

**Pontifícia Universidade Católica de Minas Gerais
Instituto de Ciências Exatas e Informática
Departamento de Ciência da Computação
Curso de Ciência da Computação**

Projeto e Análise de Algoritmos

Parte 2

**Raquel Mini
raquelmini@pucminas.br**

**Força Bruta
(Busca Exaustiva ou
Enumeração Total)**

Força Bruta

- Consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema
- Geralmente possui uma implementação simples e sempre encontrara uma solução se ela existir
- O seu custo computacional é proporcional ao número de candidatos a solução que, em problemas reais, tende a crescer exponencialmente
- É tipicamente usada em problemas cujo tamanho é limitado ou quando não se conhece um algoritmo mais eficiente
- Também pode ser usado quando a simplicidade da implementação é mais importante que a velocidade de execução, como nos casos de aplicações críticas em que os erros de algoritmo possuem sérias consequências

Força Bruta – Clique

- Considere um conjunto P de n pessoas e uma matriz M de tamanho $n \times n$, tal que $M[i][j] = M[j][i] = 1$, se as pessoas i e j se conhecem e $M[i][j] = M[j][i] = 0$, caso contrário
- Problema: existe um subconjunto C (Clique), de r pessoas escolhidas de P , tal que qualquer par de pessoas de C se conhecem?
- Solução usando força bruta: verificar, para todas as combinações simples (sem repetições) C de r pessoas escolhidas entre as n pessoas do conjunto P , se todos os pares de pessoas de C se conhecem

Força Bruta – Clique

- Considere um conjunto P de 8 pessoas representado pela matriz abaixo (de tamanho 8×8):

x	1	2	3	4	5	6	7	8
1	1	0	1	1	1	1	1	0
2	0	1	0	0	1	0	0	1
3	1	0	1	1	0	1	1	1
4	1	0	1	1	1	1	1	1
5	1	1	0	1	1	0	0	0
6	1	0	1	1	0	1	1	1
7	1	0	1	1	0	1	1	0
8	0	1	1	1	0	1	0	1

- Existe um conjunto C de 5 pessoas escolhidas de P tal que qualquer par de pessoas de C se conhecem?

Força Bruta – Clique

- Existem 56 combinações simples de 5 elementos escolhidos dentre um conjunto de 8 elementos:

1 2 3 4 5	1 2 4 6 8	1 3 5 7 8	2 3 5 6 8
1 2 3 4 6	1 2 4 7 8	1 3 6 7 8	2 3 5 7 8
1 2 3 4 7	1 2 5 6 7	1 4 5 6 7	2 3 6 7 8
1 2 3 4 8	1 2 5 6 8	1 4 5 6 8	2 4 5 6 7
1 2 3 5 6	1 2 5 7 8	1 4 5 7 8	2 4 5 6 8
1 2 3 5 7	1 2 6 7 8	1 4 6 7 8	2 4 5 7 8
1 2 3 5 8	1 3 4 5 6	1 5 6 7 8	2 4 6 7 8
1 2 3 6 7	1 3 4 5 7	2 3 4 5 6	2 5 6 7 8
1 2 3 6 8	1 3 4 5 8	2 3 4 5 7	3 4 5 6 7
1 2 3 7 8	1 3 4 6 7	2 3 4 5 8	3 4 5 6 8
1 2 4 5 6	1 3 4 6 8	2 3 4 6 7	3 4 5 7 8
1 2 4 5 7	1 3 4 7 8	2 3 4 6 8	3 4 6 7 8
1 2 4 5 8	1 3 5 6 7	2 3 4 7 8	3 5 6 7 8
1 2 4 6 7	1 3 5 6 8	2 3 5 6 7	4 5 6 7 8

Força Bruta – Clique

- Todos os pares de pessoas do subconjunto $C = \{1, 3, 4, 6, 7\}$ se conhecem:

x	1	3	4	6	7
1	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	1
6	1	1	1	1	1
7	1	1	1	1	1

- Como enumerar todas as combinações simples de r elementos de um conjunto de tamanho n ?

Força Bruta – Combinação

```
#include<iostream>
using namespace std;

void combinacao(int n, int r, int x[], int next, int k){
    int i;
    if (k == r){
        for (i = 0; i < r; i++){
            cout<<x[i]+1<<" ";
        }
        cout<<endl;
    } else {
        for (i = next; i < n; i++) {
            x[k] = i;
            combinacao(n,r,x,i+1,k+1);
        }
    }
}

int main () {
    int n, r, x[100];
    cout<<"Entre com o valor de n: ";
    cin>>n;
    cout<<"Entre com o valor de r: ";
    cin>>r;
    combinacao(n,r,x,0,0);
    return 0;
}
```


Força Bruta – Ciclo Hamiltoniano

- Considere um conjunto de n cidades e uma matriz M de tamanho $n \times n$ tal que $M[i][j] = 1$, se existir um caminho direto entre as cidades i e j , e $M[i][j] = 0$, caso contrário
- Problema: existe uma forma de, saindo de uma cidade qualquer, visitar todas as demais cidades, sem passar duas vezes por nenhuma cidade e, no final, retornar para a cidade inicial?
- Se existe uma forma de sair de uma cidade x qualquer, visitar todas as demais cidades (sem repetir nenhuma) e depois retornar para x , então existe um ciclo Hamiltoniano e qualquer cidade do ciclo pode ser usada como ponto de partida

Força Bruta – Ciclo Hamiltoniano

- Como vimos, qualquer cidade pode ser escolhida como cidade inicial. Sendo assim, vamos escolher, arbitrariamente a cidade n como ponto de partida
- Solução usando força bruta: testar todas as permutações das $n-1$ primeiras cidades, verificando se existe um caminho direto entre a cidade n e a primeira da permutação, assim como um caminho entre todas as cidades consecutivas da permutação e, por fim, um caminho direto entre a última cidade da permutação e a cidade n
- Ciclo Hamiltoniano: $n \rightsquigarrow [p_1 \rightsquigarrow p_2 \rightsquigarrow p_3 \rightsquigarrow \dots \rightsquigarrow p_{n-1}] \rightsquigarrow n$

Força Bruta – Ciclo Hamiltoniano

- Considere um conjunto de 8 cidades representado pela matriz abaixo:

x	1	2	3	4	5	6	7	8
1	0	0	1	0	1	1	1	0
2	0	1	0	0	1	0	0	1
3	1	0	1	1	0	1	1	0
4	0	0	1	1	0	0	1	0
5	1	1	0	1	1	0	0	0
6	0	0	1	1	0	0	1	1
7	1	0	0	1	0	1	1	1
8	0	1	1	1	0	1	0	1

- Existe uma forma de, a partir da cidade 8, visitar todas as demais cidades, sem repetir nenhuma e, ao final, retornar para a cidade 8?

Força Bruta – Ciclo Hamiltoniano

- Existem 5040 permutações das 7 primeiras cidades da lista original:

1 2 3 4 5 6 7	...	7 6 5 2 3 4 1
1 2 3 4 5 7 6	3 6 4 5 1 7 2	7 6 5 2 4 1 3
1 2 3 4 6 5 7	3 6 4 5 2 1 7	7 6 5 2 4 3 1
1 2 3 4 6 7 5	3 6 4 5 2 7 1	7 6 5 3 1 2 4
1 2 3 4 7 5 6	3 6 4 5 7 1 2	7 6 5 3 1 4 2
1 2 3 4 7 6 5	3 6 4 5 7 2 1	7 6 5 3 2 1 4
1 2 3 5 4 6 7	3 6 4 7 1 2 5	7 6 5 3 2 4 1
1 2 3 5 4 7 6	3 6 4 7 1 5 2	7 6 5 3 4 1 2
1 2 3 5 6 4 7	3 6 4 7 2 1 5	7 6 5 3 4 2 1
1 2 3 5 6 7 4	3 6 4 7 2 5 1	7 6 5 4 1 2 3
1 2 3 5 7 4 6	3 6 4 7 5 1 2	7 6 5 4 1 3 2
1 2 3 5 7 6 4	3 6 4 7 5 2 1	7 6 5 4 2 1 3
1 2 3 6 4 5 7	3 6 5 1 2 4 7	7 6 5 4 2 3 1
1 2 3 6 4 7 5	3 6 5 1 2 7 4	7 6 5 4 3 1 2
1 2 3 6 5 4 7	...	7 6 5 4 3 2 1

- Como enumerar todas as permutações de n valores distintos?

Força Bruta – Permutação

```
#include<iostream>
using namespace std;

void permutacao(int n, int x[], bool used[], int k){
    int i;
    if (k == n){
        for (i = 0; i < n; i++){
            cout<<x[i]+1<<" ";
        }
        cout<<endl;
    } else {
        for (i = 0; i < n; i++) {
            if (!used[i]) {
                used[i] = true;
                x[k] = i;
                permutacao(n, x, used, k+1);
                used[i] = false;
            }
        }
    }
}

int main () {
    int i, n, x[100];
    bool used[100];
    cout<<"Entre com o valor de n: ";
    cin>>n;
    for (i = 0; i < n; i++){
        used[i] = false;
    }
    permutacao(n,x,used,0);
    return 0;
}
```

Incremental



Incremental

- A ordenação por inserção utiliza uma abordagem incremental: tendo ordenado o subarranjo $A[1 \dots j-1]$, inserimos o elemento isolado $A[j]$ em seu lugar apropriado, formando o subarranjo ordenado $A[1 \dots j]$.



Incremental

```
INSERTION-SORT(A)
for j ← 2 to n do
    chave ← A[j]
    i ← j - 1
    A[0] ← chave    //sentinela
    while A[i] > chave do
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← chave
```


Exercícios

1. Implemente um algoritmo que enumere todos os arranjos de tamanho r dentro um conjunto de n elementos.

Algoritmos Tentativa e Erros (*Backtracking*)

Ziviani – págs. 44 até 48

Backtracking

- Algoritmo para encontrar todas (ou algumas) soluções de um problema computacional, que incrementalmente constrói candidatas de soluções e abandona uma candidata parcialmente construída tão logo quanto for possível determinar que ela não pode gerar uma solução válida
- Pode ser aplicado para problemas que admitem o conceito de “solução candidata parcial” e que exista um teste relativamente rápido para verificar se uma candidata parcial pode ser completada como uma solução válida

Backtracking

- Quando aplicável, *backtracking* é frequentemente muito mais rápido que algoritmos de força bruta, já que ele pode eliminar um grande número de soluções inválidas com um único teste
- Enquanto algoritmos de força bruta geram todas as possíveis soluções e só depois verificam se elas são válidas, *backtracking* só gera soluções válidas

Backtracking – Passeio do Cavalo

- Tabuleiro com $n \times n$ posições: cavalo se movimenta segundo regras do xadrez
- Problema: partindo da posição (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

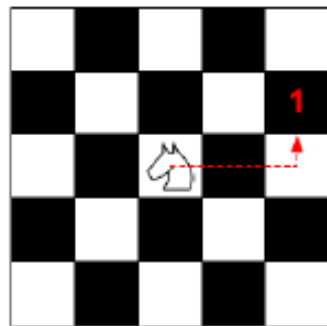
```
TENTA
1  Inicializa seleção de movimentos
2  repeat
3      Seleciona próximo candidato ao movimento
4      if aceitável
5          then Registra movimento
6              if tabuleiro não está cheio
7                  then Tenta novo movimento
8                      if não é bem sucedido
9                          then Apaga registro anterior
10 until (movimento bem sucedido) ∨ (acabaram-se candidatos ao movimento)
```

Backtracking – Passeio do Cavalo

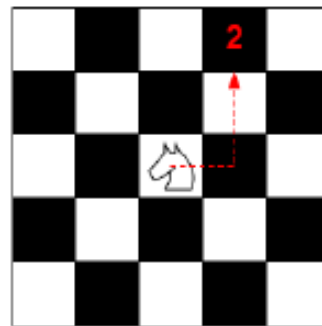
- O tabuleiro pode ser representado por uma matriz $n \times n$
- A situação de cada posição pode ser representada por um inteiro para recordar o histórico das ocupações:
 - $t[x][y] = 0$, campo $\langle x, y \rangle$ não visitado
 - $t[x][y] = i$, campo $\langle x, y \rangle$ visitado no i -ésimo movimento,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

Backtracking – Passeio do Cavalo

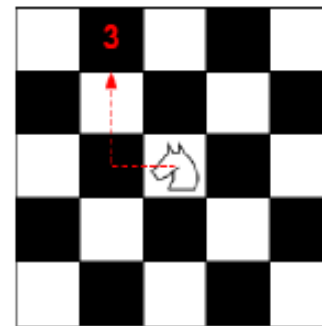
- Regras do xadrez para os movimentos do cavalo:



