Paradigmas de Projeto de Algoritmos*

Última alteração: 2 de Abril de 2004

*Transparências elaboradas por Charles Ornelas Almeida e Nivio Ziviani

Paradigmas de Projeto de Algoritmos

- indução,
- recursividade,
- algoritmos tentativa e erro,
- divisão e conquista,
- balanceamento,
- programação dinâmica,
- algoritmos gulosos,
- algoritmos aproximados.

Indução Matemática

- É útil para provar asserções sobre a correção e a eficiência de algoritmos.
- Consiste em inferir uma lei geral a partir de instâncias particulares.
- Seja T um teorema que que tenha como parâmetro um número natural n. Para provar que T é válido para todos os valores de n, provamos que:
  1. T é válido para $n = 1$;
  2. Para todo $n > 1$, se T é válido para $n - 1$, então T é válido para n.
- A condição 1 é chamada de **passo base**.
- Provar a condição 2 é geralmente mais fácil que provar o teorema diretamente (podemos usar a asserção de que T é válido para $n - 1$.
- Esta afirmativa é chamada de **hipótese de indução** ou **passo indutivo**
- As condições 1 e 2 implicam T válido para $n = 2$, o que junto com a condição 2 implica T também válido para $n = 3$, e assim por diante.

Exemplo de Indução Matemática

$S(n) = 1 + 2 + \cdots + n = n(n + 1)/2$

- Para $n = 1$ a asserção é verdadeira, pois $S(1) = 1 = 1 \times (1 + 1)/2$ (passo base).
- Assumimos que a soma dos primeiros n números naturais $S(n)$ é $n(n + 1)/2$ (hipótese de indução).
- Pela definição de $S(n)$ sabemos que $S(n + 1) = S(n) + n + 1$.
- Usando a hipótese de indução, $S(n+1) = n(n+1)/2+n+1 = (n+1)(n+2)/2$, que é exatamente o que queremos provar.

Limite Superior de Equações de Recorrência

- A solução de uma equação de recorrência pode ser difícil de ser obtida.
- Nestes casos, pode ser mais fácil tentar advinhar a solução ou obter um limite superior para a ordem de complexidade.
- Advinhar a solução funciona bem quando estamos interessados apenas em um limite superior, ao invés da solução exata.
- Mostrar que um certo limite existe é mais fácil do que obter o limite.
- Ex.: $T(2n) \leq 2T(n) + 2n - 1, \;\; T(2) = 1,$ definida para valores de n que são potências de 2.
  - O objetivo é encontrar um limite superior na notação O, onde o lado direito da desigualdade representa o pior caso.

Indução Matemática para Resolver Equação de Recorrência

$T(2n) \leq 2T(n) + 2n - 1, \;\; T(2) = 1$, definida para valores de n que são potências de 2.

- Procuramos $f(n)$ tal que $T(n) \leq O(f(n))$, mas fazendo com que $f(n)$ seja o mais próximo possível da solução real para $T(n)$.
- Vamos considerar o palpite $f(n) = n^2$.
- Queremos provar que $T(n) = O(f(n))$ utilizando indução matemática em n.
- Passo base: $T(2) = 1 \leq f(2) = 4$.
- Passo de indução: provar que $T(n) \leq f(n)$ implica $T(2n) \leq f(2n)$.

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1, && \text{(def. da recorrência)} \\ &\leq 2n^2 + 2n - 1, && \text{(hipótese de indução)} \\ &< (2n)^2, \end{aligned}$$

que é exatamente o que queremos provar. Logo, $T(n) = O(n^2)$.

Indução Matemática para Resolver Equação de Recorrência

- Vamos tentar um palpite menor, $f(n) = cn$, para alguma constante c.
- Queremos provar que $T(n) \leq cn$ implica em $T(2n) \leq c2n$. Assim:

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1, && \text{(def. da recorrência)} \\ &\leq 2cn + 2n - 1, && \text{(hipótese de indução)} \\ &> c2n. \end{aligned}$$

- cn cresce mais lentamente que $T(n)$, pois $c2n = 2cn$ e não existe espaço para o valor $2n - 1$.
- Logo, $T(n)$ está entre cn e n^2.

