‹Algoritmos de Ordenação› e ‹Bibliotecas›

Kaio Christaldo
Fabricio Matsunaga

‹Conteúdos›

- **Algoritmos de Ordenação**
  - Bubble Sort
  - Merge Sort
  - Quick Sort
  - Insertion Sort
  - Heap Sort

- **Bibliotecas Complementares**
  - **‹Algorithms›**
  - **‹Iterator›**
  - **‹Tuple›**

Apresentação Problema Motivador

beecrowd | 1088

Bolhas e Baldes

Por Cláudio L. Lucchesi 🇧🇷 Brasil

Timelimit: 3

Andrea, Carlos e Marcelo são muito amigos e passam todos os finais de semana à beira da piscina. Enquanto Andrea se bronzeia ao sol, os dois ficam jogando Bolhas. Andrea, uma cientista da computação muito esperta, já disse a eles que não entende por que passam tanto tempo jogando um jogo tão primário.

Usando o computador portátil dela, os dois geram um inteiro aleatório N e uma seqüência de inteiros, também aleatória, que é uma permutação de $1, 2, \ldots, N$.

O jogo então começa, cada jogador faz um movimento, e a jogada passa para o outro jogador. Marcelo é sempre o primeiro a começar a jogar. Um movimento de um jogador consiste na escolha de um par de elementos consecutivos da seqüência que estejam fora de ordem e em inverter a ordem dos dois elementos. Por exemplo, dada a seqüência 1, 5, 3, 4, 2, o jogador pode inverter as posições de 5 e 3 ou de 4 e 2, mas não pode inverter as posições de 3 e 4, nem de 5 e 2. Continuando com o exemplo, se o jogador decide inverter as posições de 5 e 3 então a nova seqüência será 1, 3, 5, 4, 2.

Mais cedo ou mais tarde, a seqüência ficará ordenada. Perde o jogador impossibilitado de fazer um movimento. Andrea, com algum desdém, sempre diz que seria mais simples jogar cara ou coroa, com o mesmo efeito. Sua missão, caso decida aceitá-la, é determinar quem ganha o jogo, dada a seqüência inicial.

Entrada

A entrada contém vários casos de teste. Os dados de cada caso de teste estão numa única linha, e são inteiros separados por um espaço em branco. Cada linha contém um inteiro N ($2 \leq N \leq 10^5$), seguido da seqüência inicial $P = (X_1, X_2, \ldots, X_N)$ de N inteiros distintos dois a dois, onde $1 \leq X_i \leq$ N `para` $1 \leq i \leq N$.

O final da entrada é indicado por uma linha que contém apenas o número zero.

Saída

Para cada caso de teste da entrada seu programa deve imprimir uma única linha, com o nome do vencedor, igual a `Carlos` ou `Marcelo`., sem espaços em branco.

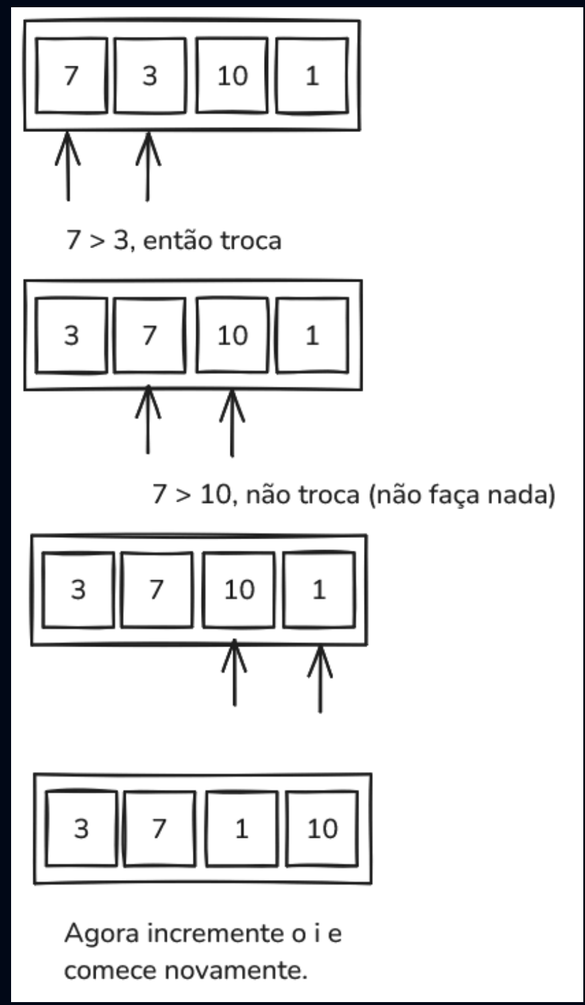
[1088 - Bolhas e Baldes]

‹Bubble Sort›

‹Bubble Sort›

- **Definição:** Algoritmo de ordenação muito simples e intuitivo.
  - Percorrer o vetor do início ao fim, sem interrupção, trocando de posição dois elementos consecutivos sempre que estes se apresentem fora de ordem.
  - Complexidade de tempo: **O(n²)**

```
para i = 1, ... n faça
    para j = 1, ... n − 1 faça
        se L[j] . chave > L[j + 1] . chave então
            trocar(L[j], L[j + 1])
```

