UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
Departamento de Ciências de Computação

SCC-501 - Capítulo 6 Paradigmas e Técnicas de Projetos de Algoritmos

João Luís Garcia Rosa[1]

[1] Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
http://www.icmc.usp.br/~joaoluis

2011

Sumário

Sumário

Apresentação

- O projeto de algoritmos requer abordagens adequadas:
  - A forma como um algoritmo aborda o problema pode levar a um desempenho ineficiente,
  - Em certo casos, o algoritmo pode não conseguir resolver o problema em tempo viável.
- Serão apresentados os principais paradigmas a serem seguidos durante o projeto de algoritmos, os quais levam a abordagens adequadas de projeto.

Apresentação

- Paradigmas de Projeto de Algoritmos [6]:
  - indução,
  - recursividade,
  - algoritmos tentativa e erro,
  - divisão e conquista,
  - programação dinâmica,
  - algoritmos gulosos,
  - algoritmos aproximados.

Apresentação

- Indução matemática e recursividade já foram estudados no início do curso, mas serão novamente considerados como paradigmas de projeto de algoritmos,
- Algoritmos **tentativa-e-erro** analisam todas as soluções possíveis do problema, sem utilizar nenhum critério para evitar a análise de certas soluções baseadas em outras já obtidas,
- O algoritmo implementa a idéia mais simples possível para se obter a solução do problema,
- Entretanto, essa abordagem é na maioria das vezes inviável.

Apresentação

- Três outras estratégias baseiam-se na idéia de decomposição de problemas complexos em outros mais simples, cujas soluções serão combinadas para fornecer uma solução para o problema original,
- As estratégias diferem na maneira de proceder:
  - **divisão e conquista** costuma fornecer algoritmos recursivos;
  - **programação dinâmica** e
  - **algoritmos gulosos** costumam levar a algoritmos iterativos.
- Algoritmos aproximados tentam resolver o problema quando a solução é considerada difícil (tempo exponencial).

Apresentação

- Infelizmente, não existe um paradigma que seja o melhor dentre todos!
- Um problema pode ser resolvido de maneira mais eficiente adotando-se determinado paradigma em detrimento de outro,
- Como será visto, um paradigma pode levar a um algoritmo $\mathcal{O}(2^n)$ e outro paradigma a um algoritmo $\mathcal{O}(n^3)$, para a resolução de um mesmo problema.

Apresentação

- Por exemplo:
  - Como ordenar um vetor de inteiros?
  - Como realizar o produto entre n matrizes de modo que o número de operações seja o menor possível?
  - Problema da mochila:
    - Considere n itens a serem levados para uma viagem, dentro de uma mochila de capacidade L que não pode comportar todos os itens,
    - Cada item tem um peso w_i e uma utilidade c_i. Quais itens escolher, que modo que a utilidade total dos itens levados seja a maior possível?

Sumário

Indução matemática [6]

- 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 tenha como parâmetro um número natural n,
- Para provar que T é válido para todos os valores de n, prova-se 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.

Indução matemática [6]

- A condição 1 é chamada de passo base,
- Provar a condição 2 é geralmente mais fácil que provar o teorema diretamente (pode-se 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.

Indução matemática [6]

- $S(n) = 1 + 2 + \cdots + n = n(n+1)/2$:
  - Para $n = 1$ a asserção é verdadeira, pois $S(1) = 1 = 1 \times (1+1)/2$ (passo base),
  - Assume-se que a soma dos primeiros n números naturais $S(n)$ é $n(n+1)/2$ (hipótese de indução),
  - Pela definição de $S(n)$ sabe-se 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 se quer provar.

Limite Superior de Equações de Recorrência [6]

- 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 se está interessado 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 $\mathcal{O}$, onde o lado direito da desigualdade representa o pior caso.

Indução Matemática para Resolver Equação de Recorrência [6]

- $T(2n) \leq 2T(n) + 2n - 1$, $T(2) = 1$, definida para valores de n que são potências de 2:
  - Procura-se $f(n)$ tal que $T(n) = \mathcal{O}(f(n))$, mas fazendo com que $f(n)$ seja o mais próximo possível da solução real para $T(n)$,
  - Considera-se o palpite $f(n) = n^2$,
  - Quer-se provar que $T(n) = \mathcal{O}(f(n))$ 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 $T(2n) \leq f(2n)$:

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

que é exatamente o que se quer provar. Logo, $T(n) = \mathcal{O}(n^2)$.