Indução Matemática para Resolver Equação de Recorrência

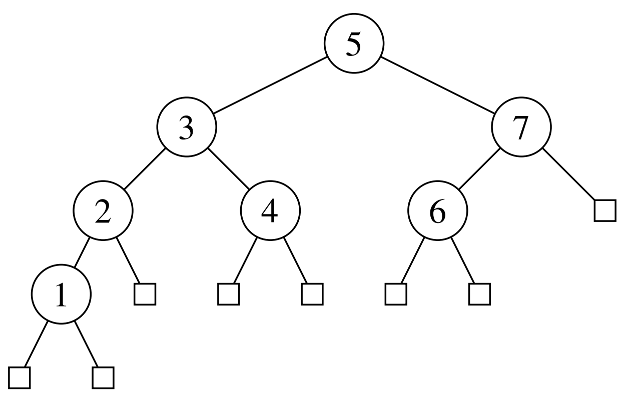
- Vamos então tentar $f(n) = n \log n$.
- Passo base: $T(2) < 2 \log 2$.
- Passo de indução: vamos assumir que $T(n) \leq n \log n$.
- Queremos mostrar que $T(2n) \leq 2n \log 2n$. Assim:

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1, && \text{(def. da recorrência)} \\ &\leq 2n \log n + 2n - 1, && \text{(hipótese de indução)} \\ &< 2n \log 2n, \end{aligned}$$

- A diferença entre as fórmulas agora é de apenas 1.
- De fato, $T(n) = n \log n - n + 1$ é a solução exata de $T(n) = 2T(n/2) + n - 1, \;\; T(1) = 0$, que descreve o comportamento do algoritmo de ordenação *Mergesort*.

Recursividade

- Um procedimento que chama a si mesmo, direta ou indiretamente, é dito ser **recursivo**.
- Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.
- Ex.: árvore binária de pesquisa:
  - Todos os registros com chaves menores estão na subárvore esquerda;
  - Todos os registros com chaves maiores estão na subárvore direita.




```
type Registro = record Chave: integer; end;
     Apontador = ^Nodo;
     Nodo = record
              Reg: Registro;
              Esq, Dir: Apontador;
            end;
```

Recursividade

- Algoritmo para percorrer todos os registros em ordem de **caminhamento central**:
  1. caminha na subárvore esquerda na ordem central;
  2. visita a raiz;
  3. caminha na subárvore direita na ordem central.
- No caminhamento central, os nodos são visitados em ordem lexicográfica das chaves.

```
procedure Central (p : Apontador);
begin
  if p <> nil
  then begin
         Central(p^.Esq);
         writeln(p^.Reg.Chave);
         Central(p^.Dir);
       end;
end;
```

Implementação de Recursividade

- Usa-se uma **pilha** para armazenar os dados usados em cada chamada de um procedimento que ainda não terminou.
- Todos os dados não globais vão para a pilha, registrando o estado corrente da computação.
- Quando uma ativação anterior prossegue, os dados da pilha são recuperados.
- No caso do caminhamento central:
  - para cada chamada recursiva, o valor de p e o endereço de retorno da chamada recursiva são armazenados na pilha.
  - Quando encontra p$=$nil o procedimento retorna para quem chamou utilizando o endereço de retorno que está no topo da pilha.

Problema de Terminação em Procedimentos Recursivos

- Procedimentos recursivos introduzem a possibilidade de iterações que podem não terminar: existe a necessidade de considerar o problema de **terminação**.
- É fundamental que a chamada recursiva a um procedimento P esteja sujeita a uma condição B, a qual se torna não-satisfeita em algum momento da computação.
- Esquema para procedimentos recursivos: composição $\mathcal{C}$ de comandos S_i e P.
  $P \equiv$ **if** B **then** $\mathcal{C}[S_i, P]$
- Para demonstrar que uma repetição termina, define-se uma função $f(x)$, sendo x o conjunto de variáveis do programa, tal que:
  1. $f(x) \leq 0$ implica na condição de terminação;
  2. $f(x)$ é decrementada a cada iteração.

Problema de Terminação em Procedimentos Recursivos

- Uma forma simples de garantir terminação é associar um parâmetro n para P (no caso **por valor**) e chamar P recursivamente com $n - 1$.
- A substituição da condição B por $n > 0$ garante terminação.
  $P \equiv$ **if** $n > 0$ **then** $\mathcal{P}[S_i, P(n-1)]$
- É necessário mostrar que o nível mais profundo de recursão é finito, e também possa ser mantido pequeno, pois cada ativação recursiva usa uma parcela de memória para acomodar as variáveis.

