Introdução à Programação Competitiva

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação I

Instituto de Computação - UNICAMP - 2018

Sumário

1. Programação competitiva
2. Tornando-se um programador competitivo
3. Estruturas de dados e bibliotecas
4. Estruturas de dados lineares
   - Vetores estáticos
   - Vetores dinâmicos
   - Ordenação e busca
   - Vetor de booleanos
   - Máscaras de bits
   - Pilhas
   - Filas
   - Deques
5. Estruturas de dados não-lineares
   - Árvores de busca binárias balanceadas
   - Heaps
6. Referências

Conceito

Programação competitiva consiste em resolver problemas conhecidos de computação sob restrições de tempo de implementação, tempo de execução e memória.

- Problemas conhecidos: em programação competitiva, os problemas são conhecidos e já foram resolvidos pela comunidade acadêmica; portanto, não se tratam de problemas que estão na fronteira da ciência.
- Tempo de implementação: trata-se de um dos elementos competitivos, presentes nas competições de programação.
- Tempo de execução e uso de memória: restrições frequentemente presentes nos problemas de programação competitiva.

Dica 1: Desenvolver habilidade em digitação

- Quando dois competidores conseguem resolver o mesmo número de problemas, o mais competitivo será aquele que possuir a melhor habilidade em programar (códigos concisos e robustos) e digitar.
- É comum observar em competições de programação times cujas classificações estão separadas por apenas alguns minutos.
- Alguns competidores podem perder pontos importantes porque não conseguiram implementar uma solução de força-bruta nos últimos minutos.
- Teste de digitação: http://www.typingtest.com
- Familiarize-se com as posições dos caracteres utilizados com frequência em linguagens de programação, como `()[]<>;:'"&|!`

Dica 2: Identificar o tipo de problema

- Determinar a qual categoria o problema melhor se enquadra (vide tabela abaixo).

No	Category	In This Book	Frequency
1.	Ad Hoc	Section 1.4	1-2
2.	Complete Search (Iterative/Recursive)	Section 3.2	1-2
3.	Divide and Conquer	Section 3.3	0-1
4.	Greedy (usually the original ones)	Section 3.4	0-1
5.	Dynamic Programming (usually the original ones)	Section 3.5	1-3
6.	Graph	Chapter 4	1-2
7.	Mathematics	Chapter 5	1-2
8.	String Processing	Chapter 6	1
9.	Computational Geometry	Chapter 7	1
10.	Some Harder/Rare Problems	Chapter 8-9	1-2
		Total in Set	8-17 ($\approx\leq 12$)

Fonte: Halim e Halim, 2013.

Dica 2: Identificar o tipo de problema

- Para ser competitivo, um programador deve ser capaz de classificar um problema em uma categoria para a qual ele já resolveu muitos outros problemas, o que o tornará capaz de resolvê-lo em pouco tempo.
- Para adquirir essa habilidade é necessário que o programador adquira um bom volume e diversidade de problemas resolvidos. Essa cultura é obtida com esforço e auto-didatismo.
- Para ser bem-sucedido nos desafios de programação, também é necessário desenvolver proficiência na resolução de novos problemas, que "parecem" não se enquadrar em nenhuma categoria. Em geral, isso envolve reduzir o novo problema a um problema conhecido, identificar características e propriedades que facilitam o problema, atacar o problema por uma abordagem diferenciada.

Dica 3: Fazer análise de algoritmos

- Normalmente há diversas formas de resolver um mesmo problema, mas muitas delas podem não ser rápidas o suficiente, enquanto outras podem ser como “canhões para matar uma mosca”.
- Uma vez definido o algoritmo a ser implementado para resolver um problema, devemos nos perguntar se a maior entrada possível (geralmente descrita no enunciado) vai ser resolvida considerando as restrições de tempo de execução e memória utilizada.
- Uma boa estratégia consiste em escolher o algoritmo mais simples que resolva o problema, respeitando as restrições impostas. É comum o algoritmo mais simples ser o menos eficiente, mas se ele atende a restrição de tempo de execução, não há porque não utilizá-lo.

Tempo de execução $T(n)$

- Um mesmo problema pode ser resolvido por diferentes algoritmos e estruturas de dados, cujas eficiências em tempo de execução podem variar significativamente.
- O hardware, sistema operacional e linguagem de programação são fatores que podem alterar o tempo de execução de um programa.
- No entanto, o tempo de execução de um programa está em geral mais fortemente atrelado a dois principais fatores:

1. Tamanho da entrada do programa n.
2. Complexidade do algoritmo.

- Seja $T(n)$ o tempo de execução de pior caso de um programa, ou seja, o tempo máximo de execução sobre todas as possíveis entradas de tamanho n.
- Ao utilizar o pior caso, temos uma garantia de tempo de execução máximo do algoritmo.

Impacto da complexidade de um algoritmo

Considere o problema de ordenação de um vetor de n elementos.

1. Insertion sort: requer no pior caso na ordem de n^2 comparações.
2. Merge sort: requer no pior caso na ordem de $n \log n$ comparações.

Experimento computacional

- Considere o seguinte experimento computacional para a comparação da eficiencia na execução de dois algoritmos de ordenação para um vetor com $n = 10^6$ elementos:
- Cenário 1:
  - Computador A: executa 10^9 instruções por segundo.
  - Algoritmo 1 (insertion sort): requer a execução de $2n^2$ instruções.
  - Tempo de execução (segundos): $T(n) = \frac{2 \cdot 10^{12}}{10^9} = 2000$
- Cenário 2:
  - Computador B: executa 10^7 instruções por segundo.
  - Algoritmo 2 (merge sort): requer a execução de $50n \cdot \log n$ instruções.
  - Tempo de execução (segundos): $T(n) \approx \frac{50 \cdot 10^6 \cdot 20}{10^7} = 100$

Tempo de execução $T(n)$

Considere que os seguintes trechos de códigos recebem uma entrada de tamanho n:

(A)

```
x += 1;
```

(B)

```
for (i = 0; i < n; i++)
    x += 1;
```

(C)

