo G ?**
- Podemos aplicar o algoritmo de Dijkstra para cada um dos vértices do grafo. A complexidade desse algoritmo será:
  - $O(|V|^3)$ se utilizarmos a implementação com busca linear, boa para grafos densos;
  - $O((|V| \cdot |E|) \log |V|)$ se utilizarmos a implementação com busca na fila de prioridades (heap), boa para grafos esparsos;
- **O algoritmo que veremos a seguir é um exemplo do uso de programação dinâmica.**

Caminhos mínimos entre todos os pares
Projeto por indução e corretude

- Podemos também projetar por indução um algoritmo específico para esse problema.
- Para isso vamos supor que há uma rotulação $\{1, 2 \dots n\}$ atribuída aos vértices.
- Vamos considerar também que, em um dado instante, conhecemos, para cada par de vértices i e j, o comprimento do menor caminho que passa por vértices de rótulos **estritamente menores** que k.
- Dessa forma, conseguimos determinar o comprimento do menor caminho que passa apenas por vértices de rótulo **menores ou iguais** a k, apenas comparando o comprimento do menor caminho de i a j que passa pelo vértice k com o comprimento mínimo calculado até então.

$$\texttt{dist}^k[i,j] = \min\{\texttt{dist}^{k-1}[i,j], \texttt{dist}^{k-1}[i,k] + \texttt{dist}^{k-1}[k,j]\}$$

Caminhos mínimos entre todos os pares
Projeto por indução e corretude

- **Base:** Para $k = 1$, estamos considerando apenas caminhos mínimos entre pares de vértices i e j sem vértices intermediários. Se $i = j$ o caminho mínimo tem comprimento nulo. Se $i \neq j$, então o caminho mínimo tem comprimento d_e se existe aresta $e = (i, j)$, ou infinito caso contrário.
- **Hipótese:** Para todo par de vértices i e j, e para qualquer k, $1 \leq k \leq |V|$ sabemos determinar o comprimento do caminho mínimo de i a j que passa apenas por vértices intermediários de rótulo menor que k.
- **Passo:** Por hipótese de indução sabemos determinar, para um par de vértices i e j, o comprimento do caminho mínimo de i a j que passa apenas por vértices intermediários de rótulo menor que k.

Caminhos mínimos entre todos os pares
Projeto por indução e corretude

- **Passo (cont.):** Considerando agora caminhos de i a j que possam conter também o vértice k como intermediário, o comprimento do caminho mínimo de i a j só não será o mesmo de antes se houver um caminho passando pelo vértice k.

  Mas o menor caminho de i a j que passa por k é certamente o caminho dado pela concatenação dos menores caminhos de i a k e de k a j que passam apenas por vértices de rótulo menor que k, cujos comprimentos são conhecidos por **H.I.** (esse caminho será simples se for o mínimo de i a j).

  Então, basta comparar a soma dos comprimentos desses dois caminhos com o comprimento do menor caminho conhecido anteriormente e tomar o menor dos dois valores.

  Esse argumento vale para todo par de vértices i e j. □

Algoritmo de Floyd-Warshall

- A demonstração indutiva nos dá um algoritmo para determinar os comprimentos e os caminhos mínimos entre todos os pares de vértices.
- O algoritmo computa duas matrizes, cujas células na iteração k contêm as seguintes informações:
  - **Matriz** `dist`: registra em $\texttt{dist}[i,j]$ o comprimento do menor caminho de i a j passando por vértices de rótulo menor ou igual a k.
  - **Matriz** `pred`: registra em $\texttt{pred}[i,j]$ o predecessor de j no caminho mínimo de i a j passando por vértices de rótulo menor ou igual a k.

Algoritmo de Floyd-Warshall

A inicialização das matrizes D e P é feita da seguinte forma:

$$\texttt{dist}[i,j] = \begin{cases} 0, & \text{se } i = j; \\ d_e, & \text{se } e = (i,j) \in E; \\ \infty, & \text{caso contrário.} \end{cases}$$

$$\texttt{pred}[i,j] = \begin{cases} i, & \text{se } e = (i,j) \in E; \\ \texttt{NULO}, & \text{caso contrário;} \end{cases}$$

onde o valor NULO na matriz pred indica que não existe caminho de i a j quando $i \neq j$, ou que o caminho não contém arestas caso contrário.

Algoritmo de Floyd-Warshall - Pseudo-código

O pseudo-código do algoritmo de Floyd-Warshall para determinar os caminhos mínimos entre todos os pares de vértices será então:

FLOYD-WARSHALL(d)

```
1. inicializar dist e pred;
2. para k de 1 até n faça
3.    para i de 1 até n faça
4.       para j de 1 até n faça
5.          se dist[i,j] > dist[i,k] + dist[k,j] então
6.             dist[i,j] ← dist[i,k] + dist[k,j]
7.             pred[i,j] ← pred[k,j]
8. retorne (dist, pred).
```

Nota: da forma como dist é atualizada, o índice k da fórmula de recorrência anterior não precisa ser usado ! **(Por quê ?)**

Algoritmo de Floyd-Warshall - Complexidade

- A complexidade do algoritmo de Floyd-Warshall é $O(|V|^3)$, dada pelos três laços aninhados.
- Para grafos esparsos ainda é melhor usar o Algoritmo de Dijkstra a partir de todos os vértices ($O((|V| \cdot |E|) \log |V|)$ na implementação com *heap*).
- Já para grafos densos é melhor usar o Algoritmo de Floyd-Warshall, pois, apesar da complexidade assintótica ser a mesma de executar o o Algoritmo de Dijkstra partir de todos os vértices (na implementação com busca linear), a constante multiplicativa da função de complexidade é menor no Algoritmo de Floyd-Warshall e sua implementação é muito mais simples.