Quando Não Usar Recursividade

- Nem todo problema de natureza recursiva deve ser resolvido com um algoritmo recursivo.
- Estes podem ser caracterizados pelo esquema $P \equiv$ **if** B **then** (S, P)
- Tais programas são facilmente transformáveis em uma versão não recursiva
  $P \equiv (x := x_0;$ **while** B **do** $S)$

Exemplo de Quando Não Usar Recursividade

- Cálculo dos **números de Fibonacci**
  $f_0 = 0, f_1 = 1,$
  $f_n = f_{n-1} + f_{n-2} \text{para} n \geq 2$
- Solução: $f_n = \frac{1}{\sqrt{5}}[\Phi^n - (-\Phi)^{-n}]$, onde $\Phi = (1 + \sqrt{5})/2 \approx 1,618$ é a **razão de ouro**.
- O procedimento recursivo obtido diretamente da equação é o seguinte:

```
function FibRec(n:integer): integer;
begin
  if n < 2
  then FibRec := n
  else FibRec := FibRec(n-1) + FibRec(n-2);
end;
```

- O programa é extremamente ineficiente porque recalcula o mesmo valor várias vezes.
- Neste caso, a complexidade de espaço para calcular f_n é $O(\Phi^n)$.
- Considerando que a medida de complexidade de tempo $f(n)$ é o número de adições, então $f(n) = O(\Phi^n)$.

Versão iterativa do Cálculo de Fibonacci

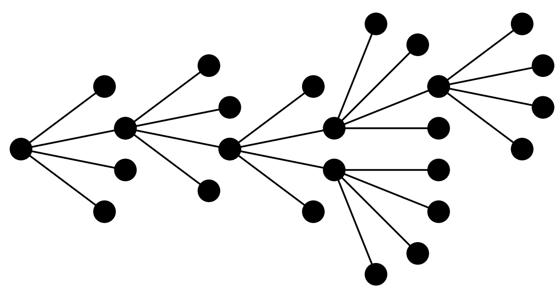
```
function FibIter(n: integer): integer;
var i, k, F: integer;
begin
  i := 1; F :=0;
  for k := 1 to n do
    begin
    F := i + F;
    i := F − i;
    end;
  FibIter := F;
end;
```

- O programa tem complexidade de tempo $O(n)$ e complexidade de espaço $O(1)$.
- Devemos evitar uso de recursividade quando existe solução óbvia por iteração.
- Comparação versões recursiva e iterativa:

n	20	30	50	100
Recursiva	1 seg	2 min	21 dias	10^9 anos
Iterativa	1/3 mseg	1/2 mseg	3/4 mseg	1,5 mseg

Algoritmos Tentativa e Erro (*Backtracking*)

- **Tentativa e erro**: decompor o processo em um número finito de subtarefas parciais que devem ser exploradas exaustivamente.
- O processo de tentativa gradualmente constrói e percorre uma árvore de subtarefas.



- Algoritmos tentativa e erro não seguem regra fixa de computação:
  - Passos em direção à solução final são tentados e registrados;
  - Caso esses passos tomados não levem à solução final, eles podem ser retirados e apagados do registro.
- Quando a pesquisa na árvore de soluções cresce rapidamente é necessário usar **algoritmos aproximados** ou **heurísticas** que não garantem a solução ótima mas são rápidas.