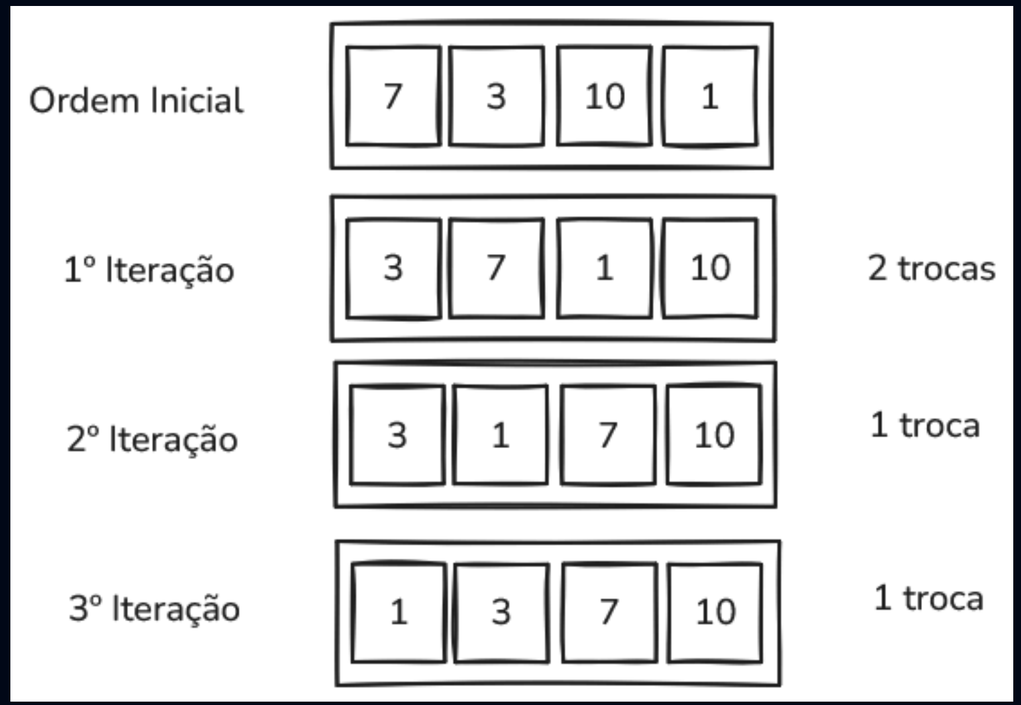


‹Bubble Sort›

- Algoritmo de ordenação **melhorado**
  - Armazena ultima posição de troca
  - Complexidade de tempo: **O(n²)**
  - Complexidade de tempo: **Θ(n)**

```
mudou := V; n' := n; guarda := n
enquanto mudou faça
        j := 1; mudou := F
        enquanto j < n' faça
                se L[ j ] . chave > L[ j + 1 ] . chave então
                        trocar(L[ j ], L[ j + 1 ])
                        mudou := V
                        guarda := j
                        j := j + 1
        n' := guarda
```

Exemplo:

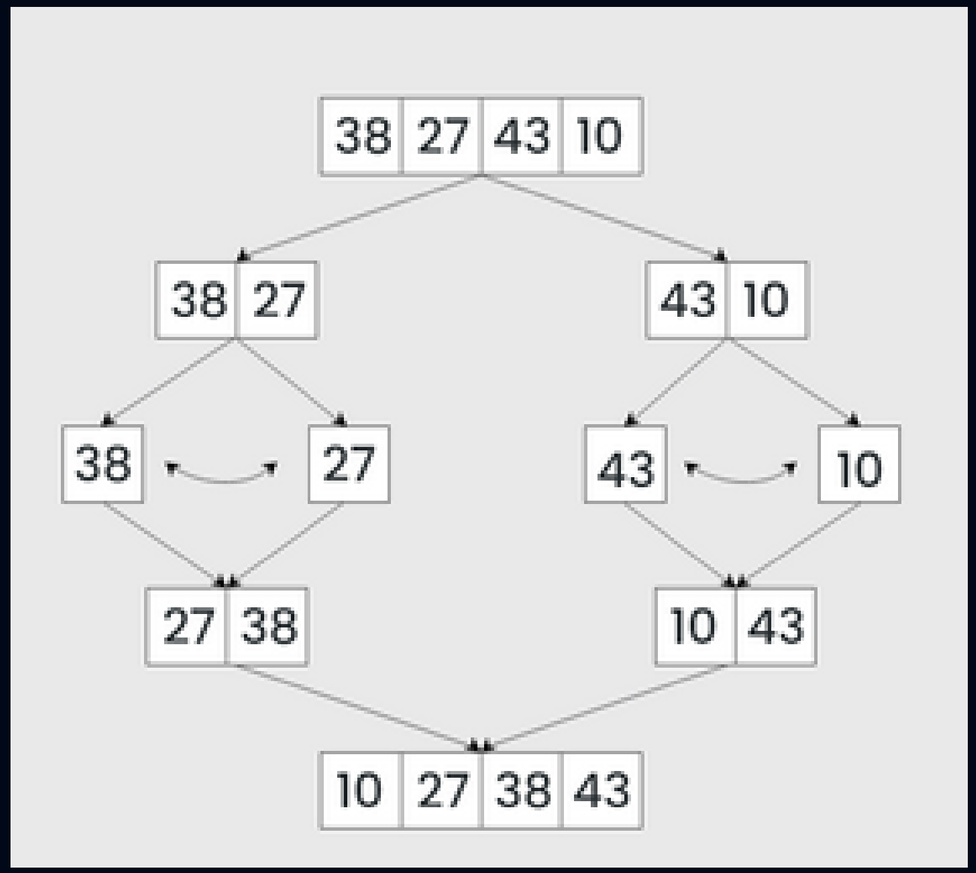


‹Merge Sort›

‹Merge Sort›

- **Definição:** Algoritmo de ordenação por intercalação.
  - O método de ordenação por intercalação consiste em dividir a lista original em duas metades e ordená-las; o resultado são duas listas ordenadas que podem ser intercaladas. Para ordenar cada uma das metades o processo considerado é o mesmo, sendo o problema dividido em problemas menores, que são sucessivamente solucionados.
  - Recursividade
  - Complexidade de tempo: **O(n log n)**