Indução Matemática para Resolver Equação de Recorrência [6]

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

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

- 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 [6]

- Vai-se então tentar $f(n) = n \log n$:
  - **Passo base**: $T(2) < 2 \log 2$,
  - **Passo de indução**: vai-se assumir que $T(n) \leq n \log n$.
- Quer-se 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*.

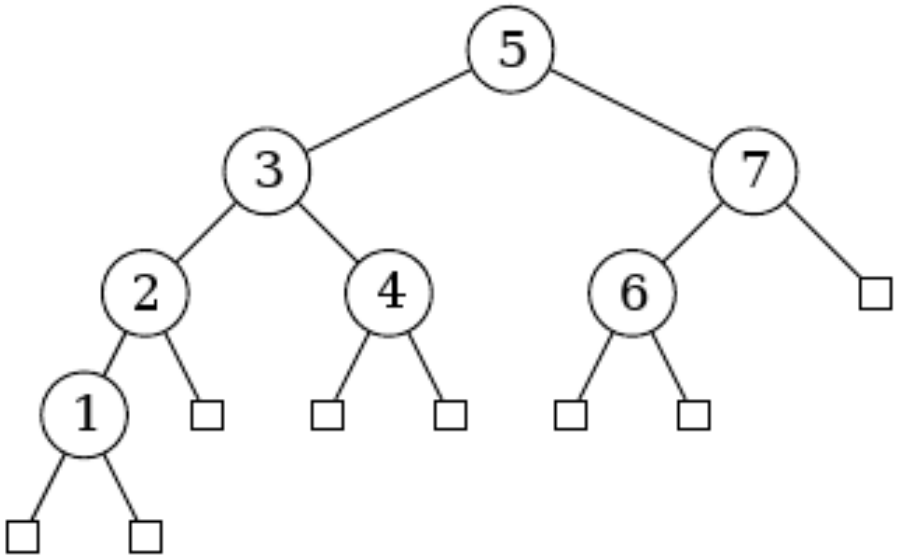
Sumário

Recursividade [6]

- 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,

Recursividade [6]

- 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.



Recursividade [6]

```
typedef long TipoChave;

typedef struct {
 TipoChave Chave;
 /* outros componentes */
} Registro;

typedef struct {
 Registro Reg;
 struct Nodo *Esq, *Dir;
} Nodo;
```

Recursividade [6]

- 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 vértices são visitados em ordem lexicográfica das chaves.

```
void Central(Nodo *p)
{
 if (p == NULL)
   return;
 Central(p -> Esq);
 printf("%d\n", p -> Reg.Chave);
 Central(p -> Dir);
}
```

Implementação de Recursividade [6]

- 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 $p = null$ o procedimento retorna para quem chamou utilizando o endereço de retorno que está no topo da pilha.

Problema de Terminação em Procedimentos Recursivos [6]

- 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:
  - $P \equiv \textit{if } B \textit{ then } \mathcal{C}[S_i, P]$.
- 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 [6]

- 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:
  - $P \equiv \mathit{if}\ n > 0\ \mathit{then}\ \mathcal{P}[S_i, P(n-1)]$.
- 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 [6]

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

Exemplo de Quando Não Usar Recursividade [6]

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

```
int FibRec(int n)
{
 if (n < 2)
   return n;
 else
   return FibRec(n-1) + FibRec(n-2);
}
```

Exemplos de Quando Não Usar Recursividade [6]

- Cálculo dos números de Fibonacci:
  - O programa é extremamente ineficiente porque recalcula o mesmo valor várias vezes,
  - Neste caso, a complexidade de espaço para calcular f_n é $\mathcal{O}(\Phi^n)$,
  - Considerando que a medida de complexidade de tempo $f(n)$ é o número de adições, e cada adição tem custo de $\mathcal{O}(1)$, então $f(n) = \mathcal{O}(\Phi^n)$.

Versão Iterativa do Cálculo de Fibonacci [6]

```
unsigned int FibIte (unsigned int n)
{
 unsigned int i = 1, k, F = 0;
 for (k = 1; k <= n; k++)
 {
   F += i;
   i = F - i;
 }
 return F;
}
```

- O programa tem complexidade de tempo $\mathcal{O}(n)$ e complexidade de espaço $\mathcal{O}(1)$,
- Deve-se evitar uso de recursividade quando existe solução óbvia por iteração.