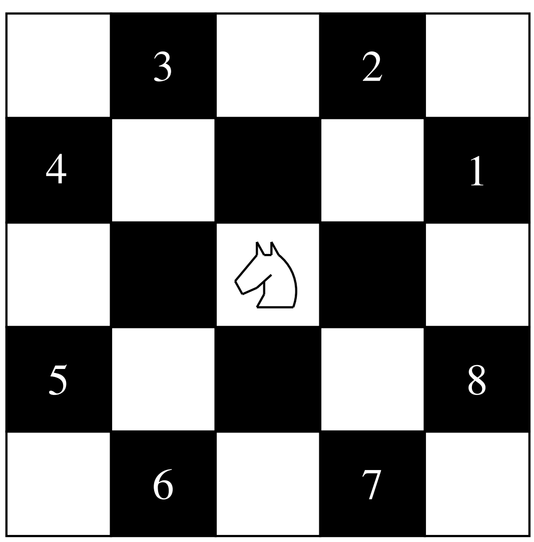
Backtracking: Passeio do Cavalo

- Tabuleiro com $n \times n$ posições: cavalo se movimenta segundo regras do xadrez.
- Problema: a partir de $(x_0,\ y_0)$, encontrar, se existir, um passeio do cavalo que visita todos os pontos do tabuleiro uma única vez.
- Tenta um próximo movimento:

```
procedure Tenta;
begin
inicializa selecao de movimentos;
repeat
  seleciona proximo candidato ao movimento;
  if aceitavel
  then begin
         registra movimento;
         if tabuleiro nao esta cheio
         then begin
                tenta novo movimento;
                if nao sucedido then apaga registro anterior;
              end;
       end;
until (mov. bem sucedido) ou (acabaram–se cand. movimento);
end;
```

Exemplo de Backtracking - Passeio do Cavalo

- O tabuleiro pode ser representado por uma matriz $n \times n$.
- A situação de cada posição pode ser representada por um inteiro para recordar o histórico das ocupações:
  - t[x,y] = 0, campo $< x, y >$ não visitado,
  - t[x,y] = i, campo $< x, y >$ visitado no i-ésimo movimento, $1 \leq i \leq n^2$.
- Regras do xadrez para os movimentos do cavalo:



Implementação do Passeio do Cavalo

```
program PasseioCavalo;
const n = 8; { Tamanho do lado do tabuleiro }
type indice = 1..n;
var i, j: integer;
    t: array[indice, indice] of integer;
    q: boolean;
    s: set of indice;
    a, b: array[indice] of integer;
{—— Entra aqui o procedimento Tenta mostrado a seguir ——}
begin { programa principal }
  s := [1,2,3,4,5,6,7,8];
  a[1] := 2;  a[2] := 1;  a[3] := −1; a[4] := −2;
  b[1] := 1;  b[2] := 2;  b[3] := 2; b[4] := 1;
  a[5] := −2; a[6] := −1; a[7] := 1; a[8] := 2;
  b[5] := −1; b[6] := −2; b[7] := −2; b[8] := −1;
  for i := 1 to n do for j := 1 to n do t[i,j] := 0;
  t[1,1] := 1; { escolhemos uma casa do tabuleiro }
  Tenta(2,1,1,q);
  if q
  then for i:=1 to n do
       begin for j:=1 to n do write(t[i,j]:4); writeln; end
  else writeln('Sem solucao');
end.
```

Implementação do Passeio do Cavalo

```
procedure Tenta(i:integer; x,y:indice; var q:boolean);
var u, v, k: integer; q1: boolean;
begin
k := 0;  { inicializa selecao de movimentos }
repeat
  k := k + 1; q1 := false;
  u := x + a[k]; v := y + b[k];
  if (u in s) and (v in s)
  then if t[u,v]=0
         then begin
                t[u,v] := i;
                if i < n*n { tabuleiro nao esta cheio }
                then begin
                       Tenta(i+1, u, v, q1); {tenta novo movimento}
                       if not q1
       then t[u,v] := 0 {nao sucedido apaga registro anterior}
                     end
                else q1 := true;
                end;
{ nao existem mais casas a visitar a partir de x,y }
until q1 or (k = 8);
q := q1;
end;
```

Divisão e Conquista

- Consiste em dividir o problema em partes menores, encontrar soluções para as partes, e combiná-las em uma solução global.
- Exemplo: encontrar o maior e o menor elemento de um vetor de inteiros, $A[1..n]$, $n \geq 1$.

```
procedure MaxMin4(Linf,Lsup:integer; var Max,Min:integer);
var Max1, Max2, Min1, Min2, Meio: integer;
begin
  if Lsup − Linf <= 1
  then if A[Linf] < A[Lsup]
       then begin Max := A[Lsup]; Min := A[Linf]; end
       else begin Max := A[Linf]; Min := A[Lsup]; end
  else begin
       Meio := (Linf + Lsup) div 2;
       MaxMin4(Linf, Meio, Max1, Min1);
       MaxMin4(Meio+1, Lsup, Max2, Min2);
       if Max1 > Max2 then Max := Max1 else Max := Max2;
       if Min1 < Min2 then Min := Min1 else Min := Min2;
       end;
end;
```

- Cada chamada de MaxMin4 atribui à Max e Min o maior e o menor elemento em $A[\text{Linf}], A[\text{Linf}+1], \cdots, A[\text{Lsup}]$, respectivamente.