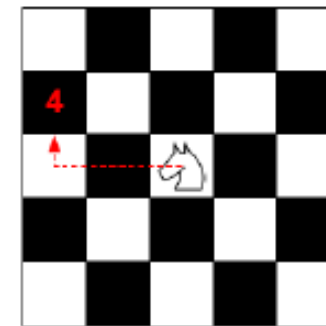
2Dir e 1Cima



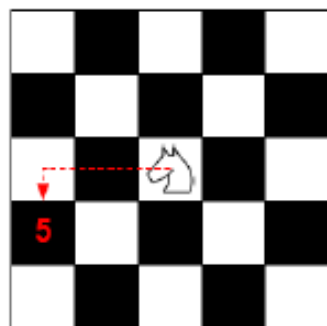
1Dir e 2Cima



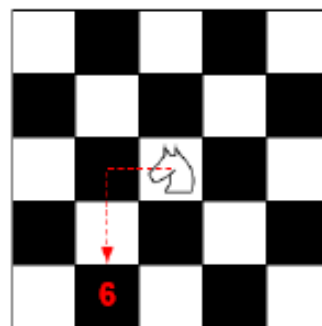
1Esq e 2Cima



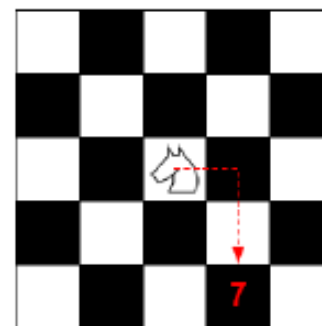
2Esq e 1Cima



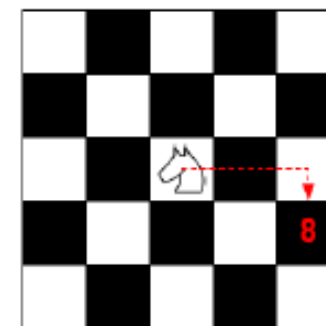
2Esq e 1Baixo



1Esq e 2Baixo



1Dir e 2Baixo



2Dir e 1Baixo

Backtracking – Passeio do Cavalo

PASSEIODOCAVALO(n)

▷ Parâmetro: n (tamanho do lado do tabuleiro)

▷ Variáveis auxiliares:

i, j

$t[1..n, 1..n]$

q

s

$h[1..8], v[1..8]$

1 $s \leftarrow \{1, 2, 3, 4, 5, 6, 7, 8\}$

2 $h[1..8] \leftarrow [2, 1, -1, -2, -2, -1, 1, 2]$

3 $v[1..8] \leftarrow [1, 2, 2, 1, -1, -2, -2, -1]$

4 **for** $i \leftarrow 1$ **to** n

5 **do for** $j \leftarrow 1$ **to** n

6 **do** $t[i, j] \leftarrow 0$

7 $t[1, 1] \leftarrow 1$

8 TENTA(2, 1, 1, q)

9 **if** q

10 **then print** Solução

11 **else print** Não há solução

▷ Contadores

▷ Tabuleiro de $n \times n$

▷ Indica se achou uma solução

▷ Movimentos identificados por um n°

▷ Existem oito movimentos possíveis

▷ Conjunto de movimentos

▷ Movimentos na horizontal

▷ Movimentos na vertical

▷ Inicializa tabuleiro

▷ Escolhe uma casa inicial do tabuleiro

▷ Tenta o passeio usando *backtracking*

▷ Achou uma solução?

Backtracking – Passeio do Cavalo

TENTA(i, x, y, q)

▷ Parâmetros: i (i -ésima casa); x, y (posição no tabuleiro); q (achou solução?)

▷ Variáveis auxiliares: $xn, yn, m, q1$

```
1  $m \leftarrow 0$ 
2 repeat
3    $m \leftarrow m + 1$ 
4    $q1 \leftarrow \text{false}$ 
5    $xn \leftarrow x + h[m]$ 
6    $yn \leftarrow y + v[m]$ 
7   if  $(xn \in s) \wedge (yn \in s)$ 
8     then if  $t[xn, yn] = 0$ 
9       then  $t[xn, yn] \leftarrow i$ 
10        if  $i < n^2$ 
11          then TENTA( $i + 1, xn, yn, q1$ )
12          if  $\neg q1$ 
13            then  $t[xn, yn] \leftarrow 0$ 
14          else  $q1 \leftarrow \text{true}$ 
15 until  $q1 \vee (m = 8)$ 
16  $q \leftarrow q1$ 
```

Backtracking – Passeio do Cavalo

- Resultado do Passeio do Cavalo em um tabuleiro 8 x 8

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

Backtracking – Labirinto

- Dado um labirinto representado por uma matriz de tamanho $n \times m$, uma posição inicial $p_i = (x_i; y_i)$ e uma posição final $p_f = (x_f; y_f)$, tal que $p_i \neq p_f$, determinar se existe um caminho entre p_i e p_f
- Podemos representar o labirinto como uma matriz M tal que:

$$M[x, y] = \begin{cases} -2, & \text{se a posição } (x, y) \text{ representa uma parede} \\ -1, & \text{se a posição } (x, y) \text{ não pertence ao caminho} \\ i, & \text{tal que } i \geq 0, \text{ se a posição } (x, y) \text{ pertence ao caminho} \end{cases}$$

- Neste caso, vamos supor que o labirinto é cercado por paredes, eventualmente apenas com exceção do local designado como saída

Backtracking – Labirinto

- A figura abaixo mostra um labirinto de tamanho 8 x 8

X	X	X	X	X	X	X	X
X	•						X
X	X		X				X
X			X	X	X		X
X		X	X				X
X		X				X	X
X				X			X
X	X	X	X	X	X	o	X

X: parede/obstáculo

•: posição inicial

o: posição final (saída do labirinto)

Backtracking – Labirinto

- Caminho encontrado usando a seguinte ordem de busca:
 - para esquerda
 - para baixo
 - para direita
 - para cima

X	X	X	X	X	X	X	X
X	00	01					X
X	X	02	X				X
X	04	03	X	X	X		X
X	05	X	X				X
X	06	X	10	11	12	X	X
X	07	08	09	X	13	14	X
X	X	X	X	X	X	15	X

Backtracking – Labirinto

- Caminho encontrado usando a seguinte ordem de busca:
 - para direita
 - para baixo
 - para esquerda
 - para cima

X	X	X	X	X	X	X	X
X	00	01	02	03	04	05	X
X	X		X			06	X
X			X	X	X	07	X
X		X	X		09	08	X
X		X			10	X	X
X				X	11	12	X
X	X	X	X	X	X	13	X

Backtracking – Labirinto

```
#include<iostream>
#include <iomanip>
#define MAX 10
using namespace std;
void imprimeLabirinto(int M[MAX][MAX], int n, int m) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            if (M[i][j] == -2) cout<<" XX";
            if (M[i][j] == -1) cout<<" ";
            if (M[i][j] >= 0) cout<<" "<<setw(2)<<M[i][j];
        }
        cout<<"\n";
    }
}
void obtemLabirinto(int M[MAX][MAX], int &n, int &m, int &Li, int &Ci, int &Lf, int &Cf) {
    int i, j, d;
    cin>>n; cin>>m; /* dimensoes do labirinto */
    cin>>Li; cin>>Ci; /* coordenadas da posicao inicial */
    cin>>Lf; cin>>Cf; /* coordeandas da posicao final (saida) */
    /* labirinto: 1 = parede ou obstaculo 0 = posicao livre */
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++) {
            cin>>d;
            if (d == 1)
                M[i][j] = -2;
            else
                M[i][j] = -1;
        }
}
```

Backtracking – Labirinto

```
int labirinto(int M[MAX][MAX], int deltaL[], int deltaC[], int Li, int Ci, int Lf, int Cf) {
    int L, C, k, passos;
    if ((Li == Lf) && (Ci == Cf)) return M[Li][Ci];
    /* testa todos os movimentos a partir da posicao atual */
    for (k = 0; k < 4; k++) {
        L = Li + deltaL[k];
        C = Ci + deltaC[k];
        /* verifica se o movimento eh valido e gera uma solucao factivel */
        if (M[L][C] == -1) {
            M[L][C] = M[Li][Ci] + 1;
            passos = labirinto(M, deltaL, deltaC, L, C, Lf, Cf);
            if (passos > 0) return passos;
        }
    }
    return 0;
}

int main() {
    int M[MAX][MAX], resposta, n, m, Li, Ci, Lf, Cf;
    // define os movimentos validos no labirinto
    int deltaL[4] = { 0, +1, 0, -1};
    int deltaC[4] = {+1, 0, -1, 0};
    // obtem as informacoes do labirinto
    obtemLabirinto(M, n, m, Li, Ci, Lf, Cf);
    M[Li - 1][Ci - 1] = 0; /* define a posicao inicial no tabuleiro */
    /* tenta encontrar um caminho no labirinto */
    resposta = labirinto(M, deltaL, deltaC, Li - 1, Ci - 1, Lf - 1, Cf - 1);
    if (resposta == 0)
        cout<<"Nao existe solucao.\n";
    else {
        cout<<"Existe uma solucao em "<<resposta<<" passos.\n";
        imprimeLabirinto(M, n, m);
    }
    return 0;
}
```


Backtracking – Labirinto

- Exemplo de entrada:

```
8 8
2 2
8 7
1 1 1 1 1 1 1 1
1 0 1 0 0 0 0 1
1 0 0 0 1 0 0 1
1 0 0 1 1 0 0 1
1 0 1 1 0 0 0 1
1 0 0 1 1 1 1 1
1 0 0 0 0 0 0 1
1 1 1 1 1 1 0 1
```

Backtracking – Labirinto

- Exemplo de saída:

```
Existe uma solucao em 13 passos .
XX XX XX XX XX XX XX
XX 0 XX 4 5 6 7 XX
XX 1 2 3 XX 13 8 XX
XX 4 3 XX XX 12 9 XX
XX 5 XX XX 12 11 10 XX
XX 6 7 XX XX XX XX XX
XX 8 9 10 11 12 XX
XX XX XX XX XX XX 13 XX
```

Backtracking – Problemas de Otimização

- Muitas vezes não estamos apenas interessados em encontrar uma solução qualquer, mas em encontrar uma solução ótima (segundo algum critério de otimalidade pré-estabelecido)
- Por exemplo, no problema do labirinto, ao invés de determinar se existe um caminho entre o ponto inicial e o final (saída), podemos estar interessados em encontrar uma solução que usa o menor número possível de passos

Branch and Bound



Branch and Bound

- Refere-se a um tipo de algoritmo usado para encontrar soluções ótimas para vários problemas de otimização
- Problemas de otimização podem ser tanto de maximização quando de minimização
- Consiste em uma enumeração sistemática de todos os candidatos à solução, com eliminação de uma candidata parcial quando uma dessas duas situações for detectada (considerando um problema de minimização):
 - A candidata parcial é incapaz de gerar uma solução válida (teste similar ao realizado pelo método *backtracking*)
 - A candidata parcial é incapaz de gerar uma solução ótima, considerando o valor da melhor solução encontrada até então (limitante superior) e o custo ainda necessário para gerar uma solução a partir da solução candidata atual (limitante inferior)

Branch and Bound

- O desempenho de um programa *branch and bound* está fortemente relacionado à qualidade dos seus limitantes inferiores e superiores
 - Quando mais precisos forem esses limitantes, menos soluções parciais serão consideradas e mais rápido o programa encontrará a solução ótima
 - O nome *branch and bound* refere-se às duas fases do algoritmo:
 - Branch*: testar todas as ramificações de uma solução candidata parcial
 - Bound*: limitar a busca por soluções sempre que detectar que o atual ramo da busca é infrutífero

Branch and Bound – Labirinto

- Podemos alterar o programa visto anteriormente para encontrar um caminho ótimo num labirinto usando a técnica *branch and bound*
- Podemos inicialmente notar que se um caminho parcial já usou tantos passos quanto o melhor caminho completo previamente descoberto, então este caminho parcial pode ser descartado
- Mais do que isso, se o número de passos do caminho parcial mais o número de passos mínimos necessários entre a posição atual e a saída (desconsiderando eventuais obstáculos) for maior ou igual ao número de passos do melhor caminho previamente descoberto, então este caminho parcial também pode ser descartado

Branch and Bound – Labirinto

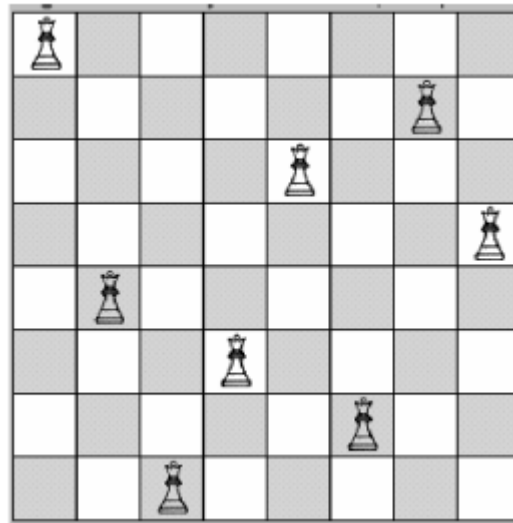
```
void labirinto(int M[MAX][MAX], int deltaL[], int deltaC[], int Li, int Ci, int Lf, int Cf, int &min) {
    int L, C, k;
    if ((Li == Lf) && (Ci == Cf)) {
        if (M[Lf][Cf] < min)
            min = M[Li][Ci];
    } else {
        /* testa todos os movimentos a partir da posicao atual */
        for (k = 0; k < 4; k++) {
            L = Li + deltaL[k];
            C = Ci + deltaC[k];
            /* verifica se o movimento eh valido e pode gerar uma solucao otima */
            if ((M[L][C] == -1) || (M[L][C] > M[Li][Ci] + 1)) {
                M[L][C] = M[Li][Ci] + 1;
                if (M[L][C] < min)
                    labirinto(M, deltaL, deltaC, L, C, Lf, Cf, min);
            }
        }
    }
}
```


Branch and Bound – Labirinto

```
int main() {
    int M[MAX][MAX], n, m, Li, Ci, Lf, Cf, min;
    // define os movimentos validos no labirinto
    int deltaL[4] = { 0, +1, 0, -1};
    int deltaC[4] = {+1, 0, -1, 0};
    // obtem as informacoes do labirinto
    obtemLabirinto(M, n, m, Li, Ci, Lf, Cf);
    M[Li - 1][Ci - 1] = 0; /* define a posicao inicial no tabuleiro */
    /* tenta encontrar um caminho no labirinto */
    min = INT_MAX;
    labirinto(M, deltaL, deltaC, Li - 1, Ci - 1, Lf - 1, Cf - 1, min);
    if (min == 0)
        cout<<"Nao existe solucao.\n";
    else {
        cout<<"Existe uma solucao em "<<min<<" passos.\n";
        imprimeLabirinto(M, n, m);
    }
    return 0;
}
```

Exercício

2. Proponha um algoritmo para a solução do problema das 8 rainhas utilizando *backtracking*. Dado um tabuleiro de xadrez (com 8 x 8 casas), o objetivo é distribuir 8 rainhas sobre este tabuleiro de modo que nenhuma delas fique em posição de ser atacada por outra rainha.