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        x += 1;
```

Tempo de execução $T(n)$

Quantidade de instruções (atribuições, comparações e operações aritméticas) em função de n:

- (A):
- (B):
- (C):

Definição

A complexidade

(A)

```
x += 1;
```

(B)

```
for (i = 0; i < n; i++)
    x += 1;
```

(C)

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        x += 1;
```

Tempo de execução $T(n)$

Quantidade de instruções (atribuições, comparações e operações aritméticas) em função de n:

- (A): $T_A(n) = 2$
- (B): $T_B(n) = 5n + 2$
- (C): $T_C(n) = 5n^2 + 5n + 2$

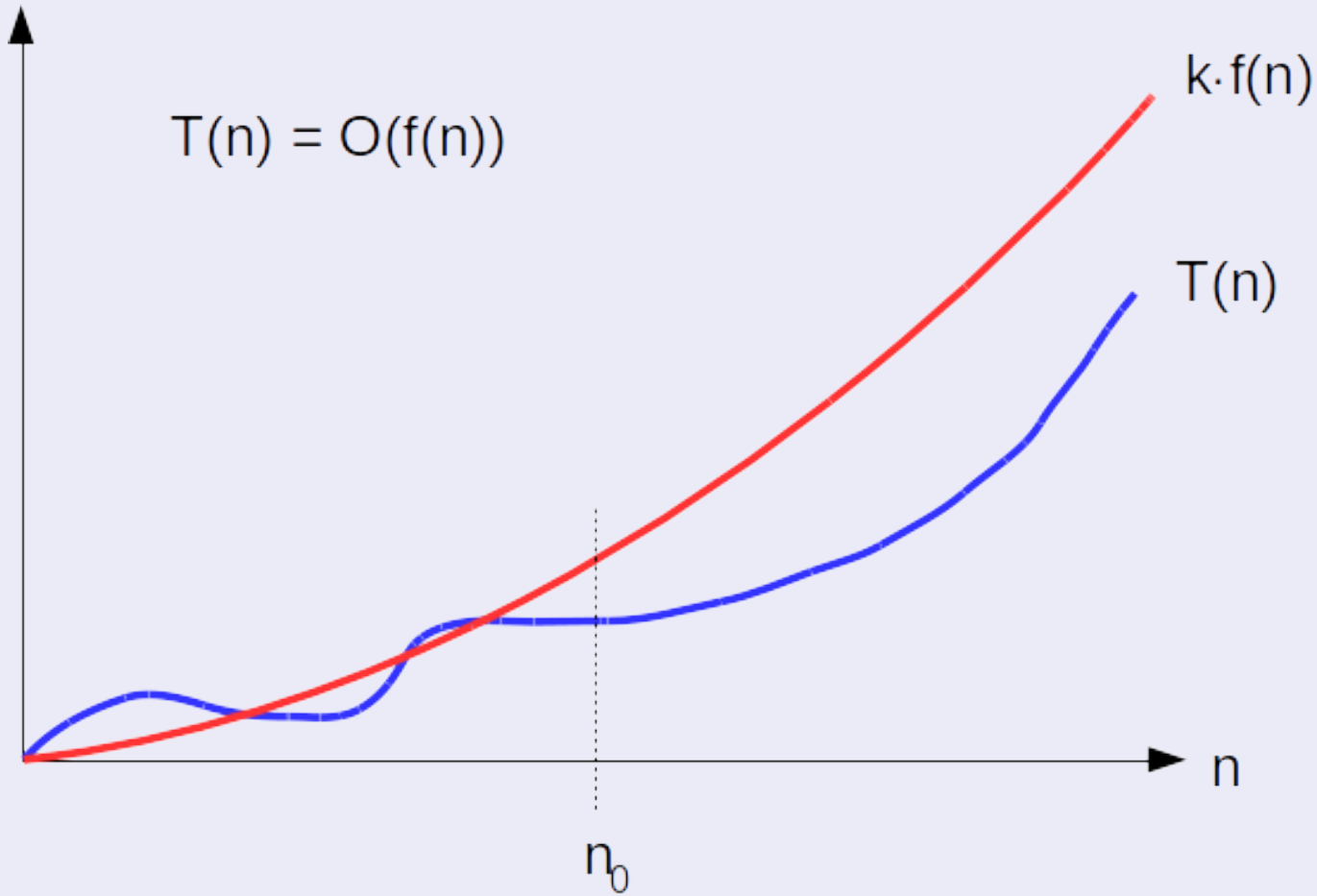
Complexidade assintótica de $T(n)$

- A complexidade assintótica de um algoritmo diz respeito ao seu tempo de execução para entradas suficientemente grandes.

Notação $O()$

- O tempo de execução $T(n)$ de um algoritmo é de ordem $O(f(n))$, ou simplesmente $T(n) = O(f(n))$, se e somente se existem constantes $k > 0$ e $n_0 > 0$ tal que $0 \leqslant T(n) \leqslant k \cdot f(n)$ para todo $n \geqslant n_0$.
- Em outras palavras, quando $T(n) = O(f(n))$, o tempo de execução do algoritmo é limitado superiormente por uma função de ordem $f(n)$ para instâncias suficientemente grandes.