Divisão e Conquista - Análise do Exemplo

- Seja $f(n)$ o número de comparações entre os elementos de A, se A contiver n elementos.

$$
\begin{array}{ll}
f(n) = 1, & \text{para } n \leq 2, \\
f(n) = f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + 2, & \text{para } n > 2.
\end{array}
$$

- Quando $n = 2^i$ para algum inteiro positivo i:

$$
\begin{array}{rcl}
f(n) & = & 2f(n/2) + 2 \\
2f(n/2) & = & 4f(n/4) + 2 \times 2 \\
\vdots & & \vdots \\
2^{i-2}f(n/2^{i-2}) & = & 2^{i-1}f(n/2^{i-1}) + 2^{i-1}
\end{array}
$$

- Adicionando lado a lado, obtemos:

$$
\begin{aligned}
f(n) &= 2^{i-1}f(n/2^{i-1}) + \textstyle\sum_{k=1}^{i-1} 2^k \\
&= 2^{i-1}f(2) + 2^i - 2 \\
&= 2^{i-1} + 2^i - 2 \\
&= \tfrac{3n}{2} - 2.
\end{aligned}
$$

- Logo, $f(n) = 3n/2 - 2$ para o melhor caso, pior caso e caso médio.

Divisão e Conquista - Análise do Exemplo

- Conforme mostrado no Capítulo 1, o algoritmo dado neste exemplo é **ótimo**.
- Entretanto, ele pode ser pior do que os apresentados no Capítulo 1, pois, a cada chamada recursiva, salva Linf, Lsup, Max e Min, além do endereço de retorno da chamada para o procedimento.
- Além disso, uma comparação adicional é necessária a cada chamada recursiva para verificar se Lsup $-$ Linf ≤ 1.
- $n + 1$ deve ser menor do que a metade do maior inteiro que pode ser representado pelo compilador, para não provocar *overflow* na operação Linf $+$ Lsup.

Balanceamento

- No projeto de algoritmos, é importante procurar sempre manter o **balanceamento** na subdivisão de um problema em partes menores.
- Divisão e conquista não é a única técnica em que balanceamento é útil.

Vamos considerar um exemplo de ordenação

- Seleciona o menor elemento de $A[1..n]$ e troca-o com o primeiro elemento $A[1]$.
- Repete o processo com os $n - 1$ elementos, resultando no segundo maior elemento, o qual é trocado com o segundo elemento $A[2]$.
- Repetindo para $n - 2$, $n - 3$, $\ldots$, 2 ordena a seqüência.

Balanceamento - Análise do Exemplo

- O algoritmo leva à equação de recorrência: $T(n) = T(n-1) + n - 1, \quad T(1) = 0$, para o número de comparações entre elementos.
- Substituindo:

$$\begin{aligned} T(n) &= T(n-1) + n - 1 \\ T(n-1) &= T(n-2) + n - 2 \\ &\vdots \\ T(2) &= T(1) + 1 \end{aligned}$$

- Adicionando lado a lado, obtemos: $T(n) = T(1) + 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$.
- Logo, o algorimo é $O(n^2)$.
- Embora o algoritmo possa ser visto como uma aplicação recursiva de divisão e conquista, ele não é eficiente para valores grandes de n.
- Para obter eficiência assintotica é necessário **b**alanceamento: dividir em dois subproblemas de tamanhos aproximadamente iguais, ao invés de um de tamanho 1 e o outro de tamanho $n - 1$.

Exemplo de Balanceamento - Mergesort

- **Intercalação**: unir dois arquivos ordenados gerando um terceiro arquivo ordenado (*merge*).
- Colocar no terceiro arquivo o menor elemento entre os menores dos dois arquivos iniciais, desconsiderando este mesmo elemento nos passos posteriores.
- Este processo deve ser repetido até que todos os elementos dos arquivos de entrada sejam escolhidos.
- Algoritmo de ordenação (**Mergesort**):
  - dividir recursivamente o vetor a ser ordenado em dois, até obter n vetores de 1 único elemento.
  - Aplicar a intercalação tendo como entrada 2 vetores de um elemento, formando um vetor ordenado de dois elementos.
  - Repetir este processo formando vetores ordenados cada vez maiores até que todo o vetor esteja ordenado.