Algoritmo de Floyd-Warshall - Observações Finais

Existem ainda outras razões para usarmos o Algoritmo de Floyd-Warshall ao invés de executarmos o algoritmo de Dijkstra para cada vértice:

- O algoritmo de Floyd-Warshall funciona mesmo na presença de arestas de peso negativo, desde que não haja ciclos negativos **(por quê ?)**.
- Além disso, analisando a matriz resultado do algoritmo de Floyd-Warshall, é possível detectar a existência de ciclo negativo no grafo **(como ?)**.

Fecho Transitivo

Definição:

O *fecho transitivo* de um grafo orientado $G = (V, E)$ é um grafo orientado $C = (V, F)$ tal que existe uma aresta (i, j) em C se, e somente se, existir um caminho orientado de i a j em G.

Dado o fecho transitivo de um grafo G, é possível determinar se G é fortemente conexo: basta verificar se seu fecho transitivo é um grafo orientado completo.

Fecho Transitivo - Algoritmo

- Podemos utilizar mesma idéia indutiva do algoritmo de Floyd-Warshall para determinar o fecho transitivo de um grafo orientado.
- Existe caminho de um vértice i a um vértice j do grafo G passando apenas por vértices intermediários de rótulos *menores ou iguais* a k, se existe caminho de i a j passando por vértices intermediários de rótulos **estritamente menores** que k ou se existe caminho de i a k e de k a j, ambos tendo vértices intermediários de rótulos **estritamente menores** que k.
- Após $|V|$ iterações teremos determinado o fecho transitivo do grafo G.

Fecho Transitivo - Pseudo-código

- A matriz A registra, na k-ésima iteração, se existe caminho orientado de um vértice i para um vértice j passando por vértices de rótulo menor ou igual a k.
- A inicialização da matriz a é feita da seguinte forma:

$$A[i,j] = \begin{cases} 1, & \text{se } i = j \text{ ou se } (i,j) \in E \\ 0, & \text{caso contrário.} \end{cases}$$

- Ao final da execução A é a matriz de adjacência do fecho transitivo.

Fecho Transitivo - Pseudo-código

O pseudo-código do algoritmo para determinar o fecho transitivo de um grafo será então:

FECHO(G)

```
▷ Entrada: grafo G
▷ Saída: matriz de adjacências do fecho transitivo de G
1.  inicializar A;
2.  para k de 1 até n faça
3.     para i de 1 até n faça
4.        para j de 1 até n faça
5.           se A[i,j] ≠ 1 e (A[i,k] = 1 e A[k,j] = 1) então
6.              A[i,j] ← 1
7.  retorne(A)
```


Fecho Transitivo - Complexidade

- A complexidade do algoritmo para determinação do fecho transitivo é $O(|V|^3)$, dada pelos três laços aninhados.
- Verificar, a partir do fecho transitivo, se o grafo é fortemente conexo leva tempo $O(|V|^2)$.
- Podemos também computar o fecho transitivo do grafo efetuando um percurso (em largura ou profundidade) a partir de cada vértice. Esse algoritmo tem complexidade $O(|V|(|V| + |E|))$.
- Para grafos esparsos, aplicar percurso em cada vértice é mais eficiente. Para grafos densos, apesar dos algoritmos serem assintoticamente equivalentes, é mais interessante usar o algoritmo do fecho transitivo que possui constante multiplicativa menor e é mais fácil de implementar.

O algoritmo de Bellman-Ford

- Como vimos, o algoritmo de Dijkstra não se aplica para o caso de **grafos orientados** que contenham arcos com peso **negativo**.
- O **algoritmo de Bellman-Ford** não só calcula caminhos mínimos na presença de arcos com pesos negativos, como também consegue **detectar a existência de ciclos negativos**.
- Para entender o funcionamento deste algoritmo, vamos formalizar o problema que estamos querendo resolver:

> Dado um grafo orientado $G = (V, E)$, uma função $d : E \rightarrow \mathbb{R}$ definindo as distâncias entre os extremos dos seus arcos, e um vértice r de V, determinar, para todo vértice u de G um caminho C de distância mínima de r a u, **ou então**, identificar a existência de um ciclo negativo em G.