Cálculo de Fibonacci [6]

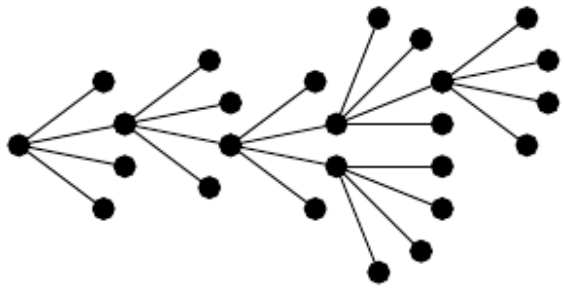
- Comparação versões recursiva e iterativa:

n	10	20	30	50	100
Recursiva	8 ms	1 s	2 min	21 dias	10^9 anos
Iterativa	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

Sumário

Algoritmos tentativa-e-erro

- Um algoritmo tentativa-e-erro é aquele que testa exaustivamente todas as soluções possíveis de um problema, de modo a obter a desejada,
- As soluções são testadas indiscriminadamente:
  - Não utiliza critérios para eliminar outras soluções que não poderão ser melhores que a obtida no estágio considerado.
- As soluções são enumeradas de modo semelhante ao percurso em uma árvore que possua todas as soluções,
- Muitas vezes a "árvore" de soluções cresce exponencialmente [6]!



Algoritmos tentativa-e-erro [6]

- 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.

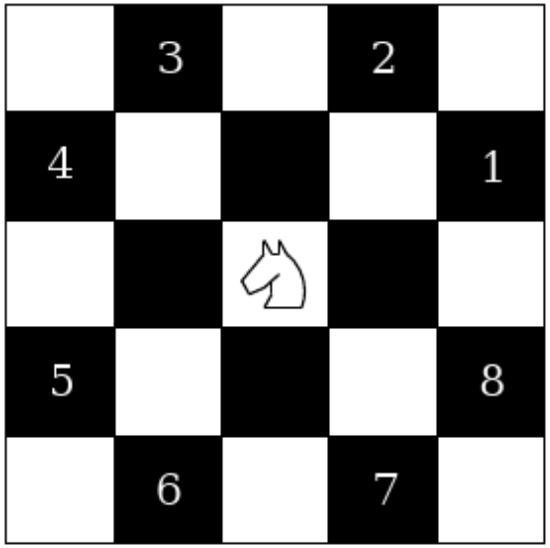
Exemplo: passeio do cavalo [6]

- 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:

```
void Tenta()
{ inicializa selecao de movimentos;
  do
  { seleciona proximo candidato ao movimento;
     if (aceitavel)
     { registra movimento;
        if (tabuleiro nao esta cheio)
        { tenta novo movimento;
           if (nao sucedido) apaga registro anterior;
        }
     }
  } while (!((mov.  bem sucedido) ou (acabaram-se cand.  movimento))); }
```

Exemplo: passeio do cavalo [6]

- 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:

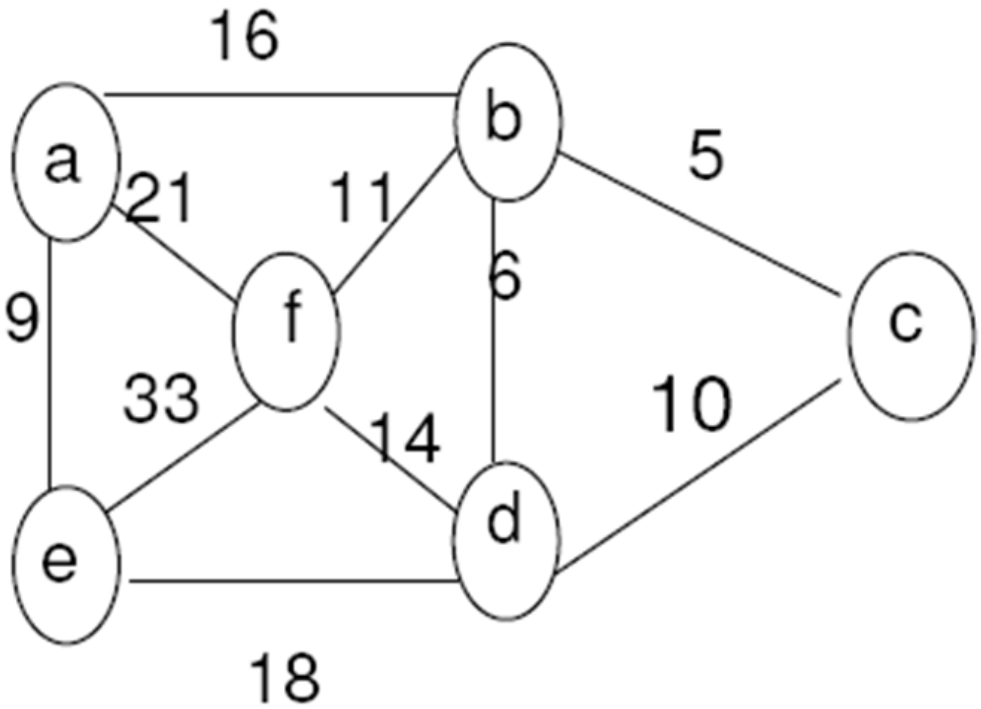


Exemplo: passeio do cavalo [6]

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

Algoritmos tentativa-e-erro

- Exercício:
  - Qual o menor caminho da cidade a até a c?



Algoritmos tentativa-e-erro

- Exercício:
  - TODOS os caminhos são enumerados:
    - $a \Rightarrow b \Rightarrow c$: 21
    - $a \Rightarrow b \Rightarrow d \Rightarrow c$: 32
    - $a \Rightarrow b \Rightarrow f \Rightarrow d \Rightarrow c$: 51
    - ...

Sumário

Algoritmos divisão-e-conquista

- O paradigma divisão-e-conquista consiste em:
  1. Dividir o problema a ser resolvido em subproblemas menores e independentes;
  2. Encontrar soluções para as partes;
  3. Combinar as soluções obtidas em uma solução global.
- Os algoritmos utilizam recursão para dividir e combinar.
- Processo recursivo :
  - dada uma entrada, se ela é suficientemente simples, obtém-se diretamente uma saída correspondente;
  - caso contrário, ela é decomposta em entradas mais simples, para as quais se aplica o mesmo processo, obtendo saídas correspondentes que são então combinadas em uma saída para a entrada original.

Algoritmos divisão-e-conquista [6]

- **Exemplo**: encontrar o maior e o menor elemento de um vetor de inteiros, $A[1..n]$, $n \geq 1$:

```
void MaxMin4 (int Linf, int Lsup, int *Max, int *Min)
{
  int Max1, Max2, Min1, Min2, Meio;
  if (Lsup - Linf <= 1)
  { if (A[Linf-1] < A[Lsup-1])
     { *Max = A[Lsup-1]; *Min = A[Linf-1]; }
     else { *Max = A[Linf-1]; *Min = A[Lsup-1]; } }
  else {
     Meio = (Linf+Lsup)/2;
     MaxMin4(Linf, Meio, &Max1, &Min1);
     MaxMin4(Meio+1, Lsup, &Max2, &Min2);
     if (Max1 > Max2) *Max = Max1; else *Max = Max2;
     if (Min1 < Min2) *Min = Min1; else *Min = Min2; }
}
```

- Cada chamada de *MaxMin4* atribui à *Max* e *Min* o maior e o menor elemento em $A[Linf]$, $A[Linf + 1]$, ..., $A[Lsup]$.

Divisão-e-conquista: análise do exemplo [6]

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

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

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

$$\begin{aligned} f(n) &= 2f(n/2) + 2 \\ 2f(n/2) &= 4f(n/4) + 2 \times 2 \\ 4f(n/4) &= 8f(n/8) + 2 \times 2 \times 2 \\ &\vdots \quad\quad \vdots \\ 2^{i-2}f(n/2^{i-2}) &= 2^{i-1}f(n/2^{i-1}) + 2^{i-1} \end{aligned}$$

Divisão-e-conquista: análise do exemplo [6]

- Adicionando lado a lado, obtém-se:

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

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

Algoritmos divisão-e-conquista

- Exemplos de algoritmos:
  - Busca binária;
  - *SelectionSort*, *MergeSorte*, *Quicksort*;
  - Maior elemento de uma sequência;
  - Fibonacci recursivo.