Exemplo de Balanceamento - Implementação do Mergesort

```
procedure Mergesort(var A:array[1..n] of integer; i,j:integer);
begin { Intercala A[i..m] e A[m+1..j] em A[i..j] }
  if i < j
  then begin
         m := (i + j − 1)/2;
         Mergesort(A, i, m);
         Mergesort(A, m+1, j);
         Merge(A, i, m, j);
         end;
end;
```

- Considere n como sendo uma potência de 2.
- *Merge*(A, i, m, j) recebe duas seqüências ordenadas $A[i..m]$ e $A[(m+1)..j]$ e produz uma outra seqüência ordenada dos elementos de $A[i..m]$ e $A[m+1..j]$.
- Como $A[i..m]$ e $A[m+1..j]$ estão ordenados, *Merge* requer no máximo $n-1$ comparações.
- *Merge* seleciona repetidamente o menor dentre os menores elementos restantes em $A[i..m]$ e $A[m+1..j]$. Caso empate, retira de qualquer uma delas.

Análise do Mergesort

- Na contagem de comparações, o comportamento do Mergesort pode ser representado por:
  $T(n) = 2T(n/2) + n - 1, \quad T(1) = 0$
- No caso da equação acima temos:

$$\begin{array}{rcl} T(n) & = & 2T(n/2) + n - 1 \\ 2T(n/2) & = & 2^2T(n/2^2) + 2\frac{n}{2} - 2 \times 1 \\ \vdots & & \vdots \\ 2^{i-1}T(n/2^{i-1}) & = & 2^iT(n/2^i) + 2^{i-1}\frac{n}{2^{i-1}} - 2^{i-1} \end{array}$$

- Adicionando lado a lado:

$$\begin{array}{rcl} T(n) & = & 2^iT(n/2^i) + \sum_{k=0}^{i-1} n - \sum_{k=0}^{i-1} 2^k \\ & = & in - \frac{2^{i-1+1} - 1}{2 - 1} \\ & = & n \log n - n + 1. \end{array}$$

- Logo, o algorimo é $O(n \log n)$.
- Para valores grandes de n, o balanceamento levou a um resultado muito superior, saimos de $O(n^2)$ para $O(n \log n)$.

Programação Dinâmica

- Quando a soma dos tamanhos dos subproblemas é $O(n)$ então é provável que o algoritmo recursivo tenha **complexidade polinomial**.
- Quando a divisão de um problema de tamanho n resulta em n subproblemas de tamanho $n - 1$ então é provável que o algoritmo recursivo tenha **complexidade exponencial**.
- Nesse caso, a técnica de programação dinâmica pode levar a um algoritmo mais eficiente.
- A programação dinâmica calcula a solução para todos os subproblemas, partindo dos subproblemas menores para os maiores, armazenando os resultados em uma tabela.
- A vantagem é que uma vez que um subproblema é resolvido, a resposta é armazenada em uma tabela e nunca mais é recalculado.

Programação Dinâmica - Exemplo

Produto de n matrizes

- $M = M_1 \times M_2 \times \cdots \times M_n$, onde cada M_i é uma matriz com d_{i-1} linhas e d_i colunas.
- A ordem da multiplicação pode ter um efeito enorme no número total de operações de adição e multiplicação necessárias para obter M.
- Considere o produto de uma matriz $p \times q$ por outra matriz $q \times r$ cujo algoritmo requer $O(pqr)$ operações.
- Considere o produto $M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$, onde as dimensões de cada matriz está mostrada entre colchetes.
- A avaliação de M na ordem $M = M_1 \times (M_2 \times (M_3 \times M_4))$ requer 125.000 operações, enquanto na ordem $M = (M_1 \times (M_2 \times M_3)) \times M_4$ requer apenas 2.200.

Programação Dinâmica - Exemplo

- Tentar todas as ordens possíveis para minimizar o número de operações $f(n)$ é exponencial em n, onde $f(n) \geq 2^{n-2}$.
- Usando programação dinâmica é possível obter um algoritmo $O(n^3)$.
- Seja m_{ij} menor custo para computar $M_i \times M_{i+1} \times \cdots \times M_j$, para $1 \leq i \leq j \leq n$.
- Neste caso,

$$m_{ij} = \begin{cases} 0, & \text{se } i = j, \\ \text{Min}_{i \leq k < j} \; (m_{ik} + m_{k+1,j} + d_{i-1} d_k d_j), & \text{se } j > i. \end{cases}$$

- m_{ik} representa o custo mínimo para calcular $M' = M_i \times M_{i+1} \times \cdots \times M_k$
- $m_{k+1,j}$ representa o custo mínimo para calcular $M'' = M_{k+1} \times M_{k+2} \times \cdots \times M_j$.
- $d_{i-1} d_k d_j$ representa o custo de multiplicar $M'[d_{i-1}, d_k]$ por $M''[d_k, d_j]$.
- $m_{ij},\ j > i$ representa o custo mínimo de todos os valores possíveis de k entre i e $j - 1$, da soma dos três termos.