‹Merge Sort›




```
procedimento mergesort (esq, dir);
    se esq < dir então
        centro := ⌊(esq + dir)/2⌋;
        mergesort(esq, centro);
        mergesort(centro + 1, dir);
        intercalar(esq, centro + 1, dir);
```

```
procedimento intercalar(L, Tmp, ini1, ini2, fim2);
    fim1 := ini2 – 1; nro := 0
    ind := ini1
        enquanto (ini1 ≤ fim1) e (ini2 ≤ fim2)faça
            se L[ini1] . chave se L[ini1] L[ini2] . chave então
                Tmp[ind] := L[ini1]
                ini1 := ini1 + 1
            senão Tmp[ind] := L[ini2]
                ini2 := ini2 + 1
            ind := ind + 1; nro := nro + 1
        enquanto {ini1 ≤ fim1} faça
            Tmp[ind] := L[ini1]
            ini1 := ini1 + 1; ind := ind + 1; nro := nro + 1
        enquanto {ini2 ≤ fim2} faça
            Tmp[ind] := L[ini2]
            ini2 := ini2 + 1; ind := ind + 1; nro := nro + 1
        para i := 1, ..., nro faça
            L[i + ini 1 – 1] := Tmp[i + ini1 – 1]
```

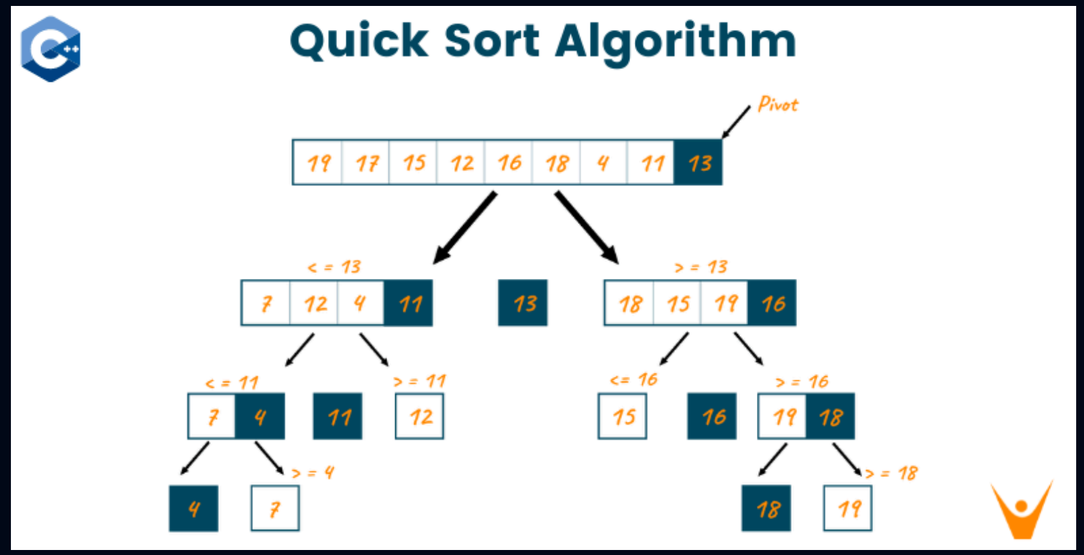
‹Quick Sort›

‹Quick Sort›

- **Definição:** (ordenação rápida), um dos mais eficientes dentre os conhecidos. Dada uma tabela L com n elementos, o procedimento recursivo para ordenar L consiste nos **seguintes passos**:
  - se n = 0 ou n = 1 então a tabela está ordenada;
  - escolha qualquer elemento x em L – este elemento é chamado pivô;
  - separe L – {x} em dois conjuntos de elementos disjuntos:
    - S1 = {w ∈ L – {x}|w < x};
    - S2 = {w ∈ L – {x}|w ≥ x};
  - o procedimento de ordenação é chamado recursivamente para S1 e S2;
  - L recebe a concatenação de S1, seguido de x, seguido de S2.
- **In-Place** - não faz muitas copias.
- Complexidade de tempo:
  - **mediana de três: $O(n^2) \rightarrow O(n \log n)$**
- Geralmente mais rápido na prática que Merge Sort, com boa escolha de pivô

‹Quick Sort›

- **Ideia do Algoritmo:**




```
procedimento quicksort (ini, fim)
    se fim – ini < 2 então
        se fim – ini = 1 então
            se L[ini] . chave > L[ fim] . chave então
                trocar(L[ini], L[ fim]);
    senão PIVO(ini, fim, mediana)
        trocar((L[mediana], L[ fim])
        i := ini; j := fim – 1
        key := L[ fim] . chave
        enquanto j ≥ i faça
            enquanto L[i] . chave < key faça
                i := i + 1
            enquanto L[ j ] . chave > key faça
                j := j – 1
            se j ≥ i então
                trocar((L[i], L[ j ])
                i := i + 1; j := j – 1
        trocar((L[i], L[ fim])
        quicksort(ini, i – 1)
        quicksort(i + 1, fim);
```