Divisão e Conquista

Ziviani – págs. 48 até 51


Cormen – págs. 21 até 28





Divisão e Conquista

- O paradigma divisão e conquista consiste em dividir o problema a ser resolvido em partes menores, encontrar soluções para as partes, e então combinar as soluções obtidas em uma solução global
- O paradigma divisão e conquista envolve três passos em cada nível de recursão:
 - **Dividir** o problema em um determinado número de subproblemas
 - **Conquistar** os subproblemas, resolvendo-os recursivamente. Se o tamanho dos subproblemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta
 - **Combinar** as soluções dadas aos problemas a fim de formar a solução para o problema original

Divisão e Conquista - Esquema

```
void divide_and_conquer(Problem P, Solution S) {  
    if (small(P))  
        S = compute_solution(P);  
    else {  
        divida P em problemas menores do mesmo  
        tipo do original, P1, P2, ..., Pk;  DIVIDIR  
        divide_and_conquer(P1, S1);  
        .  
        .  
        .  
        divide_and_conquer(Pk, Sk);  
        recombine S1, S2, ..., Sk em S, uma solução para P;  
    }  
}
```

 **CONQUISTAR**
Resolvendo
subproblemas
recursivamente

 **COMBINAR**

Divisão e Conquista

- Análise de Complexidade:
 - Se o tamanho do problema for pequeno o bastante $n \leq c$, para alguma constante c , a solução direta demorará um tempo constante $\Theta(1)$
 - Vamos supor que o problema seja dividido em a subproblemas, cada um dos quais com $1/b$ do tamanho do problema original
 - $D(n)$ é o tempo para dividir o problema em subproblemas
 - $C(n)$ tempo para combinar as soluções dadas aos subproblemas na solução para o problema original

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{caso contrário} \end{cases}$$

Exemplo: Maior e Menor Elemento

- Exemplo: encontrar o maior e o menor elemento de um vetor de inteiros, $A[0 \dots n-1]$, $n \geq 1$
- Cada chamada de `MaxMin4` atribui à `max` e `min` o maior e o menor elemento em $A[esq], A[esq+1], \dots, A[dir]$, respectivamente

Exemplo: Maior e Menor Elemento

```
void MaxMin4(int A[], int esq, int dir, int &min, int &max) {
    int min1, min2, max1, max2, meio;

    if (dir-esq <= 1) {
        if (A[esq] < A[dir]) {
            min = A[esq];
            max = A[dir];
        }
        else {
            min = A[dir];
            max = A[esq];
        }
    }
    else {
        meio = (esq+dir)/2;
        MaxMin4(A, esq, meio, min1, max1);
        MaxMin4(A, meio+1, dir, min2, max2);
        max = (max1 > max2) ? max1 : max2;
        min = (min1 < min2) ? min1 : min2;
    }
}
```


Análise do Maior e Menor Elemento

- Seja $T(n)$ o número de comparações entre os elementos de A

$$\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + 2 & \text{para } n > 2 \\ T(2) = 1 \end{cases}$$

- $T(n) = \frac{3n}{2} - 2$ para o melhor caso, pior caso e caso médio

Análise do Maior e Menor Elemento

- Conforme mostrado no início do curso, o algoritmo dado neste exemplo é ótimo
- Entretanto, ele pode ser pior que o `MaxMin3` pois, a cada chamada recursiva, salva `esq`, `dir`, `min` e `max` além do endereço de retorno da chamada para o procedimento
- Além disso, uma comparação adicional é necessária a cada chamada recursiva para verificar se $dir - esq \leq 1$
- $n+1$ deve ser menor que a metade do maior inteiro que pode ser representado pelo compilador, para não provocar *overflow* na operação `esq+dir`

Exemplo: Mergesort

- O algoritmo de ordenação Mergesort obedece ao paradigma de dividir e conquistar
 - **DIVIDIR**: divide a sequência de n elementos a serem ordenados em duas subsequências de $n/2$ elementos cada uma
 - **CONQUISTAR**: ordena as duas subsequências recursivamente, utilizando o MERGE-SORT
 - **COMBINAR**: faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada (função MERGE)

Exemplo: Mergesort

```
MERGE-SORT (A, p, r)
```

```
if p < r then
```

```
    q ← ⌊ (p + r) / 2 ⌋
```

← DIVIDIR

```
    MERGE-SORT (A, p, q)
```

```
    MERGE-SORT (A, q+1, r)
```

← CONQUISTAR

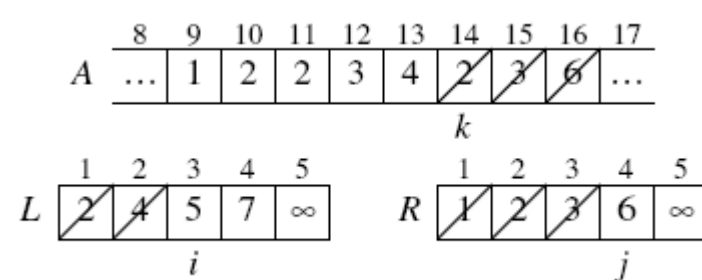
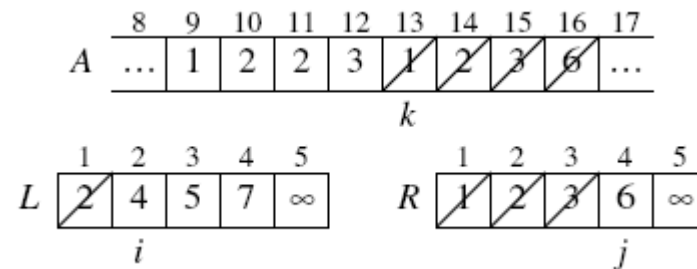
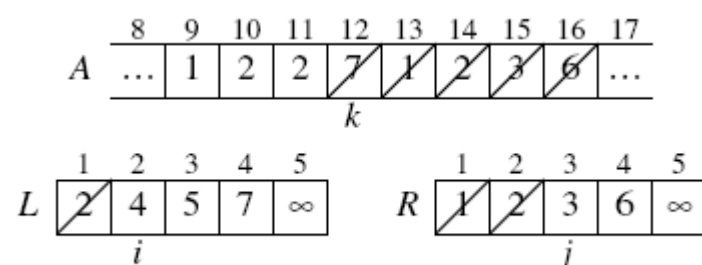
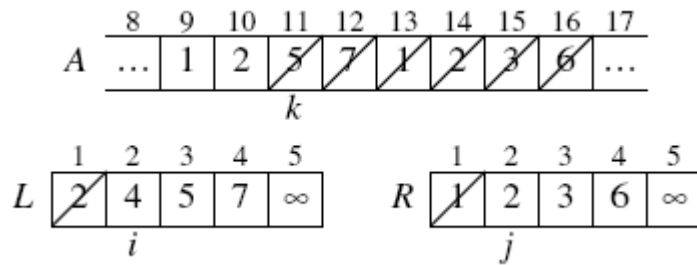
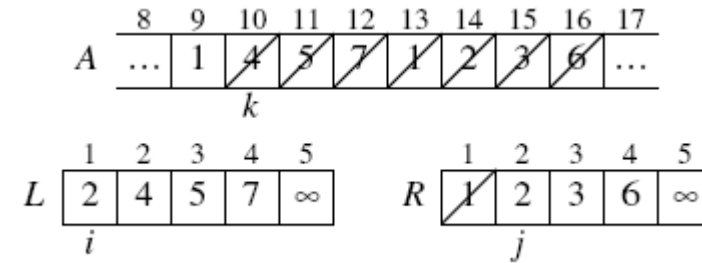
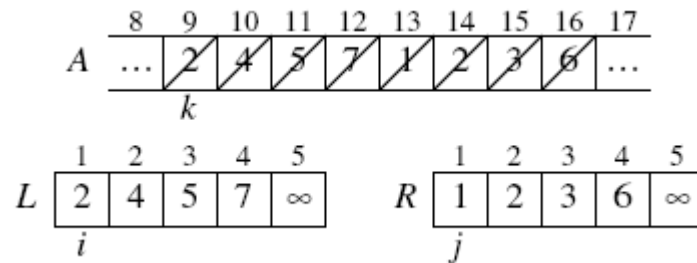
```
    MERGE (A, p, q, r)
```

← COMBINAR

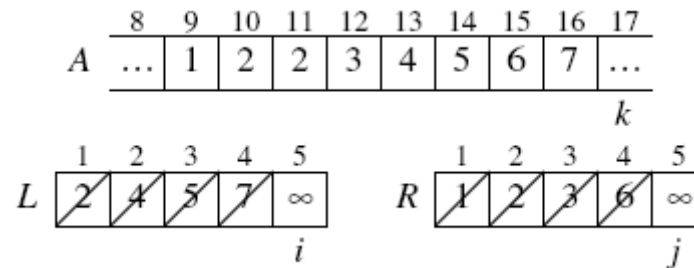
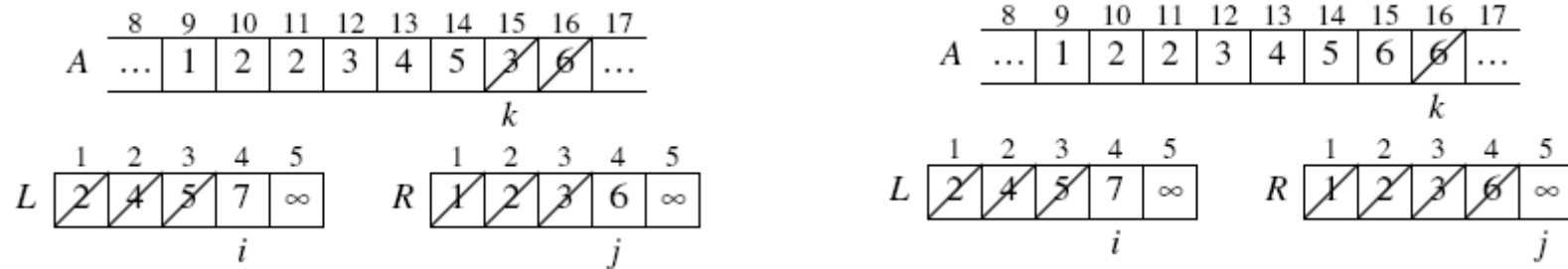
Exemplo: Mergesort

```
MERGE (A, p, q, r)
n1 ← q-p+1
n2 ← r-q
criar arranjos L[1..n1+1] e R[1..n2+1]
for i ← 1 to n1 do
    L[i] ← A[p+i-1]
for j ← 1 to n2 do
    R[j] ← A[q+j]
L[n1+1] ← ∞
R[n2+1] ← ∞
i ← 1
j ← 1
for k ← p to r do
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i+1
    else
        A[k] ← R[j]
        j ← j+1
```