Programação Dinâmica - Exemplo

- O enfoque programação dinâmica calcula os valores de m_{ij} na ordem crescente das diferenças nos subscritos.
- O calculo inicia com m_{ii} para todo i, depois $m_{i,i+1}$ para todo i, depois $m_{i,i+2}$, e assim sucessivamente.
- Desta forma, os valores m_{ik} e $m_{k+1,j}$ estarão disponíveis no momento de calcular m_{ij}.
- Isto acontece porque $j - i$ tem que ser estritamente maior do que ambos os valores de $k - i$ e $j - (k + 1)$ se k estiver no intervalo $i \leq k < j$.
- Programa para computar a ordem de multiplicação de n matrizes, $M_1 \times M_2 \times \cdots \times M_n$, de forma a obter o menor número possível de operações.

Programação Dinâmica - Implementação

```
program AvaliaMultMatrizes;
const Maxn = 10;
var i, j, k, h, n, temp: integer;
    d: array[0..Maxn] of integer;
    m: array[1..Maxn, 1..Maxn] of integer;
begin
  write('Numero de matrizes n:'); readln(n);
  write('Dimensoes das matrizes:');
  for i := 0 to n do read(d[i]);
  for i := 1 to n do m[i,i] := 0;
  for h := 1 to n−1 do
    begin
    for i := 1 to n−h do
      begin
      j := i + h; m[i,j] := MaxInt;
      for k := i to j−1 do
        begin
        temp := m[i,k] + m[k+1,j] + d[i−1]∗d[k]∗d[j];
        if temp < m[i,j] then m[i,j] := temp;
        end;
      write(' m[',i:1,',',j:1,']=',m[i,j]);
      end;
    writeln;
    end;
end.
```

Programação Dinâmica - Implementação

- A execução do programa obtém o custo mínimo para multiplicar as n matrizes, assumindo que são necessárias pqr operações para multiplicar uma matriz $p \times q$ por outra matriz $q \times r$.

- A execução do programa para as quatro matrizes onde d_0, d_1, d_2, d_3, d_4 são 10, 20, 50, 1, 100, resulta:

$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
$m_{12} = 10.000$	$m_{23} = 1.000$	$m_{34} = 5.000$	
$m_{13} = 1.200$	$m_{24} = 3.000$		
$m_{14} = 2.200$			

Programação Dinâmica - Princípio da Otimalidade

- A ordem de multiplicação pode ser obtida registrando o valor de k para cada entrada da tabela que resultou no mínimo.
- Essa solução eficiente está baseada no **princípio da otimalidade**:
  - em uma seqüência ótima de escolhas ou de decisões cada subseqüência deve também ser ótima.
- Cada subseqüência representa o custo mínimo, assim como m_{ij}, $j > i$.
- Assim, todos os valores da tabela representam escolhas ótimas.
- O princípio da otimalidade não pode ser aplicado indiscriminadamente.
- Quando o princípio não se aplica é provável que não se possa resolver o problema com sucesso por meio de programação dinâmica.

Aplicação do Princípio da Otimalidade

- Por exemplo, quando o problema utiliza recursos limitados, quando o total de recursos usados nas subinstâncias é maior do que os recursos disponíveis.

- Se o caminho mais curto entre Belo Horizonte e Curitiba passa por Campinas:
  - o caminho entre Belo Horizonte e Campinas também é o mais curto possível
  - assim como o caminho entre Campinas e Curitiba.
  - Logo, o princípio da otimalidade se aplica.