Exemplo



Definição

A complexidade

(A)

```
x += 1;
```

(B)

```
for (i = 0; i < n; i++)
    x += 1;
```

(C)

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        x += 1;
```

Tempo de execução $T(n)$

Quantidade de instruções (atribuições, comparações e operações aritméticas) em função de n:

- (A): $T_A(n) = 2 \Rightarrow T_A(n) = O(1)$
- (B): $T_B(n) = 5n + 2 \Rightarrow T_B(n) = O(n)$
- (C): $T_C(n) = 5n^2 + 5n + 2 \Rightarrow T_C(n) = O(n^2)$

Dica 3: Análise de algoritmos

Regras gerais para análise de tempo de algoritmos iterativos e recursivos:

- Um algoritmo com k laços aninhados, cada um com n iterações, terá complexidade $O(n^k)$.
- Um algoritmo recursivo com b chamadas recursivas em cada um dos L níveis, terá uma complexidade $O(b^L)$ (na verdade pode ser muito melhor do que isso).
- Um algoritmo de programação dinâmica, ou outra rotina iterativa, que processa uma matriz $n \times n$, consumindo um tempo $O(k)$ em cada célula, terá complexidade $O(kn^2)$.

Essas regras estão longe de esgotar o assunto de complexidade de algoritmos. Para uma visão mais detalhada, consultar as referências bibliográficas da disciplina.

Dica 3: Análise de algoritmos

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Fonte: Halim e Halim, 2013.

Dica 3: Análise de algoritmos

- Computadores pessoais modernos executam na ordem de 30 milhões de instruções por segundo (3×10^7) (*Intel Core i7 3770k*). É possível utilizar essa informação para avaliar se um algoritmo pode terminar antes do limite de tempo de execução.
- Exemplo: Suponha um algoritmo de complexidade $O(n \log_2 n)$ para um problema com entrada máxima de tamanho $n = 10^3$. Sabendo que $10^3 \log_2 10^3 \approx 10^4$, é possível presumir com alguma segurança que o algoritmo deverá executar em menos de um segundo.

Dica 3: Análise de algoritmos

Informações úteis:

- $2^{10} = 1.024 \approx 10^3$, $2^{20} = 1.048.576 \approx 10^6$
- o tipo `int` (C++, Java) ocupa 32 bits e pode representar até o número $2^{31} - 1 \approx 2 \times 10^9$.
- o tipo `long long` (C++; somente `long` em Java) ocupa 64 bits e pode representar até o número $2^{63} - 1 \approx 9 \times 10^{18}$.
- Há $n!$ permutações e 2^n subconjuntos de n elementos.
- Algoritmos com complexidade $O(n \log_2 n)$ são suficientes para passar por qualquer problema em programação competitiva (exceto raras exceções).
- A maior entrada de um problema em competições é raramente maior do que 1.000.000.
- Os computadores pessoais atuais executam na ordem de 30 milhões de instruções (3×10^7) por segundo (*Intel Core i7 3770k*).

Dica 4: Tornar-se fluente em linguagens de programação

- Linguagens de programação permitidas em competições oficiais incluem C, C++ e Java.
- Um programa Java é, em geral, considerado mais lento do que um programa em C++. Por outro lado:
  - Um código em Java é mais fácil de ser depurado, dado que a máquina virtual providencia toda a pilha de execução após uma exceção.
  - A linguagem Java possui um conjunto de APIs como `Regex`, `BigInteger`, `GregorianCalendar`, dentre outras que podem ser muito úteis para certos tipos de problema.

Dica 4: Tornar-se fluente em linguagens de programação

Exemplo: problema cuja solução em Java torna-se muito vantajosa.

- Escreva um programa que calcule o fatorial de 25.
- $25! = 15.511.210.043.330.985.984.000.000$

```
import java.util.Scanner;
import java.math.BigInteger;

class Main {     // standard Java class name in UVa OJ
  public static void main(String[] args) {
    BigInteger fac = BigInteger.ONE;
    for (int i = 2; i <= 25; i++)
      fac = fac.multiply(BigInteger.valueOf(i));
    System.out.println(fac);
  }
}
```

Dica 5: Aperfeiçoar sua habilidade em depurar códigos

- Ao submeter um código a um corretor automático, desejamos receber como veredito: Accepted (AC).
- Outros possíveis vereditos (indesejáveis) são:
  1. Presentation Error (PE) – a saída do seu programa está correta, porém não está formatada como especificado no enunciado.
  2. Wrong Answer (WA) – a saída do seu programa está incorreta em pelo menos um dos casos de teste.
  3. Time Limit Exceeded (TLE) – o seu programa esgotou a restrição de tempo de execução.
  4. Memory Limit Exceeded (MLE) – o seu programa esgotou a restrição de memória utilizada.
  5. Run Time Error (RTE) – o seu programa foi interrompido devido a um erro em tempo de execução.

Dica 5: Aperfeiçoar sua habilidade em depurar códigos

Regras gerais para desenvolver bons testes de depuração de código:

- Seus testes devem incluir os testes de exemplo, geralmente presentes no enunciado. Utilize o comando `fc` em Windows ou `diff` em Linux para comparar se a saída do seu programa está igual à saída do enunciado.
- Para problemas com múltiplos casos de testes, inclua duas cópias consecutivas de um mesmo caso de teste. Se as saídas diferirem entre si, torna-se um indício de que as variáveis não foram inicializadas corretamente.
- Testar condições de contorno ($N = 0$, $N = 1$, $N < 0$, $N > 2^{32}$, etc.).
- Testar casos grandes, tanto com estruturas triviais (fáceis de avaliar) como estruturas geradas aleatoriamente (para avaliar robustez).

Dica 6: Praticar, praticar e praticar

- Assim como atletas de competição, um programador competitivo deve treinar regularmente para manter-se em "boa forma", ou seja, para que sua cultura em problemas de programação mantenha-se fresca na memória.
- Há diversos "juízes online" na internet para aprimorar suas habilidades em programação. Alguns deles são:
  1. http://uva.onlinejudge.org/
  2. http://livearchive.onlinejudge.org/
  3. http://www.spoj.com/
  4. http://br.spoj.com/
  5. http://www.urionlinejudge.com.br/
  6. http://acm.timus.ru/
  7. http://poj.org/
  8. http://acm.zju.edu.cn/onlinejudge/

Introdução

- Uma estrutura de dados consiste em um meio de armazenar, organizar, atualizar e recuperar informações.
- Diferentes estruturas de dados possuem complexidades distintas para operações como busca, inserção, remoção e atualização.
- Uma estrutura não resolve um problema de programação por si só, mas a escolha de uma estrutura de dados apropriada pode ser a diferença entre passar ou não na restrição de tempo de execução.

Introdução

- Assume-se que o leitor tenha familiaridade com estruturas de dados elementares vistas em um curso de graduação.
- Serão destacadas implementações dessas estruturas de dados na biblioteca STL (Standard Template Library) de C++.
- Para visualizar o comportamento dessas estruturas, consulte o site abaixo:

www.comp.nus.edu.sg/~stevenha/visualization

Exemplos

Serão descritas as seguintes estruturas de dados lineares:

- Vetores estáticos – suporte nativo em C/C++ e Java.
- Vetores dinâmicos – C++ STL `vector` (Java `ArrayList`).
- Vetores booleanos – C++ STL `bitset` (Java `BitSet`)
- Máscaras de bits – suporte nativo em C/C++ e Java.
- Listas ligadas – C++ STL `list` (Java `LinkedList`)
- Pilhas – C++ STL `stack` (Java `Stack`)
- Filas – C++ STL `queue` (Java `Queue`)
- Deques – C++ STL `deque` (Java `Deque`)

Vetores estáticos (suporte nativo em C/C++ e Java)

- Os vetores estáticos são as estruturas de dados mais utilizadas em competições de programação.
- Trata-se da estrutura de dados natural para armazenar uma coleção de dados sequenciais que podem ser recuperados diretamente pelo índice.
- Como o tamanho máximo de uma entrada é normalmente mencionado no enunciado, o vetor pode ser dimensionado já prevendo o uso em sua máxima capacidade.
- Operações usuais com vetores estáticos consistem em acesso randômico, ordenações e buscas binárias (vetor pré-ordenado).

Vetores dinâmicos (C++ STL `vector`, Java `ArrayList`)

- Similar à versão estática, os vetores dinâmicos foram desenvolvidos para realizar o redimensionamento automático de um vetor.
- É vantajoso nas ocasiões onde não se sabe, em tempo de compilação, o número de elementos que serão armazenados.
- Para uma melhor performance é possível utilizar o método `reserve()` com uma estimativa do tamanho do vetor.
- Operações típicas em um objeto `vector` incluem `push_back()`, `at()`, `[]`, `assign()`, `clear()`, `erase()` e `iterators` que são utilizados para realizar percursos sobre os elementos armazenados.

Exemplo (vetores)