Sumário

Programação Dinâmica

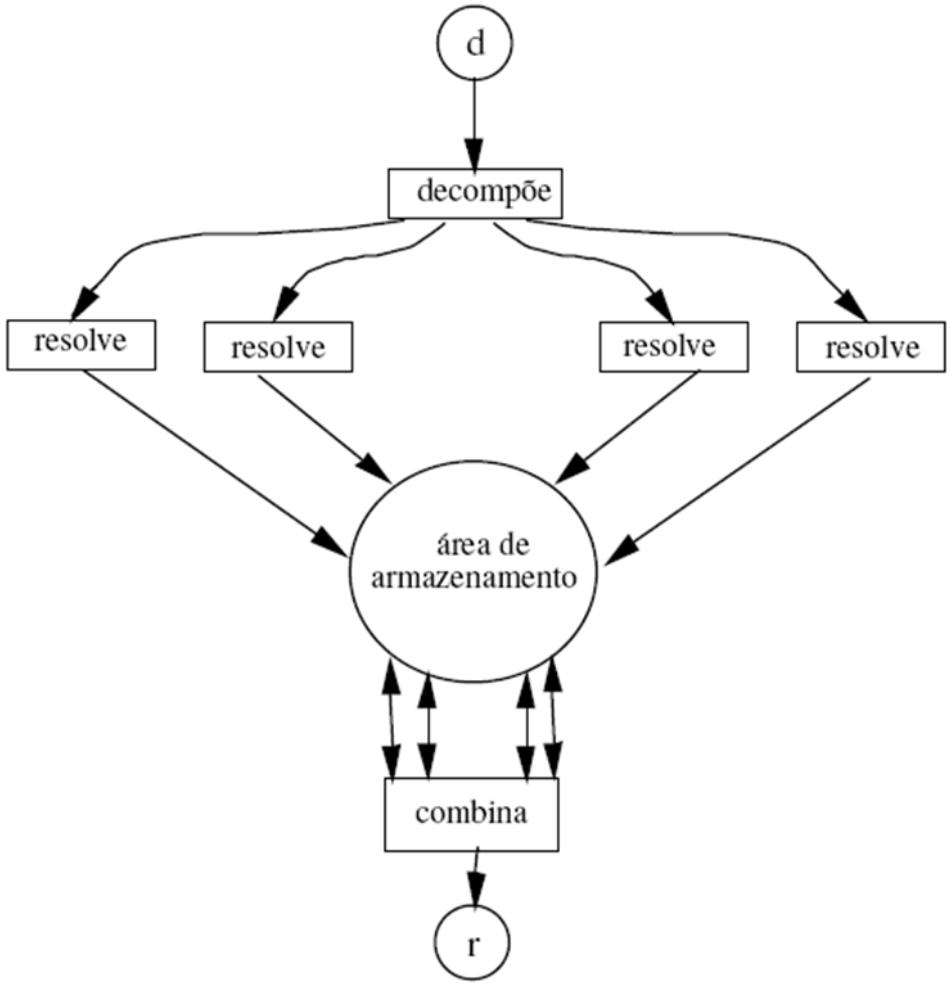
- Quando um algoritmo recursivo tem complexidade exponencial, a **programação dinâmica** pode levar a um algoritmo mais eficiente,
- A técnica de programação dinâmica consiste em dividir o problema original em subproblemas mais simples e resolvê-los, armazenando os resultados em uma tabela,
- Isso é feito iterativamente.
- A ideia básica da programação dinâmica é construir por etapas uma resposta ótima combinando respostas já obtidas para partes menores,
- Inicialmente, a entrada é decomposta em partes mínimas, para as quais são obtidas respostas.

Programação Dinâmica

- Em cada passo, sub-resultados são combinados obtendo-se respostas para partes maiores, até que se obtenha uma resposta para o problema original,
- A decomposição é feita uma única vez e os casos menores são tratados antes dos maiores,
- Suas soluções são armazenadas para serem usadas quantas vezes for necessário.

Programação Dinâmica

- Estrutura geral da programação dinâmica [4]:



Programação Dinâmica

- É baseada no princípio da otimalidade:
  - Em uma sequência ótima de escolhas ou de decisões, cada subsequência também deve ser ótima:
    - Por exemplo: o menor caminho de São Carlos a São Paulo passando por Campinas é dado pelo menor caminho de São Carlos a Campinas combinado com o menor caminho de Campinas a São Paulo.
- Reduz drasticamente o número total de verificações pois evita aquelas que sabidamente não podem ser ótimas:
  - A cada passo são eliminadas subsoluções que certamente não farão parte da solução ótima do problema.

Programação Dinâmica

- Exemplo: Qual a melhor maneira de se fazer o produto entre n matrizes?
  - O produto de uma matriz $p \times q$ por uma matriz $q \times r$ requer $\mathcal{O}(pqr)$ operações.
- Considere o produto:
  - $M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$:
    - O produto na ordem $M = M_1 \times (M_2 \times (M_3 \times M_4))$ requer 125.000 operações;
    - O produto na ordem $M = (M_1 \times (M_2 \times M_3)) \times M_4$ requer 2.200 operações.