Não Aplicação do Princípio da Otimalidade

- No problema de encontrar o caminho mais longo entre duas cidades:
  - Um caminho simples nunca visita uma mesma cidade duas vezes.
  - Se o caminho mais longo entre Belo Horizonte e Curitiba passa por Campinas, isso não significa que o caminho possa ser obtido tomando o caminho simples mais longo entre Belo Horizonte e Campinas e depois o caminho simples mais longo entre Campinas e Curitiba.
  - Quando os dois caminhos simples são ajuntados é pouco provável que o caminho resultante também seja simples.
  - Logo, o princípio da otimalidade não se aplica.

Algoritmos Gulosos

- Resolve problemas de otimização.
- Exemplo: algoritmo para encontrar o caminho mais curto entre dois vértices de um grafo.
  - Escolhe a aresta que parece mais promissora em qualquer instante;
  - Independente do que possa acontecer mais tarde, nunca reconsidera a decisão.
- Não necessita avaliar alternativas, ou usar procedimentos sofisticados para desfazer decisões tomadas previamente.
- Problema geral: dado um conjunto C, determine um subconjunto $S \subseteq C$ tal que:
  - S satisfaz uma dada propriedade P, e
  - S é mínimo (ou máximo) em relação a algum critério α.
- O **algoritmo guloso** para resolver o problema geral consiste em um processo iterativo em que S é construído adicionando-se ao mesmo elementos de C um a um.

Características dos Algoritmos Gulosos

- Para construir a solução ótima existe um conjunto ou lista de candidatos.
- São acumulados um conjunto de candidatos considerados e escolhidos, e o outro de candidatos considerados e rejeitados.
- Existe função que verifica se um conjunto particular de candidatos produz uma *solução* (sem considerar otimalidade no momento).
- Outra função verifica se um conjunto de candidatos é *viável* (também sem preocupar com a otimalidade).
- Uma *função de seleção* indica a qualquer momento quais dos candidatos restantes é o mais promissor.
- Uma *função objetivo* fornece o valor da solução encontrada, como o comprimento do caminho construído (não aparece de forma explicita no algoritmo guloso).

Pseudo Código de Algoritmo Guloso

```
function Guloso (C: conjunto): Conjunto;
{ C é o conjunto de candidatos }
begin
  S := ∅; { S contém conjunto solução }
  while (C <> ∅) and not solução(S) do
    begin
    x := seleciona(C); C := C − x;
    if viável(S + x) then S := S + x;
    end;
  if solução(S) then return(S)
                  else return('Não existe solução');
end;
```

- Inicialmente, o conjunto S de candidatos escolhidos está vazio.
- A cada passo, o melhor candidato restante ainda não tentado é considerado. O critério de escolha é ditado pela função de seleção.
- Se o conjunto aumentado de candidatos se torna inviável, o candidato é rejeitado. Senão, o candidato é adicionado ao conjunto S de candidatos escolhidos.
- A cada aumento de S verificamos se S constitui uma solução.

Características da Implementação de Algoritmos Gulosos

- Quando funciona corretamente, a primeira solução encontrada é sempre ótima.
- A função de seleção é geralmente relacionada com a função objetivo.
- Se o objetivo é:
  - maximizar $\Rightarrow$ provavelmente escolherá o candidato restante que proporcione o maior ganho individual.
  - minimizar $\Rightarrow$ então será escolhido o candidato restante de menor custo.
- O algoritmo nunca muda de idéia:
  - Uma vez que um candidato é escolhido e adicionado à solução ele lá permanece para sempre.
  - Uma vez que um candidato é excluído do conjunto solução, ele nunca mais é reconsiderado.

Algoritmos Aproximados

- Problemas que somente possuem algoritmos exponenciais para resolvê-los são considerados “difíceis”.
- Problemas considerados intratáveis ou difíceis são muito comuns.
- Exemplo: **problema do caixeiro viajante** cuja complexidade de tempo é $O(n!)$.
- Diante de um problema difícil é comum remover a exigência de que o algoritmo tenha sempre que obter a solução ótima.
- Neste caso procuramos por algoritmos eficientes que não garantem obter a solução ótima, mas uma que seja a mais próxima possível da solução ótima.

Tipos de Algoritmos Aproximados

- **Heurística**: é um algoritmo que pode produzir um bom resultado, ou até mesmo obter a solução ótima, mas pode também não produzir solução alguma ou uma solução que está distante da solução ótima.

- **Algoritmo aproximado**: é um algoritmo que gera **soluções aproximadas** dentro de um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado (comportamento monitorado sob o ponto de vista da qualidade dos resultados).