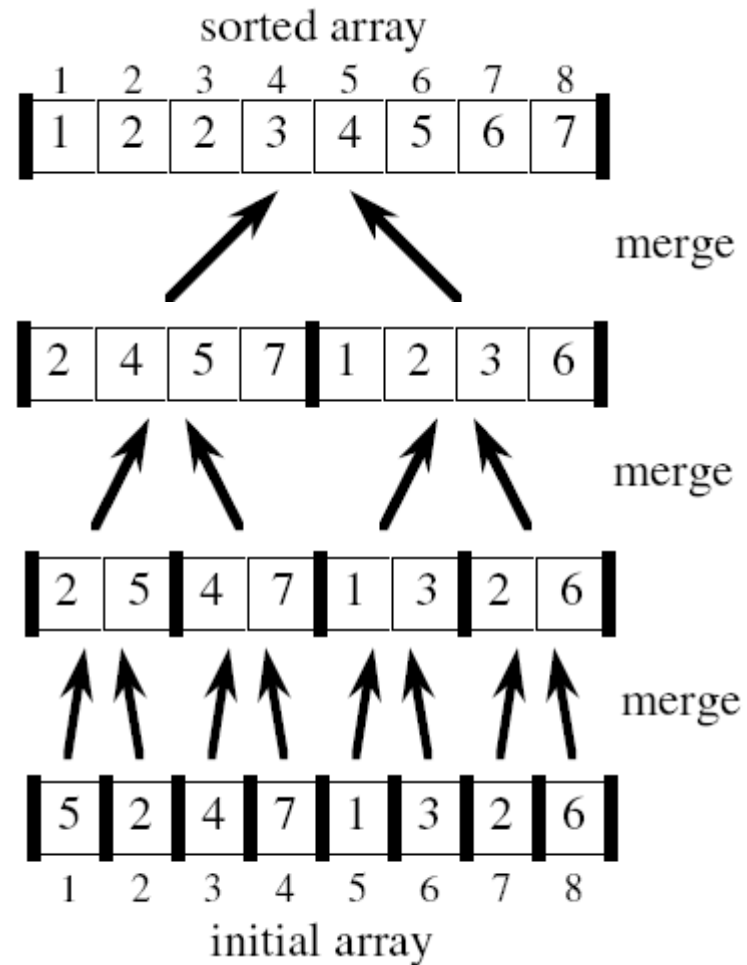
MERGE (A,9,12,16)



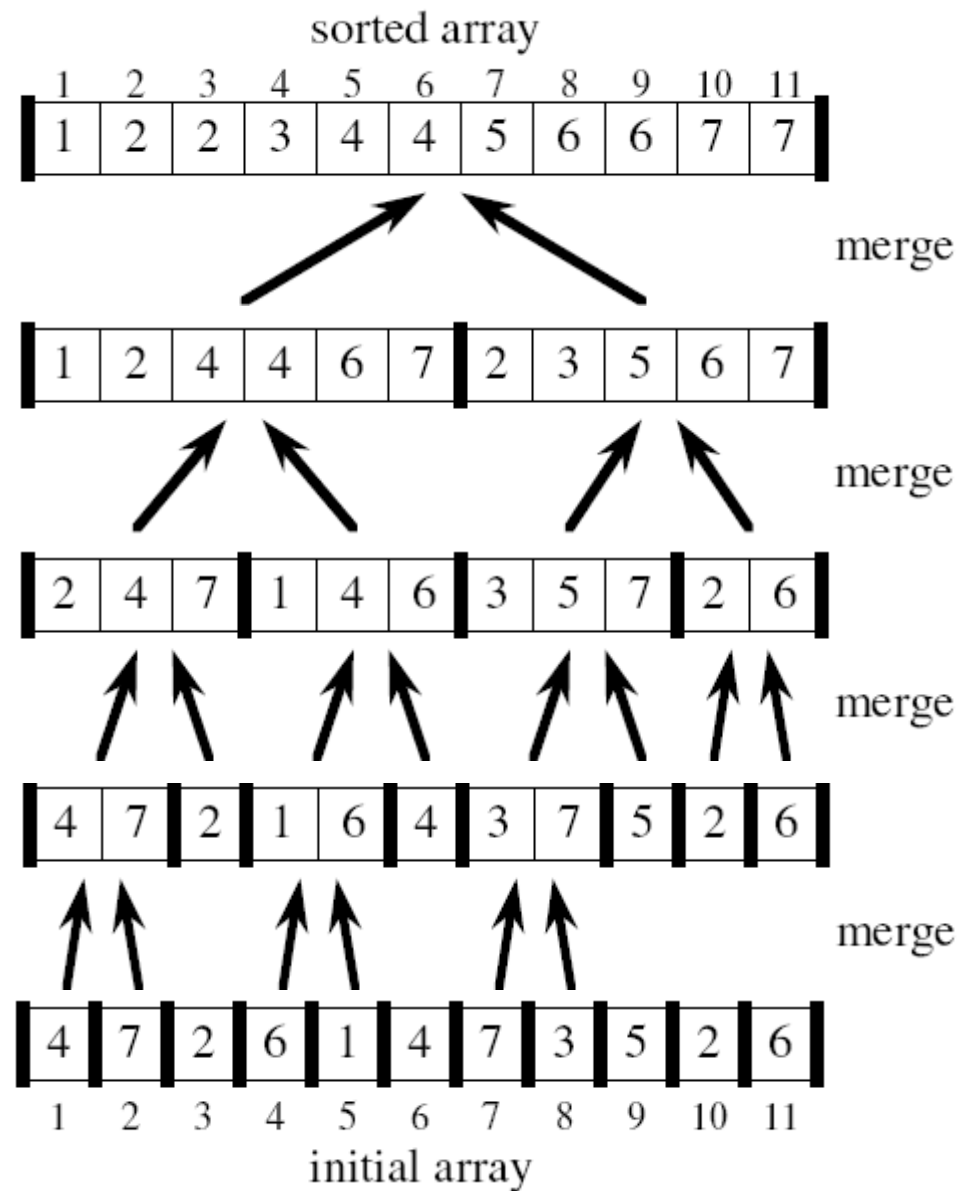
MERGE (A,9,12,16)



Exemplo: Mergesort



Exemplo: Mergesort



Análise do Mergesort

- Análise de Complexidade (supondo n par e que a operação relevante seja a comparação com os elementos do vetor):
 - **Dividir**: a etapa de dividir simplesmente calcula o ponto médio do subarranjo e não realiza comparação
 - **Conquistar**: resolvemos recursivamente dois subproblemas, cada um tem o tamanho $n/2$ e contribui com $2T(n/2)$ para o tempo de execução
 - **Combinar**: o procedimento MERGE leva o tempo n , onde $n=r-p+1$ é o número de elementos que estão sendo intercalados

$$\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + n & \text{se } n > 1 \\ T(1) = 0 \end{cases}$$

$$T(n) = n \log n$$

Quando Utilizar Divisão e Conquista

- Existem três condições que indicam que a estratégia de divisão e conquista pode ser utilizada com sucesso:
 - Deve ser possível decompor uma instância em subinstâncias
 - A combinação dos resultados deve ser eficiente (muitas vezes, trivial)
 - As subinstâncias devem ser mais ou menos do mesmo tamanho

Exemplo: Quicksort

```
void partition (int a[], int esq, int dir, int &i, int &j) {
    int aux, x;
    i = esq;
    j = dir;
    x = a[(i+j)/2];
    while (i <= j) {
        while (x > a[i]) i++;
        while (x < a[j]) j--;
        if (i <= j) {
            aux = a[i];
            a[i] = a[j];
            a[j] = aux;
            i++;
            j--;
        }
    }
}

void quicksort (int a[], int esq, int dir) {
    int i, j;
    partition(a, esq, dir, i, j);
    if (esq < j)
        quicksort(a, esq, j);
    if (i < dir)
        quicksort(a, i, dir);
}
```

Exemplo: Seleção

- Encontrar o k th menor elemento de um conjunto de números

```
int partition (int a[], int esq, int dir) {
    int i, j, aux, x;
    i = esq;
    j = dir;
    x = a[(i+j)/2];
    while (i <= j) {
        while (x > a[i]) i++;
        while (x < a[j]) j--;
        if (i <= j) {
            aux = a[i];
            a[i] = a[j];
            a[j] = aux;
            i++;
            j--;
        }
    }
    return (i-1);
}
```

```
int selection(int a[], int l, int r, int k) {
    int i;
    if (r > l) {
        i = partition(a, l, r);
        if (i == k)
            return (i);
        if (i > l+k-1)
            return (selection(a, l, i-1, k));
        if (i < l+k-1)
            return (selection(a, i+1, r, k-i));
    }
}
```

Exercício

3. Implemente a função merge com custo de $n-1$ comparações no pior caso. Encontre a função de complexidade do Mergesort quando o mesmo utiliza essa função merge.
4. Faça a análise de complexidade do melhor caso do Quicksort e do algoritmo para encontrar o k th menor elemento de um conjunto de números

Programação Dinâmica

Ziviani – págs. 54 até 57

Cormen – págs. 259 até 295



Programação Dinâmica

- **Divisão e Conquista**

- Problema é partido em subproblemas que se resolvem separadamente
- A solução é obtida por combinação das soluções
- É *top-down*

- **Programação Dinâmica**

- Resolvem-se os problemas de pequena dimensão e guardam-se as soluções
- A solução de um problema é obtida combinando as de problemas de menor dimensão
- É *bottom-up*
- 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

Aplicação Direta – Números de Fibonacci

- Encontrar o n -ésimo número de Fibonacci

$$f_0 = 0, f_1 = 1,$$

$$f_n = f_{n-1} + f_{n-2} \text{ para } n \geq 2$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

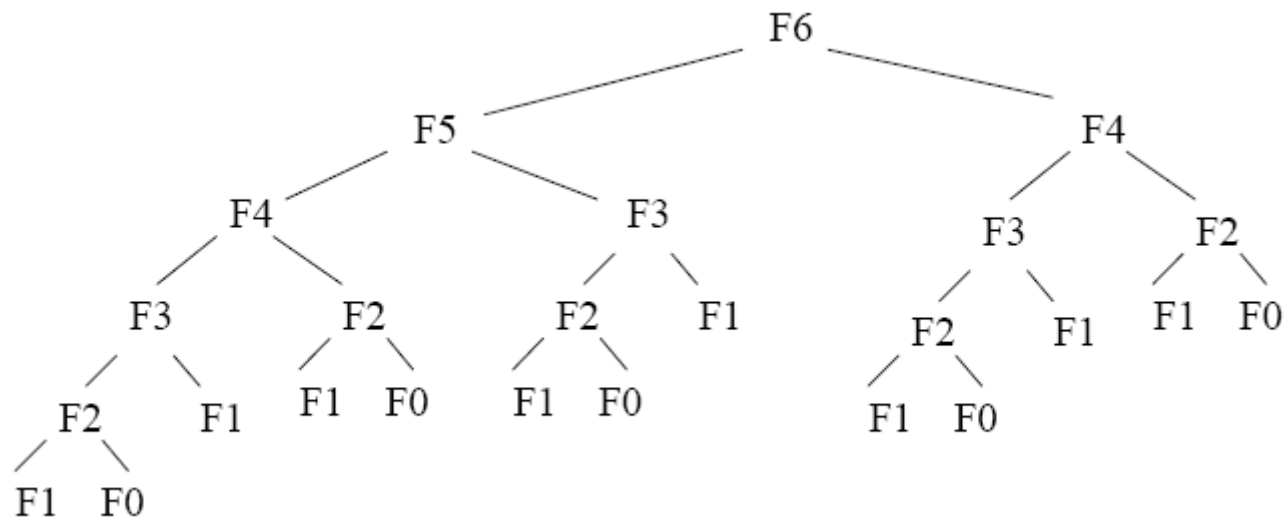
```
unsigned int FibRec (unsigned int n) {  
    if (n < 2)  
        return n;  
    else  
        return (FibRec(n-1) + FibRec(n-2));  
}
```

Simples!!! Elegante!!!

Mas vamos analisar o seu desempenho!!!

Aplicação Direta – Números de Fibonacci

- Problema do algoritmo recursivo: repetição de chamadas iguais



Aplicação Direta – Números de Fibonacci

- Análise de Complexidade:
 - Considerando que a medida de complexidade de tempo $T(n)$ é o número de adições, então

$$\begin{cases} T(n) = T(n-1) + T(n-2) + 1 & \text{para } n \geq 2 \\ T(n) = 0 & \text{para } n \leq 1 \end{cases}$$

- $T(n) = \Theta(1,618^n)$

Algoritmo recursivo é exponencial

Problema: repetição de chamadas

Aplicação Direta – Números de Fibonacci

- Solução iterativa

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

Resolvemos
problemas
menores

Em todo estágio, guardamos
as soluções para os dois
problemas anteriores nas
variáveis *i* e *F*

- Considerando que a medida de complexidade de tempo $T(n)$ é o número de adições, então $T(n) = \Theta(n)$

Devemos evitar o uso de recursividade quando
existe solução óbvia por iteração

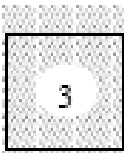
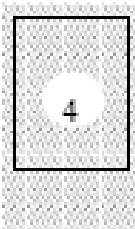
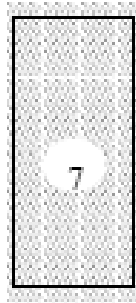
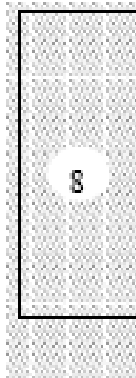
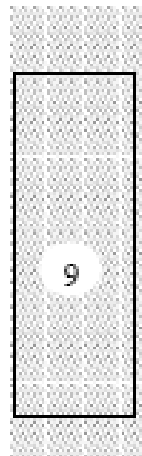
Programação Dinâmica

- Abordagem
 - Resolva problemas menores
 - Armazene as soluções
 - Use aquelas soluções para resolver problemas maiores
- Algoritmos dinâmicos usam espaço!

Problema da Mochila

- O ladrão encontra o cofre cheio de itens de vários tamanhos e valores, mas tem apenas uma mochila de capacidade limitada. Qual é a combinação de itens que deve levar para maximizar o valor do roubo?