Programação Dinâmica

- Resolver esse problema por tentativa e erro, isto é, testar todas as ordens possíveis de fazer o produto das matrizes para se obter qual é a melhor ($f(n)$), é um processo exponencial em n ($f(n) \geq 2^{n-2}$ [1]).
- Usando programação dinâmica é possível obter um algoritmo $\mathcal{O}(n^3)$!

Programação Dinâmica

- Seja m_{ij} o menor custo para calcular o produto $M_i \times M_{i+1} ... \times M_j$ para $1 \leq i \leq j \leq n$,
- Assim,

$$m_{ij} = \begin{cases} 0 & se\ i = j \\ Min_{i \leq k \leq j}(m_{ik} + m_{k+1,j} + b_{i-1} b_k b_j) & se\ j > i \end{cases}$$

  - onde:
    - o termo m_{ik} representa o custo mínimo para calcular $M' = M_i \times M_{i+1} \times ... \times M_k$,
    - o segundo termo $m_{k+i,j}$ representa o custo mínimo para calcular $M'' = M_{k+1} \times M_{k+2} \times ... \times M_j$,
    - o terceiro termo, $b_{i-1} b_k b_j$, representa o custo de multiplicar $M'[b_{i-1}, b_k]$ por $M''[b_k, b_j]$,
    - $b_{i-1} \times b_i$ são as dimensões da matriz M_i,
    - a equação acima diz que $m_{ij}, j > i$, representa o custo mínimo de todos os valores possíveis de k entre i e $j - 1$, da soma dos três termos.

Programação Dinâmica

- Seja m_{ij} o menor custo para calcular o produto $M_i \times M_{i+1} ... \times M_j$ para $1 \leq i \leq j \leq n$,
- Assim,

$$m_{ij} = \begin{cases} 0 & se\ i = j \\ Min_{i \leq k \leq j}(m_{ik} + m_{k+1,j} + b_{i-1} b_k b_j) & se\ j > i \end{cases}$$

$$\underbrace{Min_{i \leq k \leq j}}$$

principio da otimalidade

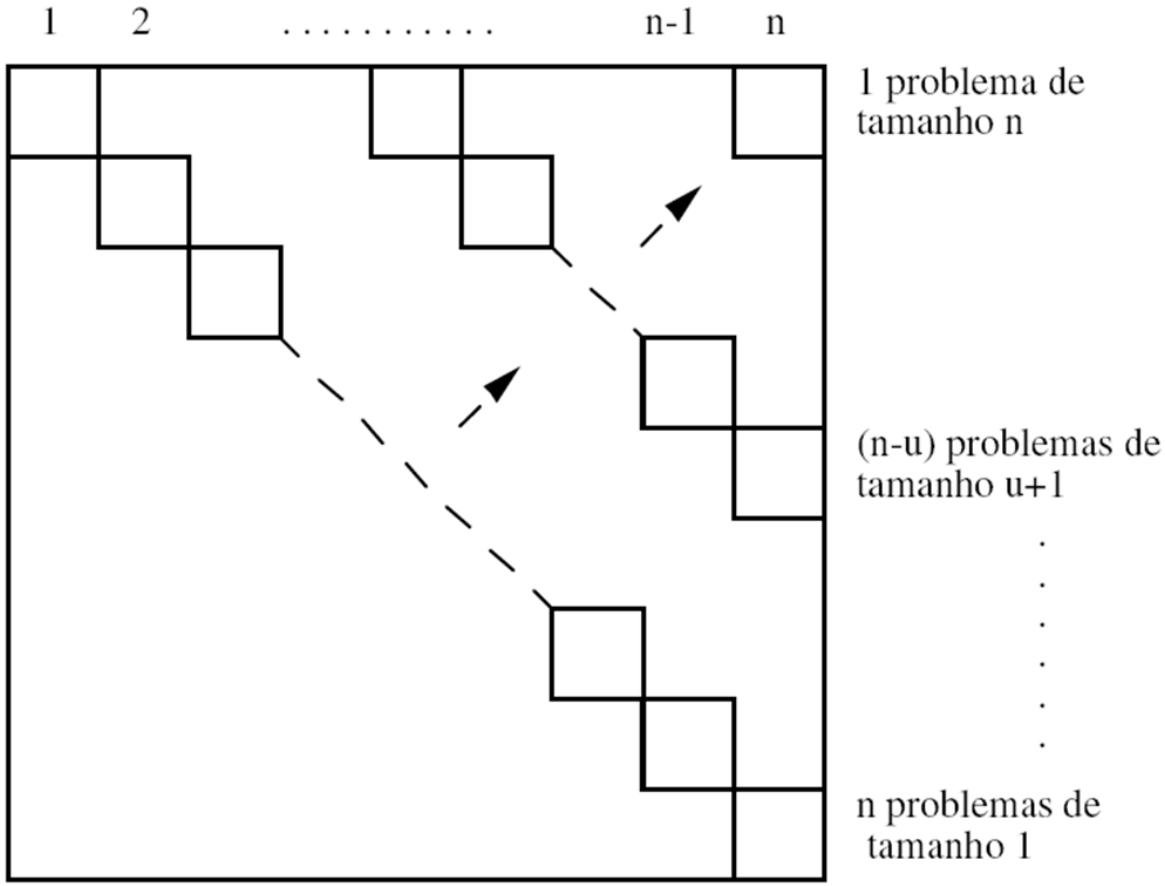
Em uma sequência ótima de escolhas ou de decisões, cada subsequência também deve ser ótima.