Implementações de ordenação (prontas)

- **sort() -** Introsort (mistura de **quicksort**, **heapsort** e **insertion sort**) - **O(n log n)**
- **stable_sort() -** mergesort com buffer auxiliar - **O(n log n)**
- **partial_sort() -** ordena apenas os **k** menores elementos em ordem, o resto não é garantido. - **O(n log k)**
- **make_heap() + sort_heap() -** **O(n)** e **O(n log n)**

Resolução do Problema Motivador

1088 - Bolhas e Baldes

Dicas:

- Implementar o MergeSort
- Usar variavel para contar numero de iterações

A resolução estará disponível no Drive. Tente resolver por conta própria e, se precisar, compare com a solução! 😊

Apresentação Problema Motivador

beecrowd | 1244

Ordenação por Tamanho

Por TopCoder* EUA

Timelimit: 1

Crie um programa para ordenar um conjunto de strings pelo seu tamanho. Seu programa deve receber um conjunto de strings e retornar este mesmo conjunto ordenado pelo tamanho das palavras, se o tamanho das strings for igual, deve-se manter a ordem original do conjunto.

Entrada

A primeira linha da entrada possui um único inteiro N, que indica o número de casos de teste. Cada caso de teste poderá conter de 1 a 50 strings inclusive, e cada uma das strings poderá conter entre 1 e 50 caracteres inclusive. Os caracteres poderão ser espaços, letras, ou números.

Saída

A saída deve conter o conjunto de strings da entrada ordenado pelo tamanho das strings. Um espaço em branco deve ser impresso entre duas palavras.

Exemplo de Entrada	Exemplo de Saída
4 Top Coder comp Wedn at midnight one three five I love Cpp sj a sa df r e w f d s a v c x z sd fd	midnight Coder comp Wedn Top at three five one love Cpp I sj sa df sd fd a r e w f d s a v c x z

1244 - Ordenação por Tamanho

‹Ordenação por Inserção›

‹Ordenação por Inserção› - Insertion Sort

Imagine uma tabela já ordenada até o i-ésimo elemento. A ordenação da tabela pode ser estendida até o (i + 1)-ésimo elemento por meio de comparações sucessivas deste com os elementos anteriores, isto é, com o i-ésimo elemento, com o (i – 1)-ésimo elemento etc., procurando sua posição correta na parte da tabela que já está ordenada.

```
para i = 2, ... n faça
        prov := L[i]; valor := L[i] . chave
        j := i – 1
        enquanto j≥1 e valor < L[ j ] . chave faça
                L[ j + 1] := L[ j ]
                j := j – 1
        L[ j + 1] := prov
```

‹Ordenação por Inserção› – Insertion Sort

Iteração	Tabela	Trocas
tabela inicial	40 37 95 42 23 51 27	
após $i = 2$	37 40 95 42 23 51 27	1
após $i = 3$	37 40 95 42 23 51 27	0
após $i = 4$	37 40 42 95 23 51 27	1
após $i = 5$	23 37 40 42 95 51 27	4
após $i = 6$	23 37 40 42 51 95 27	1
após $i = 7$	23 27 37 40 42 51 95	5

FIGURA 7.2 Um exemplo de ordenação por inserção.

‹Ordenação por Inserção› - Insertion Sort

A análise da complexidade do algoritmo é bastante simples e utiliza o conceito de inversões. Dados dois elementos da tabela $L[i]$ e $L[j]$, sendo $i < j$, uma inversão ocorre quando $L[i]$. *chave* > $L[j]$. *chave*. Como pode ser visto no exemplo apresentado na Figura 7.2, o número de trocas realizadas pelo algoritmo é exatamente o número de inversões da tabela. O percurso em cada iteração termina exatamente quando a inversão não ocorre. Então, o algoritmo tem complexidade de melhor caso $O(n)$, que ocorre quando o número de inversões é zero. No pior caso, quando a tabela está em ordem inversa, são executadas $n - 1$ iterações (para $i = 2, \ldots, n$) e, em cada uma delas, removidas $i - 1$ inversões. Logo,

$$\sum_{i=1}^{n-1} Inv(i) = 1 + 2 + \ldots + (n-1) = O(n^2).$$

‹Heapsort›

‹Heapsort› - Ordenação em Heap

Seja L a tabela a ser disposta em ordem não decrescente segundo as suas chaves, sendo L uma lista de prioridades. A prioridade atribuída a cada nó é considerada como sendo igual ao valor de sua chave de ordenação.

Sabe-se que o primeiro elemento da lista é o de maior prioridade. Esse elemento é obrigatoriamente o último na lista ordenada. Pode-se então realizar a troca do primeiro com o último elemento e diminuir a tabela. A tabela L, diminuída de um elemento, pode não ser uma lista de prioridades (o seu primeiro elemento foi modificado) e deve ser reorganizada. Repetindo-se este processo para todos os elementos de L, o resultado final é a tabela ordenada.