Exemplo

Tamanho:					
Valor:	4	5	10	11	13
Nome:	A	B	C	D	E

- Considerando que a mochila tem capacidade 17, qual é a melhor combinação?

Problema da Mochila

- Muitas situações de interesse comercial
 - Melhor forma de carregar um caminhão ou avião
- Tem variantes: número de itens de cada tipo pode ser limitado
- Abordagem programação dinâmica:
 - Calcular a melhor combinação para todas as mochilas de tamanho até M
 - Cálculo é eficiente se feito na ordem apropriada

Problema da Mochila

```
for( j = 1; j <= N; j++ )
{
    for( i=1; i <= M; i++ )
        if ( i >= size[j] )
            if ( cost[i] < cost[i-size[j]] + val[j] )
            {
                cost[i] = cost[i-size[j]] + val[j];
                best[i] = j;
            }
}
```

- `cost[i]`: maior valor que se consegue com mochila de capacidade `i`
- `best[i]`: último item acrescentado para obter o máximo
- Calcula-se o melhor valor que se pode obter usando só itens tipo A, para todos os tamanhos de mochila
- Repete-se usando só A's e B's, e assim sucessivamente

Problema da Mochila

- Quando um item j é escolhido para a mochila: o melhor valor que se pode obter é $val[j]$ (do item) mais $cost[i - size[j]]$ (para encher o resto)
- Se o valor assim obtido é superior ao que se consegue sem usar o item j , atualiza-se $cost[i]$ e $best[i]$; senão mantém-se
- Conteúdo da mochila ótima: recuperado através do vetor $best[i]$
 - $best[i]$ indica o último item da mochila
 - O restante é o indicado para a mochila de tamanho $M - size[best[i]]$
- Eficiência: A solução em programação dinâmica gasta tempo $\Theta(NM)$

Problema da Mochila

	k	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1	cost[k]	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	best[k]	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2	cost[k]	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	best[k]	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3	cost[k]	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	best[k]	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4	cost[k]	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	best[k]	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5	cost[k]	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	best[k]	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Exercício

5. Considere os seguintes itens e que a mochila tem capacidade 20, qual é a combinação de itens que devemos levar para maximizar o valor?

Item	A	B	C	D
Tamanho	3	4	5	6
Valor	13	15	20	15

Multiplicação de uma Cadeia de Matrizes

- Dada uma sequência de matrizes de dimensões diversas, como fazer o seu produto minimizando o esforço computacional
- Exemplo:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \begin{bmatrix} c_{11} \\ c_{21} \\ c_{31} \end{bmatrix} \begin{bmatrix} d_{11} & d_{12} \end{bmatrix} \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{bmatrix}$$

- Multiplicando da esquerda para a direita: 84 operações
- Multiplicando da direita para a esquerda: 69 operações
- **Qual a melhor sequência?**

Multiplicação de uma Cadeia de Matrizes

- Multiplicar N matrizes $M_1 M_2 M_3 \dots M_N$ na qual M_i tem r_i linhas e r_{i+1} colunas
- Multiplicação de uma matriz $p \times q$ por outra $q \times r$ produz uma matriz $p \times r$ requerendo q produtos para cada entrada, totalizando pqr operações de multiplicação
- Algoritmo em programação dinâmica:
 - Multiplicar 2 matrizes: só há uma maneira de multiplicar; registra-se o custo
 - Multiplicar 3 matrizes: o menor custo de realizar $M_1 M_2 M_3$ é calculado comparando os custos de multiplicar $M_1 M_2$ por M_3 e de multiplicar M_1 por $M_2 M_3$; registra-se o menor custo
 - O procedimento repete-se para sequências de tamanho crescente

Multiplicação de uma Cadeia de Matrizes

- No geral:
 - Para $1 \leq j \leq N-1$ encontra-se o custo mínimo de calcular $M_i M_{i+1} \dots M_{i+j}$ encontrando, para $1 \leq i \leq N-j$ e para cada k entre i e $i+j$ os custos de obter $M_i M_{i+1} \dots M_{k-1}$ e $M_k M_{k+1} \dots M_{i+j}$ somando o custo de multiplicar esses resultados

Multiplicação de uma Cadeia de Matrizes

```
for( i=1; i <= N; i++ )
    for( j = i+1; j <= N; j++ ) cost[i][j] = INT_MAX;
for( i=1; i <= N; i++ ) cost[i][i] = 0;
for( j=1; j < N; j++ )
    for( i=1; i <= N-j; i++ )
        for( k= i+1; k <= i+j; k++ )
        {
            t = cost[i][k-1] + cost[k][i+j] +
                r[i]*r[k]*r[i+j+1];
            if( t < cost[i][i+j] )
                { cost[i][i+j] = t; best[i][i+j] = k;
                }
        }
```

Multiplicação de uma Cadeia de Matrizes

- Para $1 \leq j \leq N-1$ encontra-se o custo mínimo de calcular $M_i M_{i+1} \dots M_{i+j}$
 - Para $1 \leq i \leq N-j$ e para cada k entre i e $i+j$ calculam-se os custos para obter $M_i M_{i+1} \dots M_{k-1}$ e $M_k M_{k+1} \dots M_{i+j}$
 - Soma-se o custo de multiplicar esses 2 resultados
- Cada grupo é partido em grupos menores
 - Custos mínimos para os 2 grupos são vistos numa tabela
- Custo da multiplicação final $M_i M_{i+1} \dots M_{k-1}$ é uma matriz $r_i \times r_k$ e $M_k M_{k+1} \dots M_{i+j}$ é uma matriz $r_k \times r_{i+j+1}$, o custo de multiplicar as duas é $r_i r_k r_{i+j+1}$

Multiplicação de uma Cadeia de Matrizes

- $\text{cost}[1][r]$ é o custo mínimo para $M_1 M_{1+1} \dots M_r$
- Programa obtém $\text{cost}[i][i+j]$ para $1 \leq i \leq N-j$ com j de 1 a $N-1$
- Chegando a $j=N-1$, tem-se o custo de calcular $M_1 M_2 \dots M_N$
- Recuperar a sequência ótima
 - Guarda o registro das decisões feitas para cada dimensão
 - Permite recuperar a sequência de custo mínimo

Multiplicação de uma Cadeia de Matrizes

	B	C	D	E	F
A	24 [A][B]	14 [A][BC]	22 [ABC][D]	26 [ABC][DE]	36 [ABC][DEF]
B		6 [B][C]	10 [BC][D]	14 [BC][DE]	22 [BC][DEF]
C			6 [C][D]	10 [C][DE]	19 [C][DEF]
D				4 [D][E]	10 [DE][F]
E					12 [E][F]

Exercícios

6. Encontre uma colocação ótima de parênteses de um produto de cadeias de matrizes cuja sequência de dimensões é $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$

Árvore de Pesquisa Binária Ótima

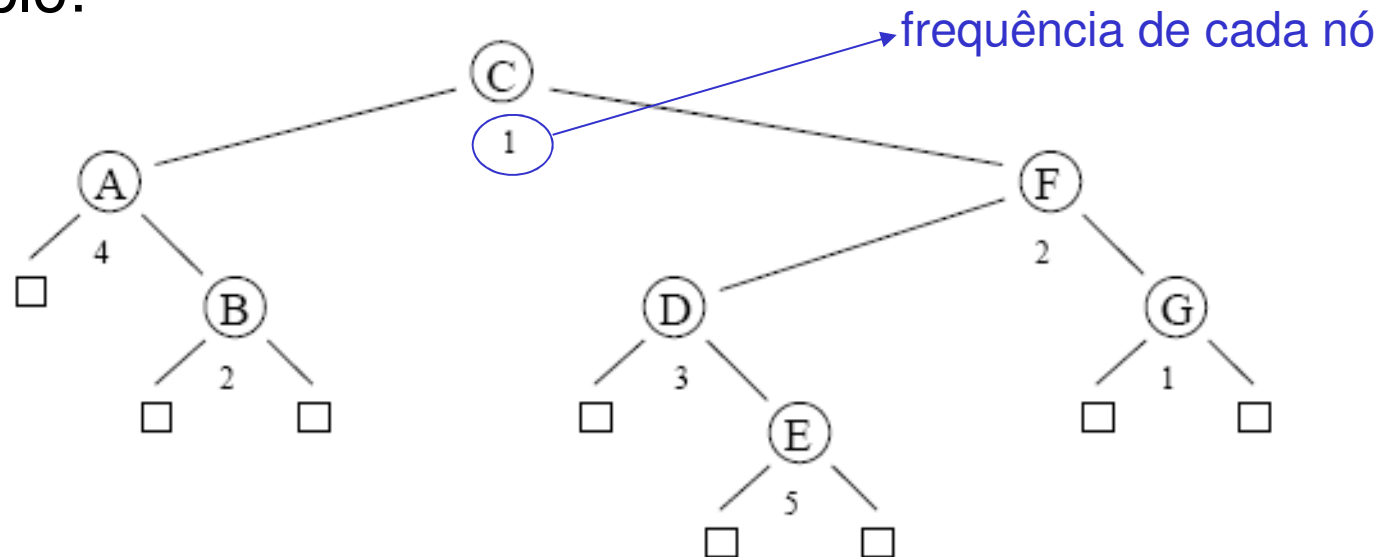
- Em pesquisa, as chaves ocorrem com frequências diversas; exemplos:
 - Verificador ortográfico: encontra mais frequentemente as palavras mais comuns
 - Compilador de C++: encontra mais frequentemente “`if`” e “`for`” que “`main`”
- Usando uma árvore de pesquisa binária, é vantajoso ter mais perto do topo as chaves mais usadas
- Algoritmo de programação dinâmica pode ser usado para organizar as chaves de forma a minimizar o custo total da pesquisa

Árvore de Pesquisa Binária Ótima

- Problema tem semelhança com o dos códigos de Huffman (minimização do tamanho do caminho externo)
 - Mas, código de Huffman não requer a manutenção da ordem das chaves
 - Na árvore de pesquisa binária, os nós à esquerda de cada nó têm chaves menores e os nós à direita têm chaves maiores
- Problema é semelhante ao da ordem de multiplicação de uma cadeia de matrizes

Árvore de Pesquisa Binária Ótima

- Exemplo:



$$\text{Custo da árvore} = 1 \times 1 + 4 \times 2 + 2 \times 3 + 2 \times 2 + 3 \times 3 + 5 \times 4 + 1 \times 3 = 51$$

- Custo da árvore é o comprimento do caminho interno ponderado da árvore:

Multiplicar a frequência de cada nó pela sua distância à raiz

Soma para todos os nós

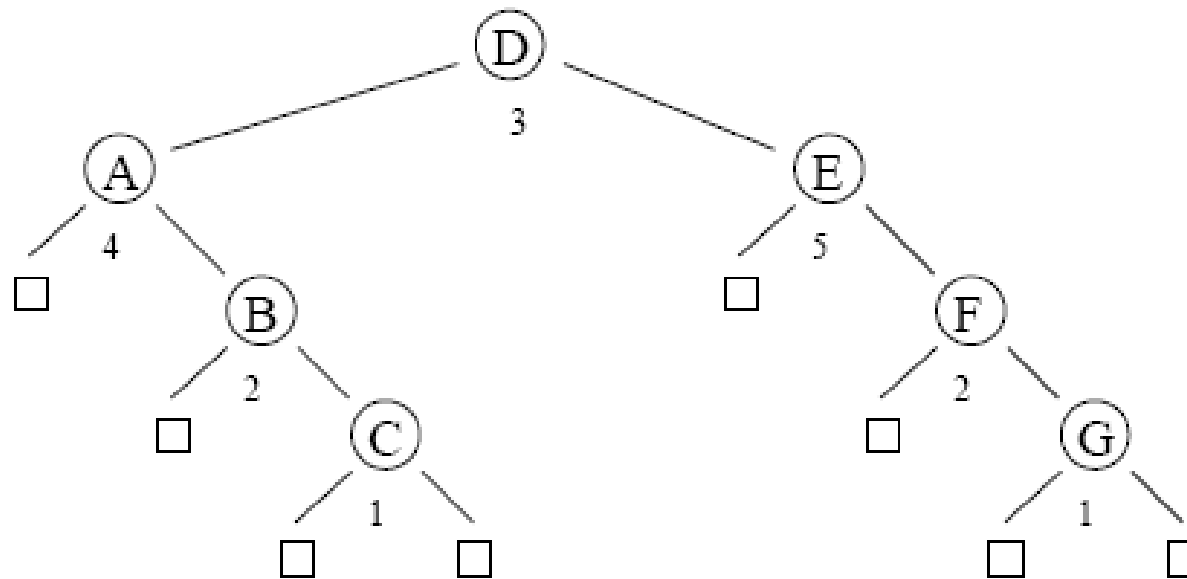
Árvore de Pesquisa Binária Ótima

- Dados:
 - Chaves $K_1 < K_2 < \dots < K_n$
 - Frequências r_1, \dots, r_n
- Construir árvore de pesquisa que minimize a soma, para todas as chaves, dos produtos das frequências pelas distâncias à raiz
- Abordagem em programação dinâmica
 - Calcular, para cada j de 1 a $N-1$, a melhor maneira de construir subárvores contendo $K_i, K_{i+1}, \dots, K_{i+j}$, para $1 \leq i \leq N-j$
 - Para cada j , tenta-se cada nó como raiz e usam-se os valores já computados para determinar as melhores escolhas para as subárvores
 - Para cada k entre i e $i+j$, construir a árvore ótima contendo $K_i, K_{i+1}, \dots, K_{i+j}$ com K_k na raiz
 - A árvore com K_k na raiz é formada usando a árvore ótima para $K_i, K_{i+1}, \dots, K_{k-1}$ como subárvore esquerda e a árvore ótima para $K_{k+1}, K_{k+2}, \dots, K_{i+j}$ como subárvore direita