```
#include <cstdio>
#include <vector>
using namespace std;

int main() {
  int arr[5] = {7,7,7};   // initial size (5) and initial value {7,7,7,0,0}
  vector<int> v(5, 5);    // initial size (5) and initial value {5,5,5,5,5}

  printf("arr[2] = %d and v[2] = %d\n", arr[2], v[2]);          // 7 and 5

  for (int i = 0; i < 5; i++) {
    arr[i] = i;
    v[i] = i;
  }

  printf("arr[2] = %d and v[2] = %d\n", arr[2], v[2]);          // 2 and 2

  // arr[5] = 5;      // static array will generate index out of bound error
  // uncomment the line above to see the error

  v.push_back(5);                              // but vector will resize itself
  printf("v[5] = %d\n", v[5]);                                  // 5

  return 0;
}
```

Ordenação e Busca

- Duas operações muito usuais em vetores são Ordenação e Busca. Essas operações já estão implementadas em APIs de C++ e Java. Dentre os algoritmos de ordenação conhecidos, temos:

1. Algoritmos $O(n^2)$ baseados em comparação: Bubblesort, Selection Sort, Insertion Sort, etc. Normalmente devem ser evitados em competição por serem lentos, mas compreendê-los pode auxiliar na solução de certos problemas.

Ordenação e Busca

1. Algoritmos $O(n \log n)$ baseados em comparação: Heapsort, Quicksort (*caso médio*), Mergesort, etc. A C++ STL possui os métodos `sort, stable_sort` da classe `algorithm` (`Collections.sort em Java`). Para utilizar esses métodos, é possível definir uma função comparadora.
2. Algoritmos de propósito específico $O(n)$: Counting sort, Radix sort, Bucket sort, etc. Esses algoritmos presumem características específicas sobre os valores a serem ordenados para reduzir a complexidade do algoritmo.

Ordenação e Busca

Para buscar um elemento em um vetor, é possível realizar:

1. Busca linear $O(n)$: percorrer todos os elementos do vetor. Esse método deve ser evitado, exceto quando trata-se de uma quantidade pequena de buscas em um vetor não-ordenado.
2. Busca binária $O(\log n)$: avaliar a posição na metade de um vetor ordenado. Se não encontrou o elemento desejado, continue a busca recursivamente na metade (esquerda ou direita) em que o elemento pode se encontrar. Essa busca está implementada em C++ a partir dos métodos `lower_bound`, `upper_bound`, `binary_search` da classe `algorithm` (`Collections.binarySearch` em Java).

Exemplo (busca e ordenação). continua...

```
#include <algorithm>
#include <cstdio>
#include <string>
#include <vector>
using namespace std;

typedef struct {
  int id;
  int solved;
  int penalty;
} team;

bool icpc_cmp(team a, team b) {
  if (a.solved != b.solved) // can use this primary field to decide sorted order
    return a.solved > b.solved;   // ICPC rule: sort by number of problem solved
  else if (a.penalty != b.penalty)       // a.solved == b.solved, but we can use
                                         // secondary field to decide sorted order
    return a.penalty < b.penalty;       // ICPC rule: sort by descending penalty
  else                            // a.solved == b.solved AND a.penalty == b.penalty
    return a.id < b.id;                          // sort based on increasing team ID
}
```

Exemplo (busca e ordenação). continua...

```
int main() {
  int *pos, arr[] = {10, 7, 2, 15, 4};
  vector<int> v(arr, arr + 5);          // another way to initialize vector
  vector<int>::iterator j;

  // sort ascending with vector
  sort(v.begin(), v.end());     // ascending
  //reverse(v.begin(), v.end());  // for descending, uncomment this line
  for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    printf("%d ", *it);                    // access the value of iterator
  printf("\n");
  printf("==================\n");

  // sort ascending with integer array
  sort(arr, arr + 5);                                    // ascending
  //reverse(arr, arr + 5);          // for descending, uncomment this line
  for (int i = 0; i < 5; i++)
    printf("%d ", arr[i]);
  printf("\n");
  printf("==================\n");
```

Exemplo (busca e ordenação). continua...