‹Heapsort› – Ordenação em Heap

Algoritmo 7.6 Ordenação em heap

```
arranjar(n)
m := n
enquanto m > 1 faça
        trocar(L[1], L[m])
```



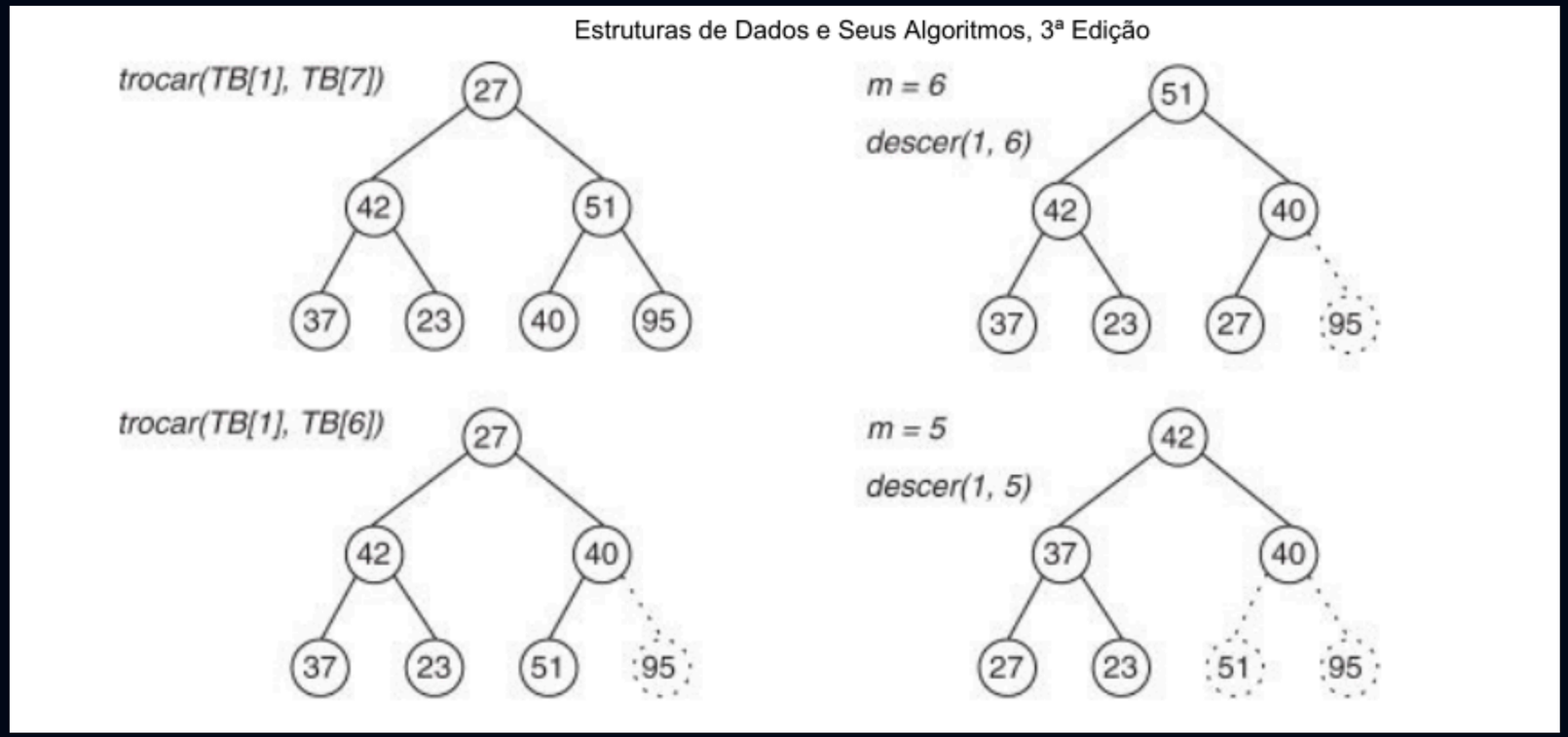
```
        m := m – 1
        descer(1, m)
```

A complexidade do Algoritmo 7.6 é de fácil avaliação. O procedimento arranjar tem complexidade $O(n)$; o procedimento descer, $O(\log n)$. Como o procedimento descer é chamado n vezes, para cada elemento colocado em sua posição definitiva na tabela, a complexidade total do algoritmo é $O(n \log n)$.