Árvore de Pesquisa Binária Ótima

```
for( i=1; i <= N; i++ )
    for( j = i+1; j <= N+1; j++ ) cost[i][j] = INT_MAX;
for( i=1; i <= N; i++ ) cost[i][i] = f[i];
for( i=1; i <= N+1; i++ ) cost[i][i-1] = 0;
for( j=1; j <=N-1; j++ )
    for( i=1; i <= N-j; i++ )
        {
            for( k= i; k <= i+j; k++ )
                {
                    t = cost[i][k-1] + cost[k+1][i+j];
                    if( t < cost[i][i+j] )
                        { cost[i][i+j] = t; best[i][i+j] = k; }
                }
            for( k= i; k <= i+j; cost[i][i+j] += f[k++] );
        }
```


Árvore de Pesquisa Binária Ótima



$$\text{Custo da árvore} = 3 \times 1 + 4 \times 2 + 2 \times 3 + 1 \times 4 + 5 \times 2 + 2 \times 3 + 1 \times 4 = 41$$

Árvore de Pesquisa Binária Ótima

- O método para determinar uma árvore de pesquisa binária ótima em programação dinâmica gasta tempo $\Theta(N^3)$ e espaço $\Theta(N^2)$
- Examinando o código:
 - O algoritmo trabalha com uma matriz de dimensão N^2 e gasta tempo proporcional a N em cada entrada
- É possível melhorar:
 - Usando o fato de que a posição ótima para a raiz da árvore não pode ser muito distante da posição ótima para uma árvore um pouco menor, no programa dado k não precisa de cobrir todos os valores de i a $i+j$

Programação Dinâmica - Resumo

- Tradução iterativa inteligente da recursão
 - Resolvem problemas menores
 - Armazenam as soluções para estes problemas menores numa tabela
 - Usam as soluções dos problemas menores para obterem a solução de problemas maiores
- Cada instância do problema é resolvida a partir da solução de subinstâncias da instância original
- O problema deve ter estrutura recursiva: a solução de toda instância do problema deve “conter” soluções de subinstâncias da instância

Exercícios

7. Determine o custo e a estrutura de uma árvore de pesquisa binária ótima para um conjunto de $n=7$ chaves com seguinte frequência de ocorrência:

	A	B	C	D	E	F	G
<i>f</i>	7	4	5	1	4	8	7

Algoritmos Gulosos

Ziviani – págs. 58 até 59

Cormen – págs. 296 até 323

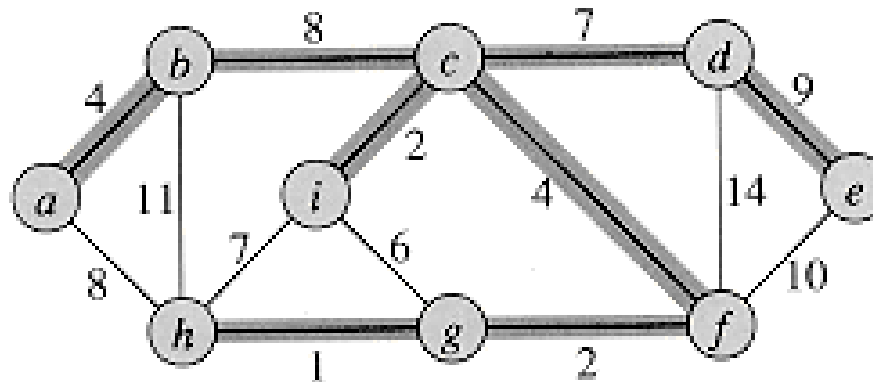


Algoritmos Gulosos

- Para resolver um problema, um algoritmo guloso escolhe, em cada iteração, a melhor opção para o momento
- A opção escolhida passa a fazer parte da solução que o algoritmo constrói
- O algoritmo faz uma escolha ótima local esperando que esta o leve a uma solução ótima global
- Um algoritmo guloso jamais se arrepende ou volta atrás, as escolhas que faz em cada iteração são definitivas

Árvore Geradora Mínima


- Árvore Geradora Mínima é a árvore geradora de menor peso de G
- Dado um grafo G com pesos associados às arestas, encontrar uma árvore geradora mínima de G



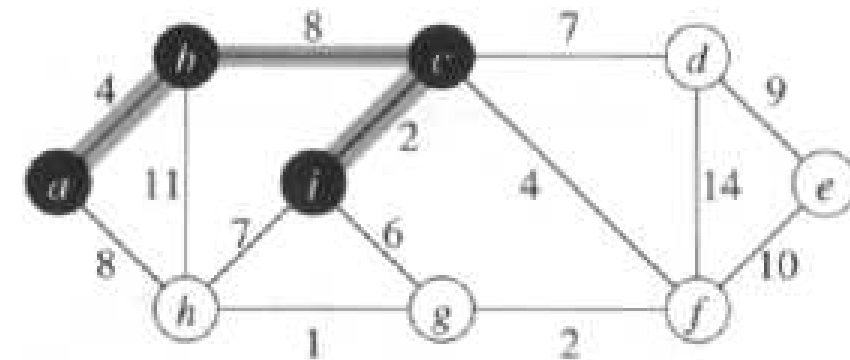
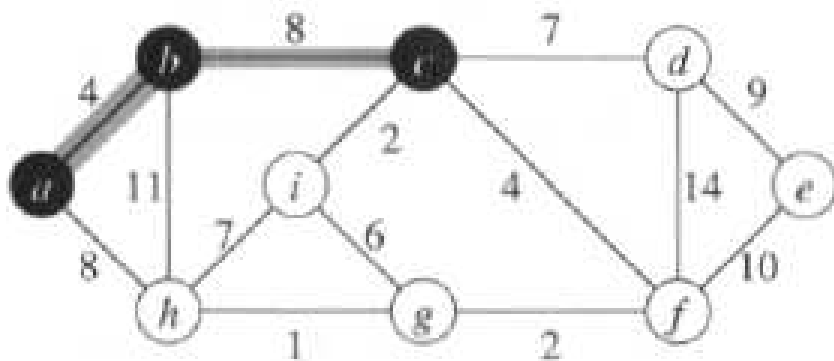
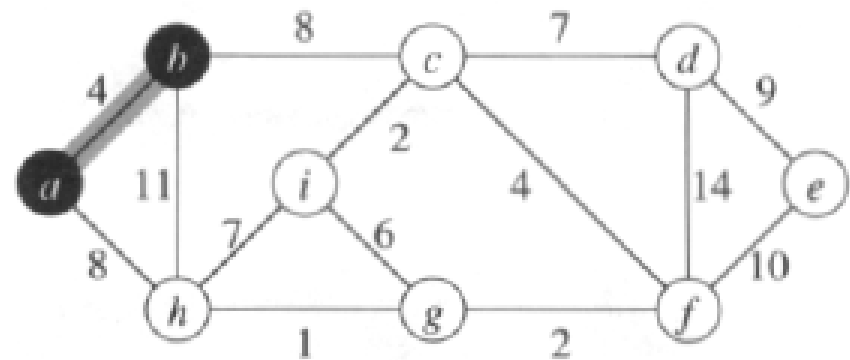
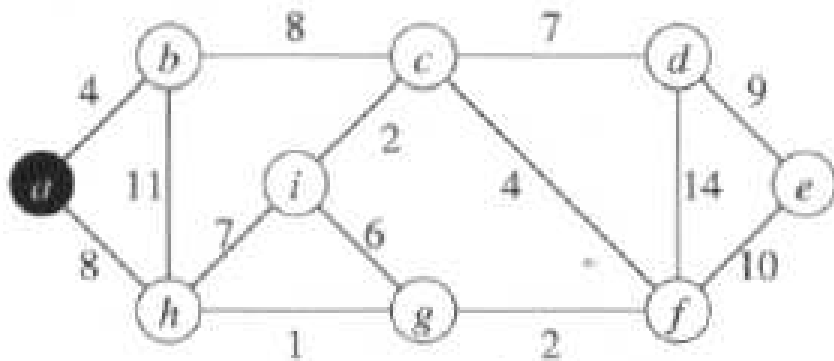
Algoritmo de Prim

- As arestas no conjunto A sempre formam uma árvore única
- A árvore começa a partir de um vértice de raiz arbitrária r e aumenta até a árvore alcançar todos os vértices em V
- Em cada etapa, uma aresta leve conectando um vértice de A a um vértice em $V-A$ é adicionada à árvore
- Quando o algoritmo termina, as arestas em A formam uma árvore geradora mínima
- Durante a execução do algoritmo, todos os vértices que não estão na árvore residem em uma fila de prioridade mínima Q baseada em um campo `chave`
- Para cada vértice v , `chave[v]` é o peso mínimo de qualquer aresta que conecta v a um vértice na árvore
- $\pi[v]$ é o pai de v na árvore

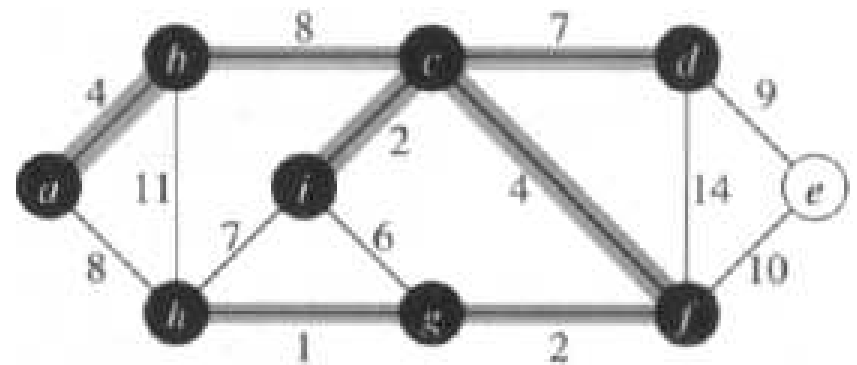
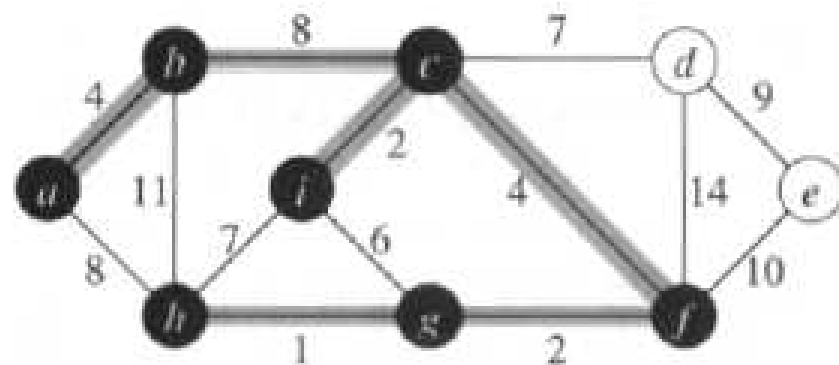
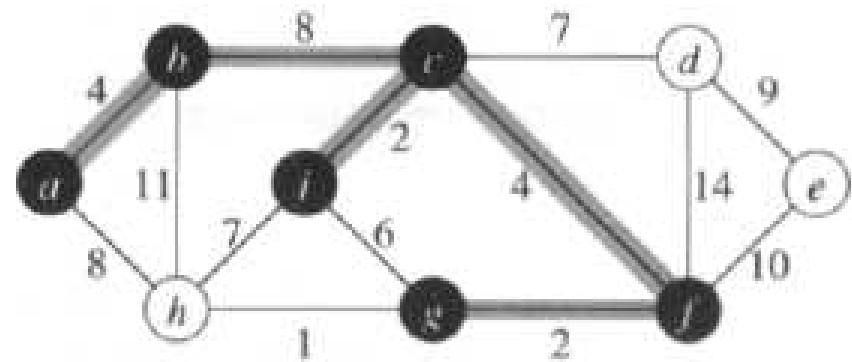
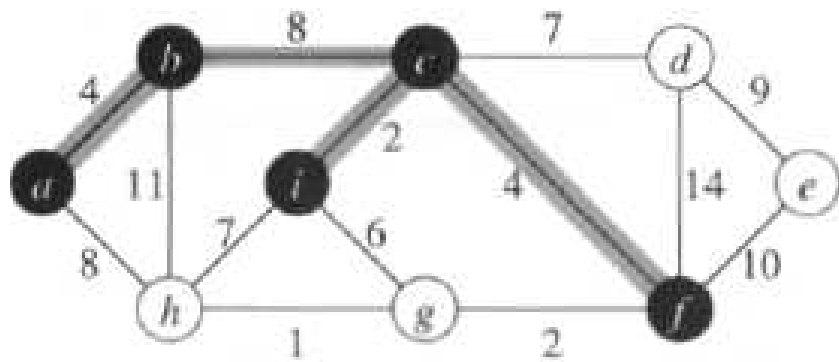
Algoritmo de Prim

```
MST-PRIM (G, w, r)
for cada u ∈ V[G] do
    chave[u] ← ∞
    π[u] ← NIL
chave[r] ← 0
Q ← V[G]
while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q)  INSERE NA AGM
    for cada v ∈ Adj[u] do
        if v ∈ Q e w(u, v) < chave[v] then
            π[v] ← u
            chave[v] ← w(u, v)
```