```
// binary search using lower bound
pos = lower_bound(arr, arr + 5, 7);                    // found
printf("%d\n", *pos);
j = lower_bound(v.begin(), v.end(), 7);
printf("%d\n", *j);

pos = lower_bound(arr, arr + 5, 77);                   // not found
if (pos == arr + 5 || arr[pos] != 77)
  printf("77 not found\n");

j = lower_bound(v.begin(), v.end(), 77);               // not found
if (j == v.end() || *j != 77)
  printf("77 not found\n");
printf("==================\n");
```

Exemplo (busca e ordenação)

```
// multi-field sorting example, suppose we have 4 ICPC teams
team nus[4] = { {1, 1, 10},
                {2, 3, 60},
                {3, 1, 20},
                {4, 3, 60} };

// without sorting, they will be ranked like this:
for (int i = 0; i < 4; i++)
  printf("id: %d, solved: %d, penalty: %d\n",
         nus[i].id, nus[i].solved, nus[i].penalty);

sort(nus, nus + 4, icpc_cmp);          // sort using a comparison function
printf("==================\n");
// after sorting using ICPC rule, they will be ranked like this:
for (int i = 0; i < 4; i++)
  printf("id: %d, solved: %d, penalty: %d\n",
         nus[i].id, nus[i].solved, nus[i].penalty);
printf("==================\n");
```

Exemplo (busca e ordenação)

```
  // useful if you want to generate permutations of set
  next_permutation(arr, arr + 5); // 2, 4, 7, 10, 15 -> 2, 4, 7, 15, 10
  next_permutation(arr, arr + 5); // 2, 4, 7, 15, 10 -> 2, 4, 10, 7, 15
  for (int i = 0; i < 5; i++)
    printf("%d ", arr[i]);
  printf("\n");

  next_permutation(v.begin(), v.end());
  next_permutation(v.begin(), v.end());
  for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    printf("%d ", *it);
  printf("\n");
  printf("==================\n");

  return 0;

} // end of main
```

Exemplo (busca e ordenação)

Saída:

```
2 4 7 10 15
==================
2 4 7 10 15
==================
7
7
77 not found
77 not found
==================
id: 1, solved: 1, penalty: 10
id: 2, solved: 3, penalty: 60
id: 3, solved: 1, penalty: 20
id: 4, solved: 3, penalty: 60
==================
id: 2, solved: 3, penalty: 60
id: 4, solved: 3, penalty: 60
id: 1, solved: 1, penalty: 10
id: 3, solved: 1, penalty: 20
==================
2 4 10 7 15
2 4 10 7 15
==================
```

Vetor de booleanos

- Quando um vetor precisa conter somente valores booleanos, uma estrutura de dados eficiente consiste no tipo `bitset` de C++ STL.
- Essa estrutura é eficiente em termos de uso de memória, dado que cada campo ocupa somente um bit de memória.
- Essa estrutura de dados suporta operações como `reset()`, `set()`, `[]`, `test()`.

Exemplo (vetor de booleanos)

```
// bitset operators
#include <iostream>       // cout
#include <string>         // string
#include <bitset>         // bitset
using namespace std;

int main () {
  bitset<4> foo (9); // 1001
  bitset<4> bar (string("0011"));

  cout << (foo&bar) << '\n';         // 0001 (AND)
  cout << (foo|bar) << '\n';         // 1011 (OR)
  cout << (foo^bar) << '\n';         // 1010 (XOR)
  cout << (~bar) << '\n';            // 1100 (NOT)
  cout << (bar<<1) << '\n';          // 0110 (SHL)
  cout << (bar>>1) << '\n';          // 0001 (SHR)
  cout << (foo==bar) << '\n';        // false (0110==0011)
  cout << (foo!=bar) << '\n';        // true  (0110!=0011)

  return 0;
}
```

Máscaras de bits

- Uma máscara de bits consiste em um conjunto pequeno de booleanos, que podem ser tratados de modo nativo (C/C++/Java).
- Um número inteiro é armazenado em memória como uma cadeia de bits. Portanto, é possível utilizar números inteiros para representar pequenos conjuntos de valores booleanos.
- Todas as operações de conjuntos envolvem somente manipulação nos bits do número inteiro correspondente, o que torna essa estrutura muito eficiente.
- Muitas operações de manipulação de bits podem ser escritas como macros em C/C++.

Exemplo (máscaras de bits). continua...

```
#include <cmath>
#include <cstdio>
#include <stack>
using namespace std;

#define isOn(S, j) (S & (1 << j))
#define setBit(S, j) (S |= (1 << j))
#define clearBit(S, j) (S &= ~(1 << j))
#define toggleBit(S, j) (S ^= (1 << j))
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1 << n) - 1)

void printSet(int vS) {                          // in binary representation
  printf("S = %2d = ", vS);
  stack<int> st;
  while (vS)
    st.push(vS % 2), vS /= 2;
  while (!st.empty())                           // to reverse the print order
    printf("%d", st.top()), st.pop();
  printf("\n");
}
```

Exemplo (máscaras de bits). continua...

```
int main() {
  int S, T;

  printf("1. Representation (all indexing are 0-based and counted from right)\n");
  S = 34; printSet(S);
  printf("\n");

  printf("2. Multiply S by 2, then divide S by 4 (2x2), then by 2\n");
  S = 34; printSet(S);
  S = S << 1; printSet(S);
  S = S >> 2; printSet(S);
  S = S >> 1; printSet(S);
  printf("\n");

  printf("3. Set/turn on the 3-th item of the set\n");
  S = 34; printSet(S);
  setBit(S, 3); printSet(S);
  printf("\n");

  printf("4. Check if the 3-th and then 2-nd item of the set is on?\n");
  S = 42; printSet(S);
  T = isOn(S, 3); printf("T = %d, %s\n", T, T ? "ON" : "OFF");
  T = isOn(S, 2); printf("T = %d, %s\n", T, T ? "ON" : "OFF");
  printf("\n");
```

Exemplo (máscaras de bits).