O algoritmo de Bellman-Ford

- O algoritmo de Bellman-Ford é um exemplo de aplicação da técnica de **programação dinâmica** no projeto de algoritmos.
- Suponha que $G = (V, E)$ seja o grafo de entrada. Para todo, $k \in \{0, \ldots, |V|\}$ e todo $u \in V$, seja $\texttt{dist}^k[u]$ o comprimento do caminho mínimo de r até u usando **no máximo** k arcos.
- Pode-se **projetar um algoritmo por indução** para o seguinte problema

> Dado um grafo orientado $G = (V, E)$, uma função $d : E \rightarrow \mathbb{R}$ definindo as distâncias entre os extremos dos seus arcos, e um vértice r de V, determinar, para todo vértice u de G um caminho C de distância mínima de r a u usando no máximo k arcos, onde $k \in \{0, \ldots, |V|\}$

O algoritmo de Bellman-Ford

- **Base**: para $k = 0$, tem-se que $\texttt{dist}^k[r] = 0$ e $\texttt{dist}^k[u] = \infty$, para todo $u \in V - \{r\}$.
- **Hipótese indutiva**: supor que o resultado é verdadeiro para $k - 1$.
- **Passo**: para provar que o resultado é verdadeiro para k, basta considerar a seguinte **fórmula de recorrência**:

$$\texttt{dist}^k[v] = \min\{\min_{u:(u,v)\in E}\{d_{uv} + \texttt{dist}^{k-1}[u]\}, \texttt{dist}^{k-1}[v]\}.$$

- Esta fórmula nos diz que o comprimento do caminho mínimo de r para v que usa no máximo k arcos, deve ser formado por um caminho mínimo de no máximo $k - 1$ arcos que chegue em algum vértice u que tenha v como adjacente e pelo arco (u, v), ou então, é o próprio caminho mínimo de r a v que usa até $k - 1$ arcos.

O algoritmo de Bellman-Ford

- Para resolver o problema de detectar a existência de ciclo negativo, basta notar que se $\texttt{dist}^{|V|}[u] < \texttt{dist}^{|V|-1}[u]$ para algum vértice u de V, então existe um caminho com $|V|$ arcos, ou seja, com algum ciclo (por quê ?), que vai de r para u e que é menor do que qualquer outro caminho de r para u que **não** contém ciclo.

No pseudo-código dado a seguir, o vetor a é usado para armazenar o valor de $\texttt{dist}^k$ a partir de $\texttt{dist}^{k-1}$ e a variável `CICLO` é usada para identificar se foi (`verdadeiro`) ou não (`falso`) encontrado um ciclo negativo no grafo G alcançável a partir da origem r.

Algoritmo de Bellman-Ford: pseudo-código

Bellman-Ford(G, d, r)

▷ **Saída**: vetor `dist` com o comprimento dos caminhos mais curtos e a variável booleana `CICLO` que é verdadeira *se e somente se* existir ciclo negativo em G

1. **para** todo $v \in V - \{r\}$ **faça** $\texttt{dist}[v] \leftarrow \infty$;
2. $\texttt{dist}[r] \leftarrow 0$; $\texttt{continua} \leftarrow \texttt{verdadeiro}$; $k \leftarrow 1$;
3. **enquanto** $(k \leq |V|)$ **e** $(\texttt{continua})$ **faça**
4.   **para** todo $u \in V$ **faça**
5.   **para** todo $v \in \text{Adj}[u]$ **faça**
6.   **se** $(\texttt{dist}[v] > \texttt{dist}[u] + d[u, v])$ **então**
7. $\{ a[v] \leftarrow \texttt{dist}[u] + d[u, v]$; $\texttt{pred}[v] \leftarrow u$; $\}$
8. $\texttt{continua} \leftarrow \texttt{falso}$;
9.   **para** todo $v \in V$ **faça**
10.   **se** $(a[v] \neq \texttt{dist}[v])$ **então** $\texttt{continua} \leftarrow \texttt{verdadeiro}$;
11. $\texttt{dist}[v] \leftarrow a[v]$;
12. $k \leftarrow k + 1$;
13. $\texttt{CICLO} \leftarrow ((\texttt{continua})$ **e** $(k = |V| + 1))$;
14. **retorne**$(\texttt{CICLO}, \texttt{dist}, \texttt{pred})$.

O algoritmo de Bellman-Ford: exemplo e observações

- É fácil ver que a complexidade do algoritmo dado na transparência anterior é $O(|E|.|V| + |V|^2)$. Isto ocorre porque o laço das linhas 3–12 é executado $O(|V|)$ vezes e *(i)* as linhas 5–7 tem complexidade *agregada* $O(|E|)$ e *(ii)* o laço das linhas 9–11 tem complexidade $O(|V|)$.
- Podemos supor que todos os vértices sejam alcançáveis a partir de r. Caso contrário, limitamo-nos a trabalhar com a componente fortemente conexa de r (que pode ser encontrada em $O(|V| + |E|)$). Neste caso, $|E| \in \Omega(|V|)$ e, assim, a complexidade do algoritmo é dada por $O(|E|.|V|)$.
- É possível ainda eliminar a necessidade de utilização do vetor auxiliar $\underline{a}$. No entanto, a prova de corretude do algoritmo fica um pouco menos clara (veja por exemplo o livro do Cormen).
- **Exemplo**: grafo ...