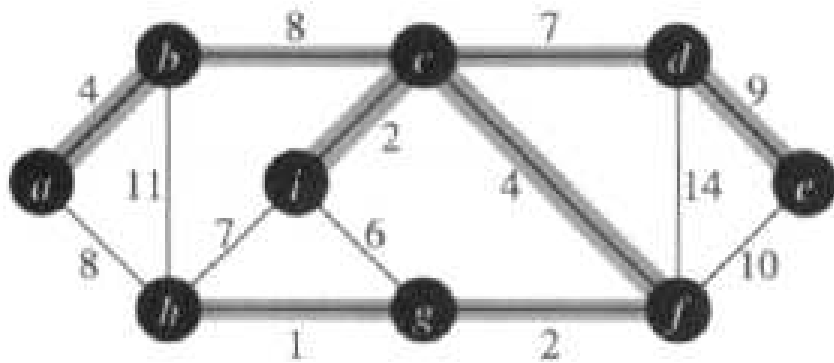
Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim



Versões do Problema da Mochila

- Problema da Mochila 0-1 ou 0-1 *Knapsack Problem*:
 - O item i é levado integralmente ou é deixado
- Problema da Mochila Fracionário:
 - Fração do item i pode ser levada

Considerações sobre as duas versões

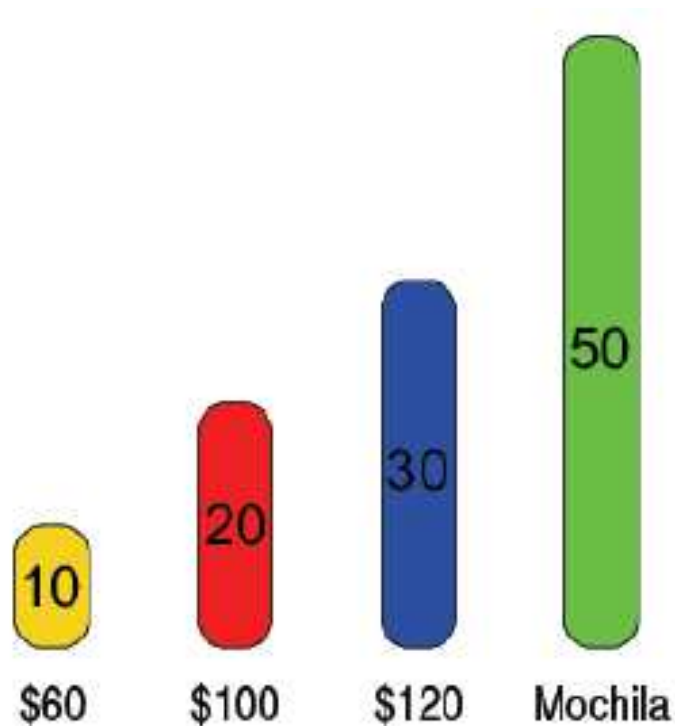
- Possuem a propriedade de subestrutura ótima
- Problema inteiro:
 - Considere uma carga que pesa no máximo W com n itens
 - Remova o item j da carga
 - Carga restante deve ser a mais valiosa pesando no máximo $W - w_j$ com $n - 1$ itens
- Problema fracionário:
 - Considere uma carga que pesa no máximo W com n itens
 - Remova um peso w do item j da carga
 - Carga restante deve ser a mais valiosa pesando no máximo $W - w$ com $n - 1$ itens mais o peso $w_j - w$ do item j

Considerações sobre as duas versões

- Problema inteiro
 - Não é resolvido usando a técnica gulosa
- Problema fracionário
 - É resolvido usando a técnica gulosa
- Estratégia para resolver o problema fracionário:
 - Calcule o valor por unidade de peso v_i / w_i para cada item
 - Estratégia gulosa é levar tanto quanto possível do item de maior valor por unidade de peso
 - Repita o processo para o próximo item com esta propriedade até alcançar a carga máxima
- Complexidade para resolver o problema fracionário:
 - Ordene os itens i ($i = 1, \dots, n$) pelas frações v_i / w_i
 - $\Theta(n \log n)$

Exemplo: Situação inicial

Problema 0-1

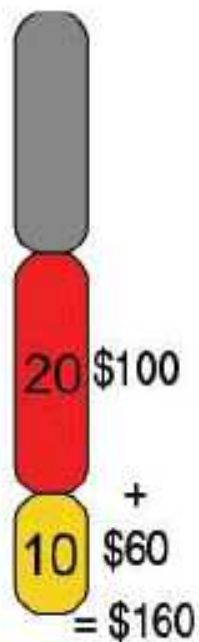
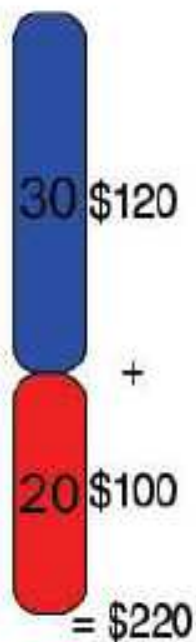


Item	Peso	Valor	V/P
1	10	60	6
2	20	100	5
3	30	120	4

Carga máxima da mochila: 50

Exemplo: Estratégia Gulosa

Problema 0-1



Soluções possíveis:

#	Item (Valor)
1	2 + 3 = 100 + 120 = 220
2	1 + 2 = 60 + 100 = 160
3	1 + 3 = 60 + 120 = 180

→ Solução 2 é a gulosa.

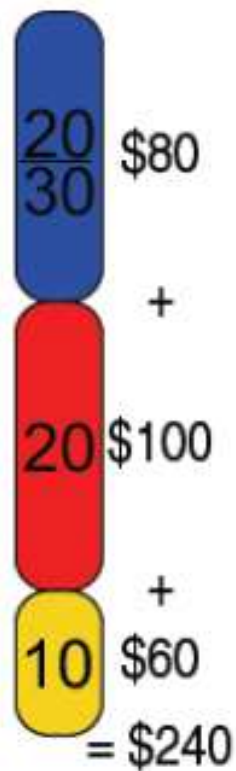
Exemplo: Estratégia Gulosa

Problema 0-1

- Considerações:
 - Levar o item 1 faz com que a mochila fique com espaço vazio
 - Espaço vazio diminui o valor efetivo da relação v/w
 - Neste caso deve-se comparar a solução do subproblema quando:
Item é incluído na solução X Item é excluído da solução
 - Passam a existir vários subproblemas
 - Programação dinâmica passa a ser a técnica adequada

Exemplo: Estratégia Gulosa

Problema Fracionário



Item	Peso	Valor	Fração
1	10	60	1
2	20	100	1
3	30	80	$\frac{2}{3}$

→ Total = 240.

→ Solução ótima!

O Problema da Mochila Fracionária

- O algoritmo é guloso porque, em cada iteração, abocanha o objeto de maior valor específico dentre os disponíveis, sem se preocupar com o que vai acontecer depois. O algoritmo jamais se arrepende do valor atribuído a um componente de x

Exercício

8. Resolva o problema da mochila fracionária considerando uma mochila com capacidade 50 e 4 itens conforme peso e valor especificados na tabela abaixo

	A	B	C	D	E
Peso	40	30	20	10	20
Valor	840	600	400	100	300

Teoria da Complexidade

Introdução

- A maioria dos problemas conhecidos e estudados se divide em dois grupos:
 - Problemas cuja solução é limitada por um polinômio de grau pequeno
 - Pesquisa binária: $\Theta(\log n)$
 - Ordenação: $\Theta(n \log n)$
 - Multiplicação de matriz: $\Theta(n^{2.81})$
 - Problemas cujo melhor algoritmo conhecido é não-polinomial
 - Problema do Caixeiro Viajante: $\Theta(n^2 2^n)$
 - Knapsack Problem: $\Theta(2^{n/2})$

Algoritmos Polinomiais X Algoritmos Exponenciais

- Algoritmos polinomiais são obtidos através de um entendimento mais profundo da estrutura do problema
- Um problema é considerado **tratável** quando existe um algoritmo polinomial para resolvê-lo
- Algoritmos exponenciais são, em geral, simples variação de pesquisa exaustiva
- Um problema é considerado **intratável** se ele é tão difícil que não existe um algoritmo polinomial para resolvê-lo

Algoritmos Polinomiais X Algoritmos Exponenciais

- Entretanto,
 - Um algoritmo $\Theta(2^n)$ é mais rápido que um algoritmo $\Theta(n^5)$ para $n \leq 20$
 - Existem algoritmos exponenciais que são muito úteis na prática
 - Algoritmo Simplex para programação linear é exponencial mas, executa muito rápido na prática
 - Na prática os algoritmos polinomiais tendem a ter grau 2 ou 3 no máximo e não possuem coeficientes muito grandes n^{100} ou $10^{99}n^2$ NÃO OCORREM

Decisão x Otimização

- Em um problema de **otimização** queremos determinar uma solução possível com o melhor valor.
 - Em um problema de **decisão** queremos responder “sim” ou “não”.
 - Para cada problema de otimização podemos encontrar um problema de decisão equivalente a ele.
-

Problemas “Sim/Não” ou Problemas de Decisão

- Para o estudo teórico da complexidade de algoritmos é conveniente considerar problemas cujo resultado seja “sim” ou “não”
- Exemplo: Problema do Caixeiro Viajante
 - **Dados:** Um conjunto de cidades $C = \{c_1, c_2, \dots, c_n\}$, uma distância $d(c_i, c_j)$ para cada par de cidades $c_i, c_j \in C$ e uma constante K .
 - **Questão:** Existe um roteiro para todas as cidades em C cujo comprimento total seja menor ou igual a K ?

Classe P e NP

- Classe P:
 - Um algoritmo está na Classe P se a complexidade do seu pior caso é uma função polinomial do tamanho da entrada de dados
- Classe NP:
 - Classe de problemas “Sim/Não” para os quais uma dada solução pode ser verificada facilmente
 - Existe uma enorme quantidade de problemas em NP para os quais não se conhece um único algoritmo polinomial para resolver qualquer um deles

Ordenação está em NP

```
VOrdenacao (A, n)
inicio
    ordenado ← verdadeiro
    para i ← 1 até n-1 faça
        se A[i] > A[i+1] então
            ordenado ← falso
        fim se
    fim para
    se ordenado = falso então
        escreva "NAO"
    senão
        escreva "SIM"
    fim se
fim
```

- Complexidade: $\Theta(n)$

Coloração de Grafos está em NP

```
VColoracao(G,C,K)
inicio
    colorido ← verdadeiro
    para i ← 1 até |E| faça
        se C[Ei.V1] = C[Ei.V2] então
            colorido ← falso
        fim se
    fim para
    se colorido = verdadeiro e |C| ≤ K então
        escreva "SIM"
    senão
        escreva "NAO"
    fim se
fim
```

- Complexidade: $\Theta(n^2)$, onde n é o número de vértices

Algoritmos Não-deterministas

- Um computador não-determinista, quando diante de duas ou mais alternativas, é capaz de produzir cópias de si mesmo e continuar a computação independentemente para cada alternativa
- Um algoritmo não-determinista é capaz de escolher uma dentre as várias alternativas possíveis a cada passo (o algoritmo é capaz de adivinhar a alternativa que leva a solução)

Algoritmos Não-deterministas

- Utilizam
 - a função `escolhe(C)` : escolhe um dos elementos de `C` de forma arbitrária.
 - SUCESSO: sinaliza uma computação com sucesso
 - INSUCESSO: sinaliza uma computação sem sucesso
- Sempre que existir um conjunto de opções que levam a um término com sucesso então, este conjunto é sempre escolhido
- A complexidade da função `escolhe` é $\Theta(1)$

Exemplo: Pesquisa

- Pesquisar o elemento x em um conjunto de elementos $A[1..n], n \geq 1$

```
j ← escolhe (A, x)
se A[j] = x então
    SUCESSO
senão
    INSUCESSO
fim se
```

- Complexidade: $\Theta(1)$
- Para um algoritmo determinista a complexidade é $\Theta(n)$

Exemplo: Ordenação

- Ordenar um conjunto A contendo n inteiros $n \geq 1$

```
NDOrdenacao (A,n)
inicio
  para  $i \leftarrow 1$  até  $n$  faça  $B[i]=0$ ;
  para  $i \leftarrow 1$  até  $n$  faça
    inicio
       $j \leftarrow \text{escolhe}(A,i)$ ;
      se  $B[j] = 0$  então
         $B[j] = A[i]$ ;
      senão
        INSUCESSO
      fim se
    fim para
  SUCESSO
fim
```

- B contém o conjunto ordenado
- A posição correta em B de cada inteiro de A é obtida de forma não-determinista

- Complexidade do algoritmo não-determinista: $\Theta(n)$
- Complexidade do algoritmo determinista: $\Theta(n \log n)$

Algoritmos Deterministas X Algoritmos Não-deterministas

- Classe P (*Polynomial-time Algorithms*)
 - Conjunto de todos os problemas que podem ser resolvidos por algoritmos deterministas em tempo polinomial
- Classe NP (*Nondeterministic Polynomial Time Algorithms*)
 - Conjunto de todos os problemas que podem ser resolvidos por algoritmos não-deterministas em tempo polinomial

Como Mostrar que um Determinado Problema está em NP?