Programação Dinâmica

- Os valores m_{ij} são calculados na ordem crescente das diferenças nos subscritos,
- O cálculo inicia com m_{ii} para todo i, depois $m_{i,i+1}$ para todo i, depois $m_{i,i+2}$, ...
- Assim, esse método é chamado ascendente, ao contrário dos métodos recursivos, que são chamados descendentes.

Programação Dinâmica

- Hierarquia de instâncias na programação dinâmica [4]:



Programação Dinâmica

- **Exemplo**: Tabela de dados calculados pela programação dinâmica para o produto:
  - $M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$

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

- **Exercício**: Escrever um algoritmo para achar o produto de menor custo entre n matrizes, usando programação dinâmica.

Programação Dinâmica

```
#define Maxn 10
int main(int argc, char *argv[])
{
  int i, j, k, h, n, temp;
  int b[Maxn+1];
  int m[Maxn][Maxn];
  printf("Numero de matrizes n:  ");
  scanf("%d", &n);
  getchar();
  printf("Dimensoes das matrizes:  ");
  for (i = 0; i <= n; i++)
     scanf("%d", &b[i]);
  for (i = 0; i < n; i++)
     m[i][i] = 0;
  for (h = 1; h <= n-1; h++) {
     for (i = 1; i <= n-h; i++) {
        j = i+h;
        m[i-1][j-1] = INT_MAX;
        for (k = i; k <= j-1; k++)
        {
           temp = m[i-1][k-1] + m[k][j-1] + b[i-1] * b[k] * b[j];
           if (temp < m[i-1][j-1])
              m[i-1][j-1] = temp;
        }
        printf("m[%d][%d] = %d\n", i-1, j-1, m[i-1][j-1]);
     }
     putchar('\n');
  }
  return 0;
}
```

Sumário

Algoritmos gulosos

- São tipicamente usados para resolver problemas de otimização,
- Por exemplo, o algoritmo para encontrar o caminho mais curto entre duas cidades:
  - Um algoritmo guloso escolhe a estrada que parece mais promissora no instante atual e nunca muda essa decisão, independentemente do que possa acontecer depois.
- A cada iteração:
  - seleciona um elemento conforme uma função gulosa,
  - marca-o para não considerá-lo novamente nos próximos estágios,
  - atualiza a entrada,
  - examina o elemento selecionado quanto sua viabilidade,
  - decide a sua participação ou não na solução.

Algoritmos gulosos

- Algoritmo guloso genérico:
  - C: conjunto de candidatos;
  - S: conjunto solução.
  - $S = \emptyset$
  - Enquanto ($C \neq \emptyset$) e (S não tem solução)
    - x = seleciona(C)
    - $C = C - \{x\}$
    - Se ($S + \{x\}$ é viável) então $S = S + \{x\}$
  - Se (S tem solução) então retorna S
  - Senão não existe solução

Características dos Algoritmos Gulosos [6]