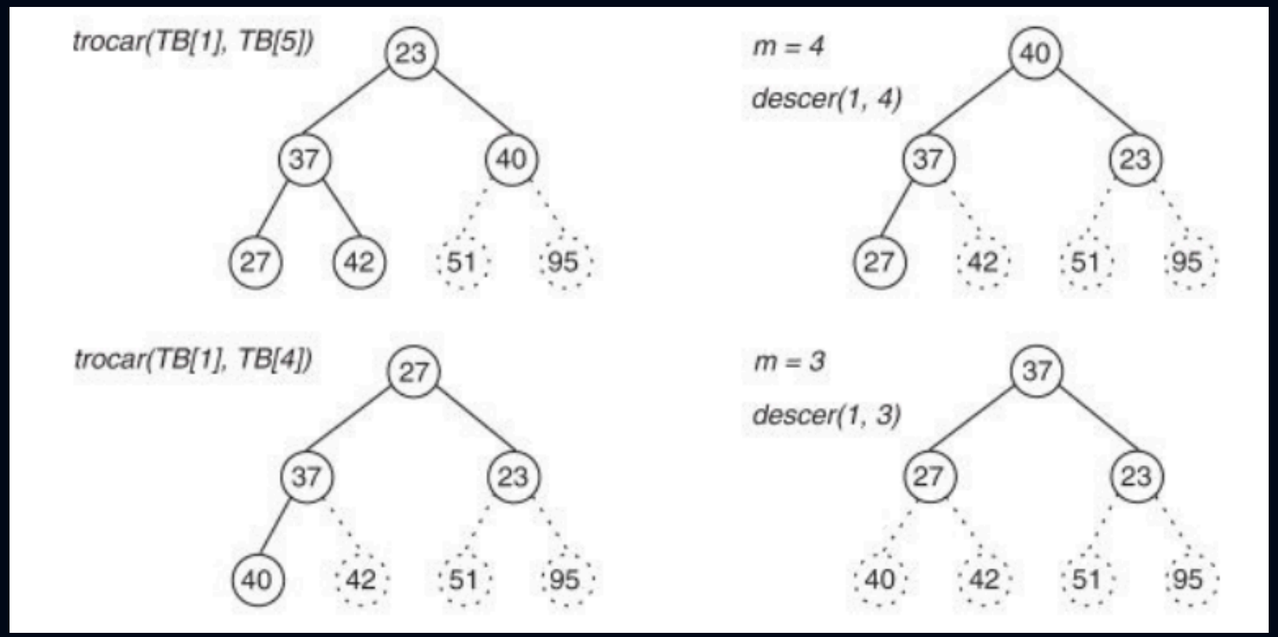
‹Heapsort› – Ordenação em Heap

95 60 78 39 28 66 70 33

FIGURA 7.5 Lista de prioridades.



‹Heapsort› – Ordenação em Heap



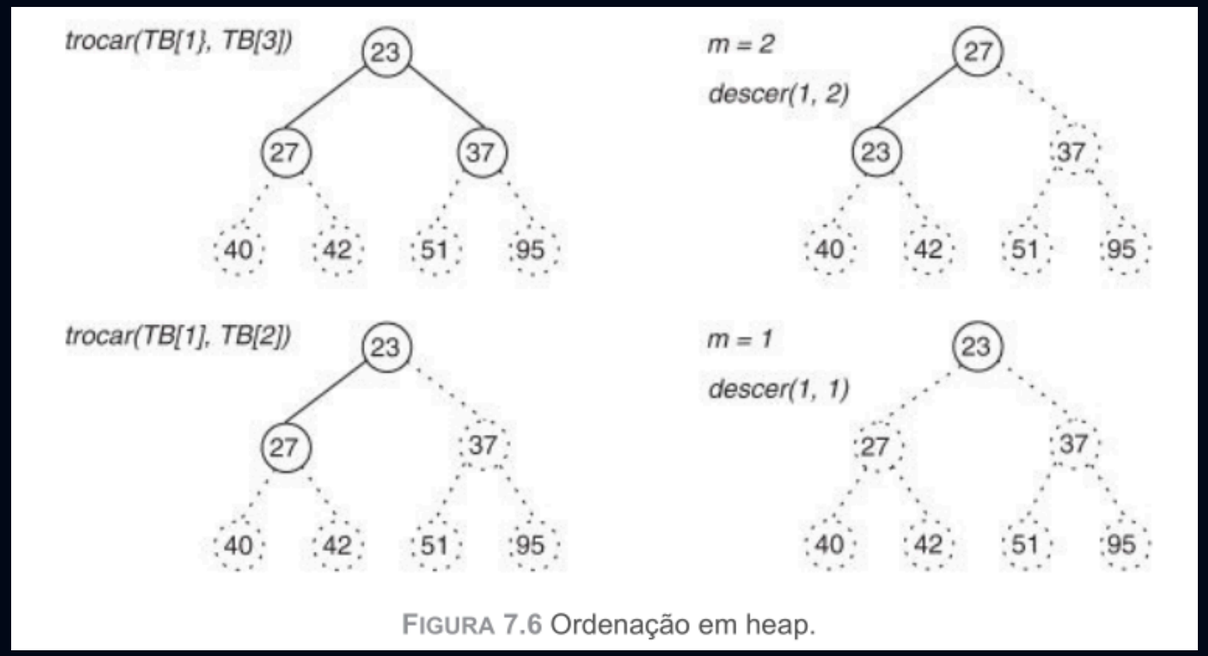


FIGURA 7.6 Ordenação em heap.

‹Bibliotecas›

‹Bibliotecas› - ‹algorithm›

Em aulas passadas aprendemos como estruturas de dados armazenam e organizam os dados.

Usamos algoritmos para resolver problemas atravez de ordenação, busca e manipulação das estruturas de dados.

A biblioteca ‹algorithm› prove diversar funções uteis para realizar essas tarefas usando iterators.

<algorithm> - Ordenação

```cpp
// Create a vector called cars that will store strings
vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

// Sort cars alphabetically
sort(cars.begin(), cars.end());
```

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Sort numbers numerically
sort(numbers.begin(), numbers.end());
```

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Sort numbers numerically in reverse order
sort(numbers.rbegin(), numbers.rend());
```

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Sort numbers numerically, starting from the fourth element (only sort 5, 9, and 2)
sort(numbers.begin() + 3, numbers.end());

```

<algorithm> - Buscas

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Search for the number 3
auto it = find(numbers.begin(), numbers.end(), 3);

```

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Sort the vector in ascending order
sort(numbers.begin(), numbers.end());

// Find the first value that is greater than 5 in the sorted vector
auto it = upper_bound(numbers.begin(), numbers.end(), 5);
```

<algorithm> - Buscas

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Find the smallest number
auto it = min_element(numbers.begin(), numbers.end());
```