```
  printf("5. Clear/turn off the 1-st item of the set\n");
  S = 42; printSet(S);
  clearBit(S, 1); printSet(S);
  printf("\n");

  printf("6. Toggle the 2-nd item and then 3-rd item of the set\n");
  S = 40; printSet(S);
  toggleBit(S, 2); printSet(S);
  toggleBit(S, 3); printSet(S);
  printf("\n");

  printf("7. Check the first bit from right that is on\n");
  S = 40; printSet(S);
  T = lowBit(S); printf("T = %d (this is always a power of 2)\n", T);
  S = 52; printSet(S);
  T = lowBit(S); printf("T = %d (this is always a power of 2)\n", T);
  printf("\n");

  printf("8. Turn on all bits in a set of size n = 6\n");
  setAll(S, 6); printSet(S);
  printf("\n");

  return 0;
}
```

Exemplo (máscaras de bits)

Saída (continua...):

```
1. Representation (all indexing are 0-based and counted from right)
S = 34 = 100010

2. Multiply S by 2, then divide S by 4 (2x2), then by 2
S = 34 = 100010
S = 68 = 1000100
S = 17 = 10001
S =  8 = 1000

3. Set/turn on the 3-th item of the set
S = 34 = 100010
S = 42 = 101010

4. Check if1 the 3-th 'and then 2-nd item of the set is on?
S = 42 = 101010
T = 8, ON
T = 0, OFF

5. Clear/turn off the 1-st item of the set
S = 42 = 101010
S = 40 = 101000
```

Exemplo (máscaras de bits)

Saída:

```
6. Toggle the 2-nd item and then 3-rd item of the set
S = 40 = 101000
S = 44 = 101100
S = 36 = 100100

7. Check the first bit from right that is on
S = 40 = 101000
T = 8 (this is always a power of 2)
S = 52 = 110100
T = 4 (this is always a power of 2)

8. Turn on all bits in a set of size n = 6
S = 63 = 111111
```

Pilhas (C++ STL `stack`, Java `Stack`)

- A pilha é uma estrutura de dados utilizada para o tratamento de diversos problemas, dentre os quais: cálculo e conversões de notação pós-fixa, infixa e pré-fixa, encontrar componentes fortemente conexas em grafos, encontrar caminhos eulerianos em grafos.
- Uma pilha admite operações de inserção e remoção em tempo $O(1)$ a partir do topo da pilha, o que confere seu comportamento como "Last In First Out" (LIFO).
- Na biblioteca C++, as operações em pilha são chamadas pelos métodos `push()`, `pop()`, `empty()`, `top()`.

Filas (C++ STL `queue`, Java `Queue`)

- Uma fila é também uma estrutura de dados muito comum. Exemplos: simulação de fenômenos que obedecem a política “First In First Out” (FIFO), como impressoras, chamadas de call center, tratamento de eventos, dentre outras aplicações como busca em largura em grafos.
- Uma fila admite operações de inserção (no fim) e remoção (no início) em tempo $O(1)$.
- Na biblioteca C++, as operações em fila são chamadas pelos métodos `push()`, `pop()`, `front()`, `back()`, `empty()`.

Deques (C++ STL `deque`, Java `Deque`)

- Um deque generaliza a funcionalidade de pilhas e filas.
- Essa estrutura de dados admite operações de inserção e remoção (no início e no fim do deque) em tempo $O(1)$.
- Na biblioteca C++, as operações em deque são chamadas pelos métodos `push_back()`, `pop_front()`, `push_front()`, `pop_back()`, `empty()`.

Exemplo (Pilhas, Filas e Deques). continua...

```
#include <cstdio>
#include <stack>
#include <queue>
using namespace std;

int main() {
  stack<char> s;
  queue<char> q;
  deque<char> d;

  printf("%d\n", s.empty());                    // currently s is empty, true (1)
  printf("==================\n");
  s.push('a');
  s.push('b');
  s.push('c');
  // stack is LIFO, thus the content of s is currently like this:
  // c <- top
  // b
  // a
  printf("%c\n", s.top());                                  // output 'c'
  s.pop();                                                 // pop topmost
  printf("%c\n", s.top());                                  // output 'b'
  printf("%d\n", s.empty());          // currently s is not empty, false (0)
  printf("==================\n");
```

Exemplo (Pilhas, Filas e Deques). continua...

```
printf("%d\n", q.empty());                  // currently q is empty, true (1)
printf("==================\n");
while (!s.empty()) {                        // stack s still has 2 more items
  q.push(s.top());                              // enqueue 'b', and then 'a'
  s.pop();
}
q.push('z');                                            // add one more item
printf("%c\n", q.front());                                      // prints 'b'
printf("%c\n", q.back());                                       // prints 'z'

// output 'b', 'a', then 'z' (until queue is empty), according to the insertion
   order above
printf("==================\n");
while (!q.empty()) {
  printf("%c\n", q.front());                            // take the front first
  q.pop();                                  // before popping (dequeue-ing) it
}
```

Exemplo (Pilhas, Filas e Deques). continua...

```
    printf("==================\n");
    d.push_back('a');
    d.push_back('b');
    d.push_back('c');
    printf("%c - %c\n", d.front(), d.back());              // prints 'a - c'
    d.push_front('d');
    printf("%c - %c\n", d.front(), d.back());              // prints 'd - c'
    d.pop_back();
    printf("%c - %c\n", d.front(), d.back());              // prints 'd - b'
    d.pop_front();
    printf("%c - %c\n", d.front(), d.back());              // prints 'a - b'

    return 0;
}
```

Exemplo (Pilhas, Filas e Deques)

Saída:

```
1
==================
c
b
0
==================
1
==================
b
z
==================
b
a
z
==================
a — c
d — c
d — b
a — b
```

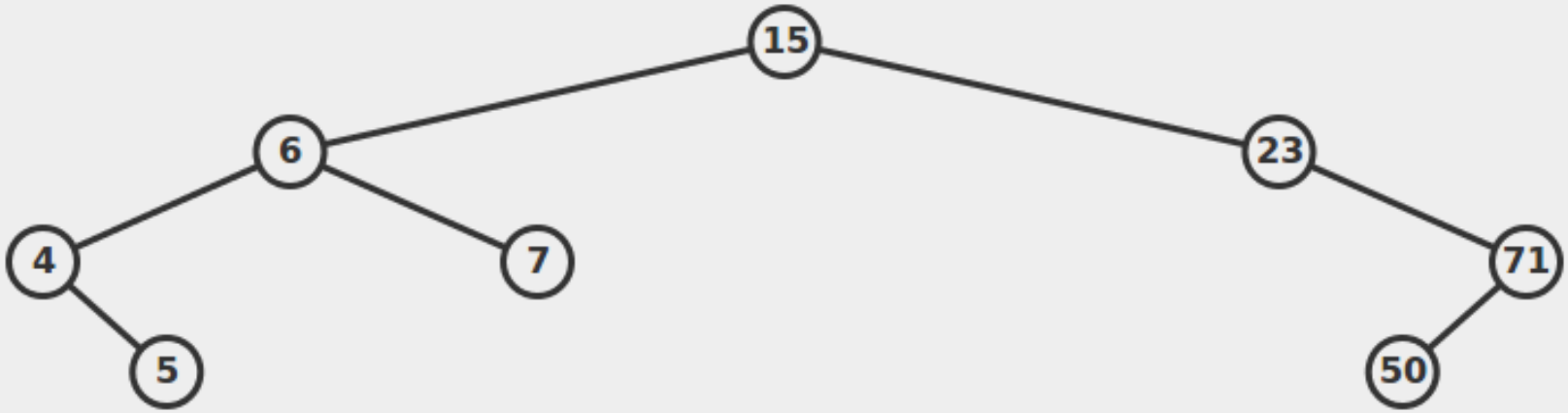
Estruturas de dados não-lineares

Informações não-sequenciais

- Nem sempre uma estrutura linear é conveniente para um problema.
- Serão discutidas duas estruturas não-lineares: árvores de busca binárias balanceadas e fila de prioridades.

Árvores de busca binárias balanceadas (C++ STL `map/set`, Java `TreeMap/TreeSet`)

- Uma árvore de busca binária possui a seguinte propriedade: para cada subárvore enraizada em um nó x, os nós à esquerda de x são menores do que x, enquanto os nós à direita são maiores do que x.



- A árvore binária de busca é considerada balanceada quando sua altura é assintoticamente limitada por uma função logarítmica do número de nós $h = O(\log n)$.
- Quando a árvore é balanceada, as operações de busca, inserção, máximo, mínimo, sucessor, predecessor e remoção passam a ter complexidade $O(\log n)$

Árvores de busca binárias balanceadas (C++ STL `map/set`, Java `TreeMap/TreeSet`)

- As classes `map` e `set` da C++ STL (`TreeMap, TreeSet` em Java) são implementações de árvores rubro-negras, que correspondem a um tipo de árvores de busca binárias balanceadas.
- A diferença entre as classes `map` e `set` é de que a primeira armazena pares de chave e valor, enquanto a segunda armazena somente chaves.

Exemplo (Árvores de busca binárias balanceadas). continua...

```
#include <cstdio>
#include <map>
#include <set>
#include <string>
using namespace std;

int main() {
  char name[20];
  int value;
  // note: there are many clever usages of this set/map
  // that you can learn by looking at top coder's codes
  // note, we don't have to use .clear() if we have just initialized the set/map
  set<int> used_values; // used_values.clear();
  map<string, int> mapper; // mapper.clear();

  // entering 7 name-score pairs below
  mapper["john"] = 78;   used_values.insert(78);  // john 78
  mapper["billy"] = 69;  used_values.insert(69);  // billy 69
  mapper["andy"] = 80;   used_values.insert(80);  // andy 80
  mapper["steven"] = 77; used_values.insert(77);  // steven 77
  mapper["felix"] = 82;  used_values.insert(82);  // felix 82
  mapper["grace"] = 75;  used_values.insert(75);  // grace 75
  mapper["martin"] = 81; used_values.insert(81);  // martin 81
```

Exemplo (Árvores de busca binárias balanceadas). continua...

```
// then the internal content of mapper MAY be something like this:
// re-read balanced BST concept if you do not understand this diagram
// the keys are names (string)!
//                          (grace,75)
//           (billy,69)                    (martin,81)
//     (andy,80)   (felix,82)    (john,78)  (steven,77)

// iterating through the content of mapper will give a sorted output
// based on keys (names)
for (map<string, int>::iterator it = mapper.begin(); it != mapper.end(); it++)
  printf("%s %d\n", ((string)it->first).c_str(), it->second);

// map can also be used like this
printf("stevens score is %d, graces score is %d\n",
  mapper["steven"], mapper["grace"]);
printf("==================\n");

// interesting usage of lower_bound and upper_bound
// display data between ["f".."m") ('felix' is included, martin' is excluded)
for (map<string, int>::iterator it = mapper.lower_bound("f"); it != mapper.
   upper_bound("m"); it++)
  printf("%s %d\n", ((string)it->first).c_str(), it->second);
```

Exemplo (Árvores de busca binárias balanceadas).