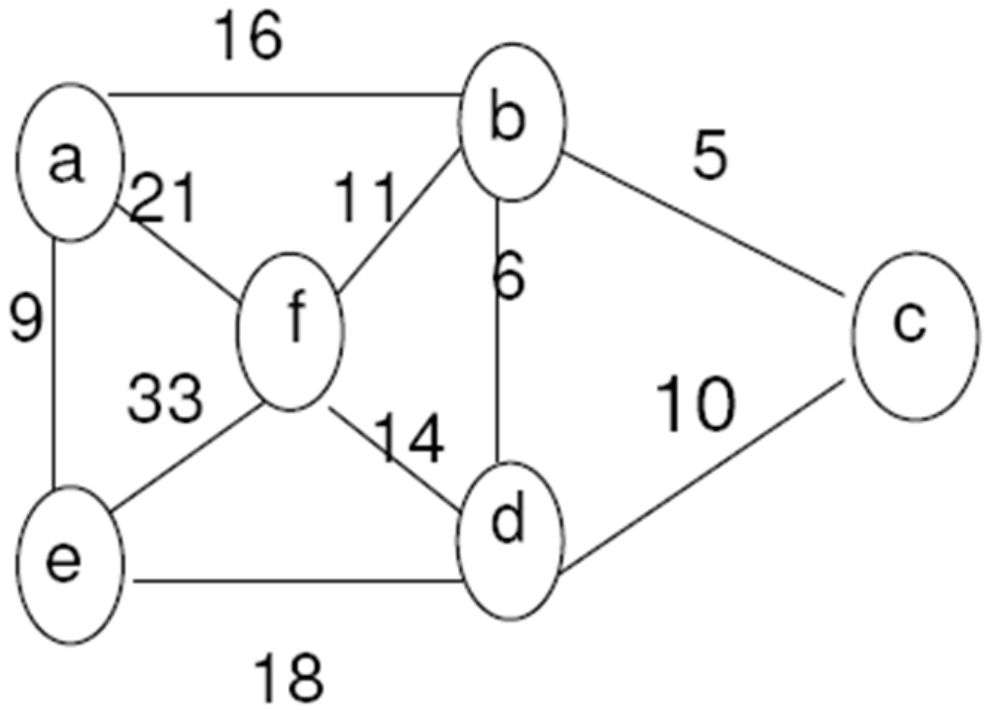
- Para construir a solução ótima existe um conjunto ou lista de candidatos,
- São acumulados um conjunto de candidatos considerados e escolhidos, e o outro de candidatos considerados e rejeitados,
- Existe uma função que verifica se um conjunto particular de candidatos produz uma solução (sem considerar otimalidade no momento),
- Outra função verifica se um conjunto de candidatos é viável (também sem se 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 explícita no algoritmo guloso).

Características da Implementação de Algoritmos Gulosos [6]

- 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 ideia:
  - 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 gulosos

- Exercício:
  - Calcule o menor caminho da cidade *a* até a *c*, utilizando um algoritmo guloso.



Sumário

Algoritmos Aproximados [6]

- 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 $\mathcal{O}(n!)$.
- Diante de um problema difícil é comum remover a exigência de que o algoritmo tenha sempre que obter a solução ótima.
- Neste caso procura-se por algoritmos eficientes que não garantem obter a solução ótima, mas uma que seja a mais próxima possível da solução ótima.

Algoritmos Aproximados [6]

- 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).

O Problema da Mochila

- Lembra do problema da mochila?
  - Problema da mochila:
    - Considere n itens a serem levados para uma viagem, dentro de uma mochila de capacidade L que não pode comportar todos os itens,
    - Cada item tem um peso w_i e uma utilidade c_i. Quais itens escolher, que modo que a utilidade total dos itens levados seja a maior possível?
- Qual paradigma utilizar para desenvolver um algoritmo que resolva esse problema?

Bibliografia I

[1] Aho, A. V., Hopcroft. J. E., Ullman, J. D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Algoritmos - Teoria e Prática.
Ed. Campus, Rio de Janeiro, Segunda Edição, 2002.

[3] Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.

Bibliografia II

[4] Munari Junior, Pedro Augusto
Paradigmas e Técnicas de Projeto de Algoritmos.
SCE-181 Introdução à Ciência da Computação II.
Slides. Ciência de Computação. ICMC/USP, 2007.

[5] Toscani, Laira Vieira; Veloso, Paulo A. S.
Complexidade de algoritmos.
Séries livros didáticos, no. 13. 2002.

[6] Ziviani, Nivio
Projeto de Algoritmos - com implementações em Java e C++.
Thomson, 2007.