- Basta apresentar um algoritmo não-determinista que execute em tempo polinomial para resolver o problema

ou

- Basta encontrar um algoritmo determinista polinomial para verificar que uma dada solução é válida

Caixeiro Viajante está em NP

- Algoritmo não-determinista em tempo polinomial

```
NDPCV(G, n, k)
inicio
  Soma ← 0
  para i ← 1 até n faça
    A[i] ← escolhe(G, n)
  fim para
  A[n+1] ← A[1]
  para i ← 1 até n faça
    Soma ← Soma + distancia entre A[i] e A[i+1]
  fim para
  se Soma ≤ k então
    SUCESSO
  senão
    INSUCESSO
  fim se
fim
```

- Complexidade do algoritmo não-determinista: $\Theta(n)$
- Complexidade do algoritmo determinista: $\Theta(n^2 \cdot 2^n)$

Caixeiro Viajante está em NP

- Algoritmo determinista polinomial para verificar a solução

```
DPCVV (G, S, n, k)
inicio
    Soma ← 0
    para i ← 1 até n faça
        Soma ← Soma + distancia entre S[i] e S[i+1]
    se Soma ≤ k então
        escreva "SIM"
    senão
        escreva "NAO"
fim
```

- Complexidade: $\Theta(n)$

$P = NP$ ou $P \neq NP$?

- Como algoritmos deterministas são apenas um caso especial de algoritmos não-deterministas, podemos concluir que

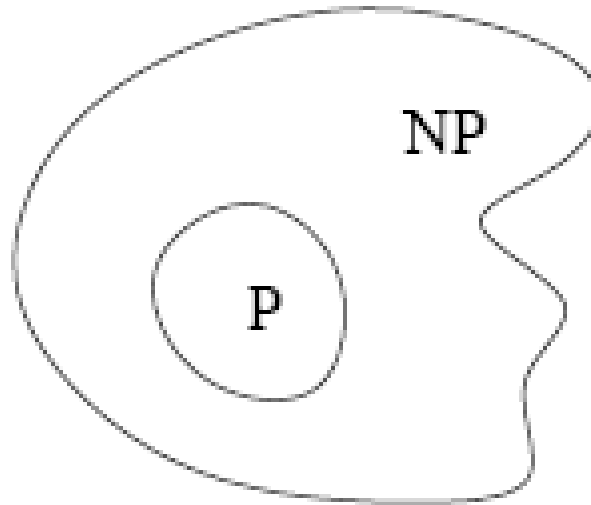
$$P \subseteq NP$$

- O que não sabemos é se

$$P = NP \text{ ou } P \neq NP$$

- Será que existem algoritmos polinomiais deterministas para todos os problemas em NP?
- Por outro lado, a prova de que $P \neq NP$ parece exigir técnicas ainda desconhecidas

Descrição tentativa de NP



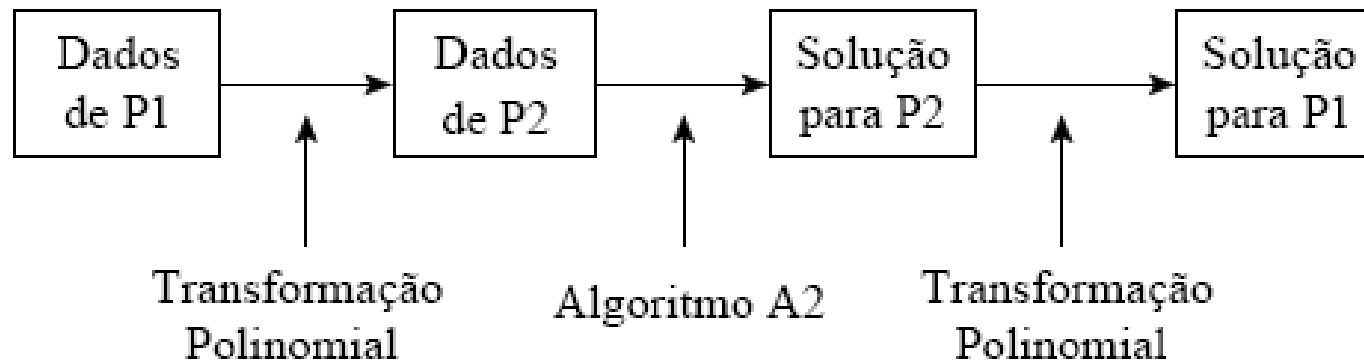
- P está contida em NP
- Acredita-se que NP seja muito maior que P

Consequências

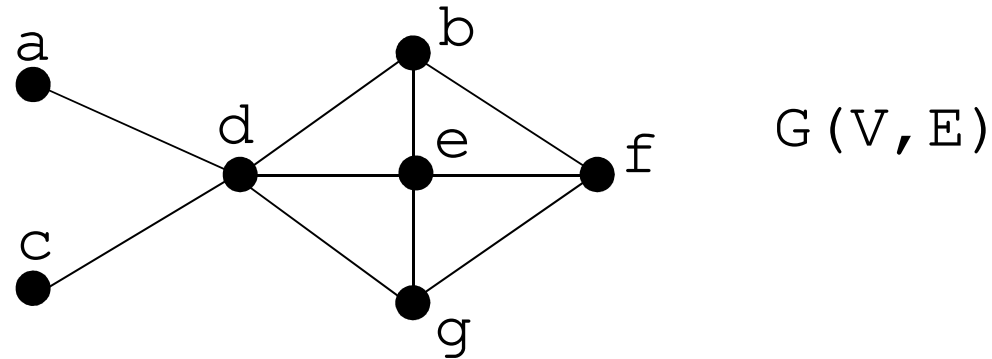
- Existem muitos problemas práticos em NP que podem ou não pertencer a P (não conhecemos nenhum algoritmo eficiente que execute em uma máquina determinista)
- Se conseguirmos provar que um problema não pertence a P, então não precisamos procurar por uma solução eficiente para ele
- Como não existe tal prova sempre há esperança de que alguém descubra um algoritmo eficiente
- Quase ninguém acredita que $NP = P$
- Existe um esforço considerável para provar o contrário: MAS O PROBLEMA CONTINUA EM ABERTO!

Redução Polinomial

- Sejam Π_1 e Π_2 dois problemas “sim/não”.
- Suponha que exista um algoritmo A2 para resolver Π_2 . Se for possível transformar Π_1 em Π_2 e sendo conhecido um processo de transformar a solução de Π_2 numa solução de Π_1 então, o algoritmo A2 pode ser utilizado para resolver Π_1
- Se estas duas transformações puderem ser realizadas em tempo polinomial então, Π_1 é polinomialmente redutível a Π_2 ($\Pi_1 \leq \Pi_2$)



Exemplo de Transformação Polinomial



- Conjunto independente de vértices
 - $V' \subseteq V$ tal que todo par de vértices de V' é não adjacente, ou seja, se $v, w \in V' \Rightarrow (v, w) \notin E$
 - a, c, b, g é um exemplo de um conjunto independente de cardinalidade 4
- Clique
 - $V' \subseteq V$ tal que todo par de vértices de V' é adjacente, V' é um subgrafo completo, ou seja, se $v, w \in V' \Rightarrow (v, w) \in E$
 - d, b, e é um exemplo de um clique de cardinalidade 3

Exemplo de Transformação Polinomial

- Instância I do Clique
 - **Dados:** Grafo $G(V, E)$ e um inteiro $K > 0$
 - **Decisão:** G possui um clique de tamanho $\geq k$?
- Instância $f(I)$ do Conjunto Independente
 - Considere o grafo complementar \bar{G} de G e o mesmo inteiro K , f é uma transformação polinomial porque:
 1. \bar{G} pode ser obtido a partir de G em tempo polinomial
 2. G possui clique de tamanho $\geq k$ se e somente se \bar{G} possui conjunto independente de vértices de tamanho $\geq k$

Exemplo de Transformação Polinomial

Se existir um algoritmo que resolva o conjunto independente em tempo polinomial, este algoritmo pode ser utilizado para resolver o clique também em tempo polinomial

Clique \propto Conjunto Independente

Satisfabilidade

- Definir se uma expressão booleana E contendo produto de adições de variáveis booleanas é satisfatível
 - Exemplo: $(x_1 + x_2) * (x_1 + \overline{x_3} + x_2) * (x_3)$
onde x_i representa variáveis lógicas
+ representa OR
* representa AND
 \overline{x} representa NOT
- **Problema:** Existe uma atribuição de valores lógicos (V ou F) às variáveis que torne a expressão verdadeira (“satisfaça”)?
 $x_1=F, x_2=V, x_3=V$ Satisfaz!
 - Exemplo: $(x_1) * (\overline{x_1})$ não é satisfatível

Satisfabilidade

```
NDAval(E,n)
inicio
  para i<-1 até n faça
    xi <- escolhe(true,false)
  se E(x1,x2,...,xn) = true então
    SUCESSO
  senão
    INSUCESSO
  fim se
fim
```

- O algoritmo obtém uma das 2^n atribuições possíveis de forma não-determinista em $O(n)$

SAT \in NP

Teorema de Cook

- S. Cook formulou a seguinte questão (em 1971): existe algum problema em NP tal que se ele for mostrado estar em P então este fato implicaria que $P=NP$.
- **Teorema de Cook:** Satisfabilidade está em P se, e somente se, $P = NP$
- Isto é: Se existe um algoritmo polinomial determinista para a satisfabilidade então, todos os problemas em NP poderiam ser resolvidos em tempo polinomial

Teorema de Cook

- SAT está em P sse $P = NP$
- Esboço da prova
 1. SAT está em NP. Logo, se $P = NP$ então SAT está em P.
 2. Se SAT está em P então $P = NP$
 - Prova descreve como obter de qualquer algoritmo polinomial não determinista de decisão A com entrada E uma fórmula $Q(A,E)$ de forma que Q é satisfatível se e somente se A termina com sucesso para a entrada E. Isto significa que ele mostrou que para qualquer problema $\Pi \in NP$, $\Pi \leq SAT$

NP-Completo

- Um problema de decisão Π é denominado NP-Completo se
 1. $\Pi \in \text{NP}$
 2. Todo problema de decisão $\Pi' \in \text{NP}$ satisfaz $\Pi' \leq \Pi$
- Apenas problemas de decisão (sim/não) podem ser NP-Completo

Como Provar que um Problema é NP-Completo

1. Mostre que o problema está em NP
2. Mostre que um problema NP-Completo conhecido pode ser polinomialmente transformado para ele

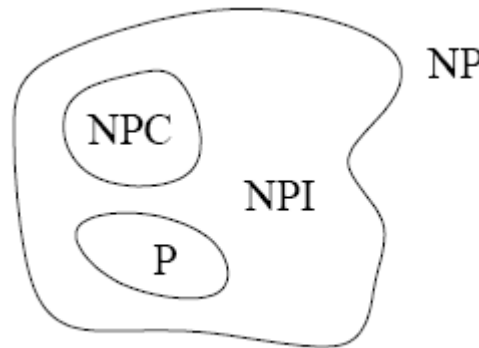
- Porque:

- Cook apresentou uma prova direta de que SAT é NP-Completo
- Transitividade da redução polinomial

$$\text{SAT} \propto \Pi_1 \text{ e } \Pi_1 \propto \Pi_2 \Rightarrow \text{SAT} \propto \Pi_2$$

Descrição tentativa de NP

Assumindo que $P \neq NP$



- Se alguém encontrar um algoritmo polinomial que resolva algum problema NP-Completo então, todos os problemas em NP também terão solução polinomial, ou seja, **P será igual a NP.**
- Se alguém provar que um determinado problema em NP não tem solução polinomial então, todos os problemas em NP-Completo também não terão solução polinomial, ou seja, **P será diferente de NP.**

Qual é a Contribuição Prática da Teoria de NP-Completo?

- Fornece um mecanismo que permite descobrir se um novo problema é “fácil” ou “difícil”
- Se encontrarmos um algoritmo eficiente para o problema, então não há dificuldades. Senão, uma prova de que o problema é NP-Completo nos diz que se acharmos um algoritmo eficiente então estaremos obtendo um grande resultado.

Como Resolver Problemas NP-completos?

- Quando não existe solução polinomial é necessário usar **algoritmos aproximados ou heurísticas** que não garantem a solução ótima mas são rápidos

Algoritmos Aproximados e Heurísticas

- Algoritmos aproximados:
 - Algoritmos usados normalmente para resolver problemas para os quais não se conhece uma solução polinomial
 - Devem executar em tempo polinomial dentro de limites “prováveis” de qualidade absoluta ou assintótica
- Heurísticas:
 - Algoritmos que têm como objetivo fornecer soluções sem um limite formal de qualidade, em geral avaliado empiricamente em termos de complexidade (média) e qualidade de soluções
 - É projetada para obter ganho computacional ou simplicidade conceitual, possivelmente ao custo de precisão