```
  // the internal content of used_values MAY be something like this
  // the keys are values (integers)!
  //                  (78)
  //          (75)              (81)
  //      (69)    (77)    (80)    (82)

  // O(log n) search, found
  printf("%d\n", *used_values.find(77));
  // returns [69, 75] (these two are before 77 in the inorder traversal of this
     BST)
  for (set<int>::iterator it = used_values.begin(); it != used_values.lower_bound
     (77); it++)
    printf("%d,", *it);
  printf("\n");
  // returns [77, 78, 80, 81, 82] (these five are equal or after 77 in the inorder
      traversal of this BST)
  for (set<int>::iterator it = used_values.lower_bound(77); it != used_values.end
     (); it++)
    printf("%d,", *it);
  printf("\n");
  // O(log n) search, not found
  if (used_values.find(79) == used_values.end())
    printf("79 not found\n");

  return 0;
}
```

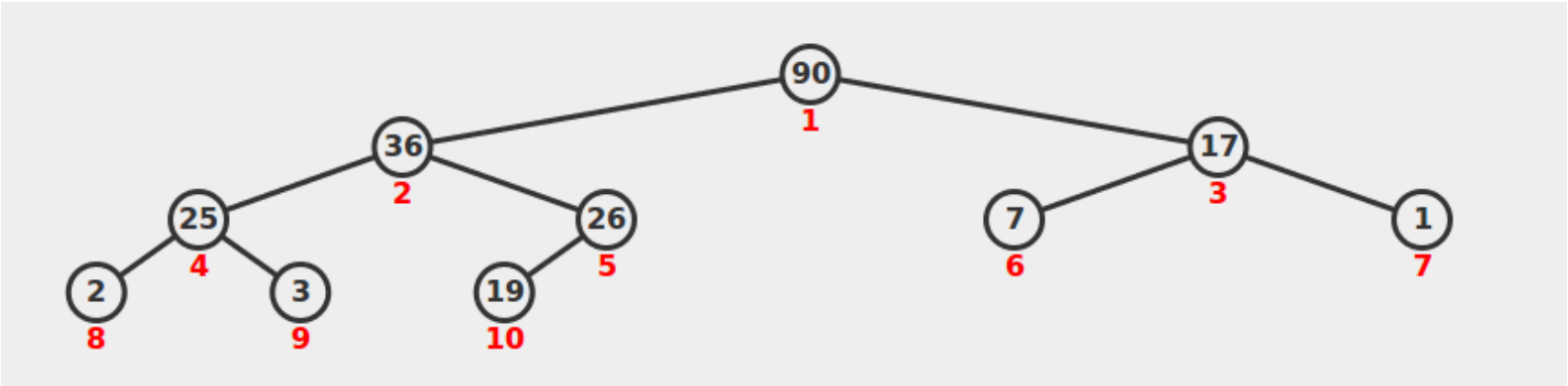
Exemplo (Árvores de busca binárias balanceadas).

Saída:

```
andy 80
billy 69
felix 82
grace 75
john 78
martin 81
steven 77
stevens score is 77, graces score is 75
==================
felix 82
grace 75
john 78
77
69,75,
77,78,80,81,82,
79 not found
```

Filas de prioridades ou heaps (C++ STL `priority_queue`, Java `PriorityQueue`)

- Um heap de máximo é uma árvore binária completa, tal que cada nó x possui a propriedade de heap, que consiste na restrição de que todos os filhos do nó x possuem valores menores do que x. Isso implica que a raiz será sempre o maior elemento do heap.
- Um heap pode ser representado por um vetor. Nesse caso, os elementos da árvore são visitados de cima para baixo e da esquerda para a direita para serem armazenados sequencialmente no vetor.



Filas de prioridades ou heaps (C++ STL `priority_queue`, Java `PriorityQueue`)

- Dado um índice i do vetor, é possível visitar o nó pai, filho esquerdo e filho direito do nó i a partir dos índices $\lfloor \frac{i}{2} \rfloor$, $2i$ e $2i + 1$, respectivamente. Por manipulação de bits, esses cálculos ficariam $i >> 1$, $i << 1$ e $(i << 1) + 1$.
- Um heap é uma estrutura de dados muito útil para representar fila de prioridades, onde um item de maior prioridade (maior elemento) pode ser removido e um novo elemento qualquer pode ser inserido em tempo $O(\log n)$.
- São utilizados em problemas importantes de grafos como árvore geradora mínima (Prim) e caminhos mínimos (Djikstra).
- Uma implementação de fila de prioridades pode ser encontrada na classe C++ STL `priority_queue`.

Exemplo (Filas de prioridades). continua...

```
#include <cstdio>
#include <iostream>
#include <string>
#include <queue>
using namespace std;

int main() {
  int money;
  char name[20];
  priority_queue< pair<int, string> > pq;              // introducing 'pair'
  pair<int, string> result;

  // suppose we enter these 7 money-name pairs below
  pq.push(make_pair(100, "john"));           // inserting a pair in O(log n)
  pq.push(make_pair(10, "billy"));
  pq.push(make_pair(20, "andy"));
  pq.push(make_pair(100, "steven"));
  pq.push(make_pair(70, "felix"));
  pq.push(make_pair(2000, "grace"));
  pq.push(make_pair(70, "martin"));
  // priority queue will arrange items in 'heap' based
  // on the first key in pair, which is money (integer), largest first
  // if first keys tie, use second key, which is name, largest first
```

Exemplo (Filas de prioridades)

```
  // the internal content of pq heap MAY be something like this:
  // re-read (max) heap concept if you do not understand this diagram
  // the primary keys are money (integer), secondary keys are names (string)!
  //                          (2000,grace)
  //            (100,steven)                  (70,martin)
  //     (100,john)   (10,billy)     (20,andy)  (70,felix)

  // let's print out the top 3 person with most money
  result = pq.top();                  // O(1) to access the top / max element
  pq.pop();           // O(log n) to delete the top and repair the structure
  printf("%s has %d $\n", ((string)result.second).c_str(), result.first);
  result = pq.top(); pq.pop();
  printf("%s has %d $\n", ((string)result.second).c_str(), result.first);
  result = pq.top(); pq.pop();
  printf("%s has %d $\n", ((string)result.second).c_str(), result.first);

  return 0;
}
```

Exemplo (Árvores binárias de busca balanceadas).

Saída:

```
grace has 2000 $
steven has 100 $
john has 100 $
```

Referências

1. S. Halim e F. Halim. Competitive Programming 2, Second Edition Lulu (www.lulu.com), 2011. (IMECC – 005.1 H139c)
2. S. S. Skiena, M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
3. T.H. Cormen, C.E. Leiserson, R.L.Rivest e C. Stein. Introduction to Algorithms. 2nd Edition, McGraw-Hill, 2001. (no. chamada IMECC – 005.133 Ar64j 3.ed.)
4. U. Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989. (no. chamada IMECC – 005.133 Ec53t 2.ed.)