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Find the largest number
auto it = max_element(numbers.begin(), numbers.end());
```

<algorithm> - Modificação

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Create a vector called copiedNumbers that should store 6 integers
vector<int> copiedNumbers(6);

// Copy elements from numbers to copiedNumbers
copy(numbers.begin(), numbers.end(), copiedNumbers.begin());
```

```cpp
// Create a vector called numbers that will store 6 integers
vector<int> numbers(6);

// Fill all elements in the numbers vector with the value 35
fill(numbers.begin(), numbers.end(), 35);

```

<algorithm> - Modificação

```cpp
// Create a vector called numbers that will store integers
vector<int> numbers = {1, 7, 3, 5, 9, 2};

// Create a vector called copiedNumbers that should store 6 integers
vector<int> copiedNumbers(6);

// Copy elements from numbers to copiedNumbers
copy(numbers.begin(), numbers.end(), copiedNumbers.begin());
```

```cpp
// Create a vector called numbers that will store 6 integers
vector<int> numbers(6);

// Fill all elements in the numbers vector with the value 35
fill(numbers.begin(), numbers.end(), 35);

```

Resolução do Problema Motivador

A resolução estará disponível no Drive. Tente resolver por conta própria e, se precisar, compare com a solução! 😊


‹Iterator›

Biblioteca <Iterator>

- **Definição:** A forma de iterador mais óbvia é um ponteiro: um ponteiro pode apontar para elementos em um array e pode percorrê-los usando o operador de incremento (++). Mas outros tipos de iteradores são possíveis. Por exemplo, cada tipo de container (como uma lista) possui um tipo específico de iterador projetado para percorrer os seus elementos.

- Iteradores são classificados em cinco categorias, dependendo da funcionalidade que implementam:
  - Input (entrada)
  - Output (saída)
  - Forward (avançado)
  - Bidirectional (bidirecional)
  - Random Access (acesso aleatório)

Template ‹Iterator›

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

Template <iterator>

Operações

Principais Operações Usadas:

- **advance(it, n)** - Move o iterador **it** adiante (ou para trás se n for negativo).
  - **O(n)** para input/bidirectional, **O(1)** para random access
- **distance(first, last)** - Retorna o número de passos entre first e last.
  - **O(n)** para input/bidirectional, **O(1)** para random access
- **begin()**- Retorna um iterador para o início do container - **O(1)**
- **end()** - Retorna um iterador para o final do container - **O(1)**

Template <iterator>

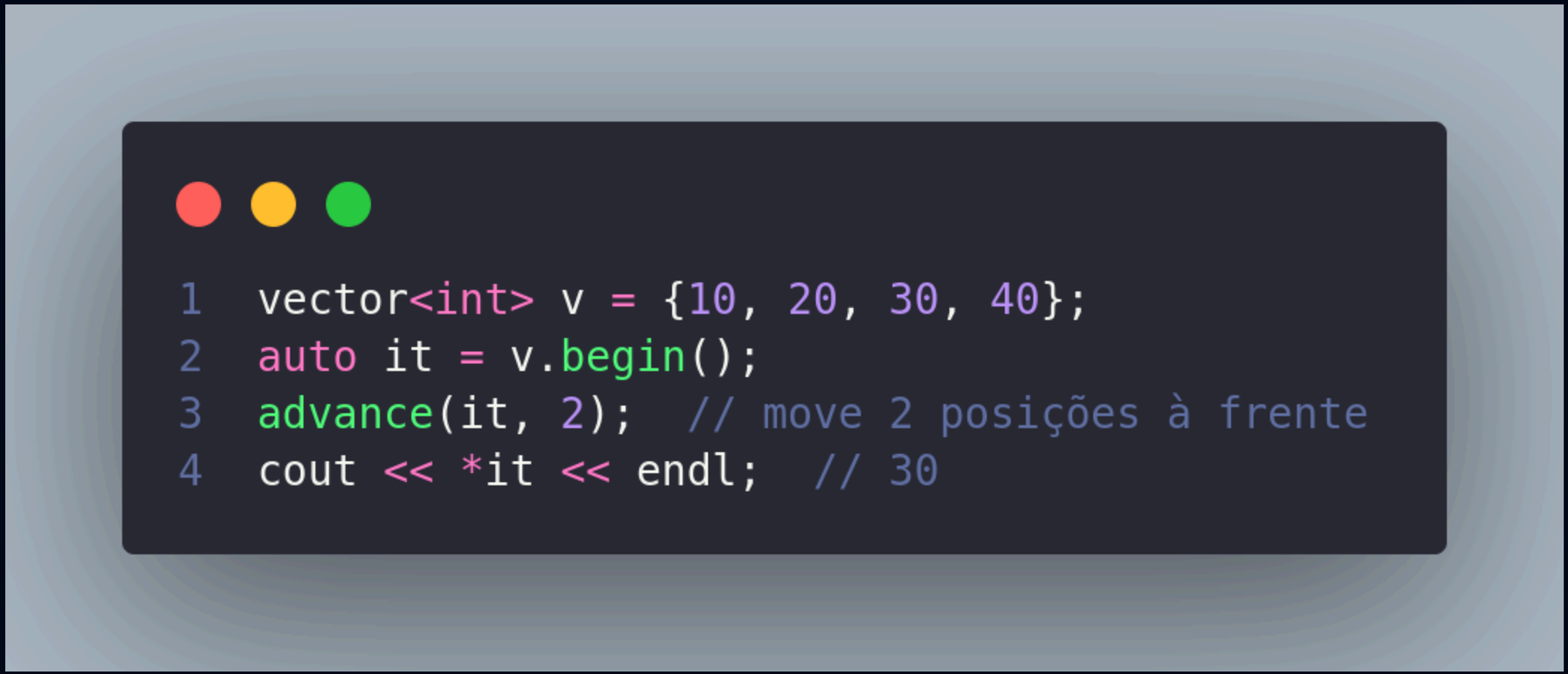
Operações

Principais Operações Usadas:

- **prev(it)** ou **prev(it, n)** - Retorna um iterador **n** posições antes de **it**.
  - **O(n)** para bidirectional, **O(1)** para random access
- **next(it)** ou **next(it, n)** - Retorna um iterador **n** posições depois de **it**.
  - **O(n)** para input/bidirectional, **O(1)** para random access
- **back_inserter(container)** -Cria um iterador que insere elementos no fim do container - **O(1)**
- **front_inserter(container)** - Cria um iterador que insere elementos no início do container - **O(1)**
- **inserter(container, it)** - Cria um iterador que insere elementos na posição it do container - **O(1)**
- **make_move_iterator(it)** - Cria um iterador que move em vez de copiar elementos - **O(1)**

Template <iterator>

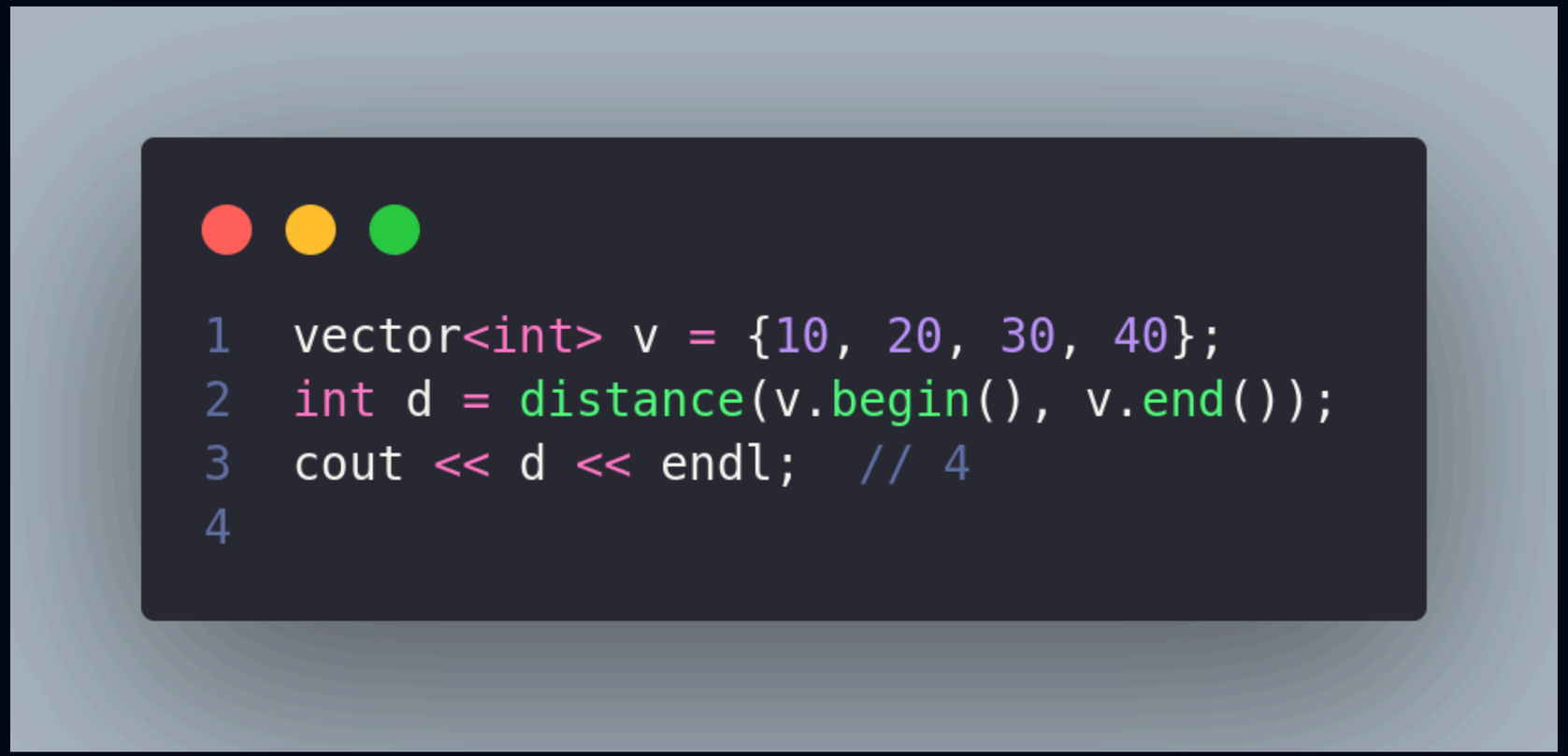
- advance(it, n)



```
1  vector<int> v = {10, 20, 30, 40};
2  auto it = v.begin();
3  advance(it, 2);  // move 2 posições à frente
4  cout << *it << endl;  // 30
```

Template <iterator>

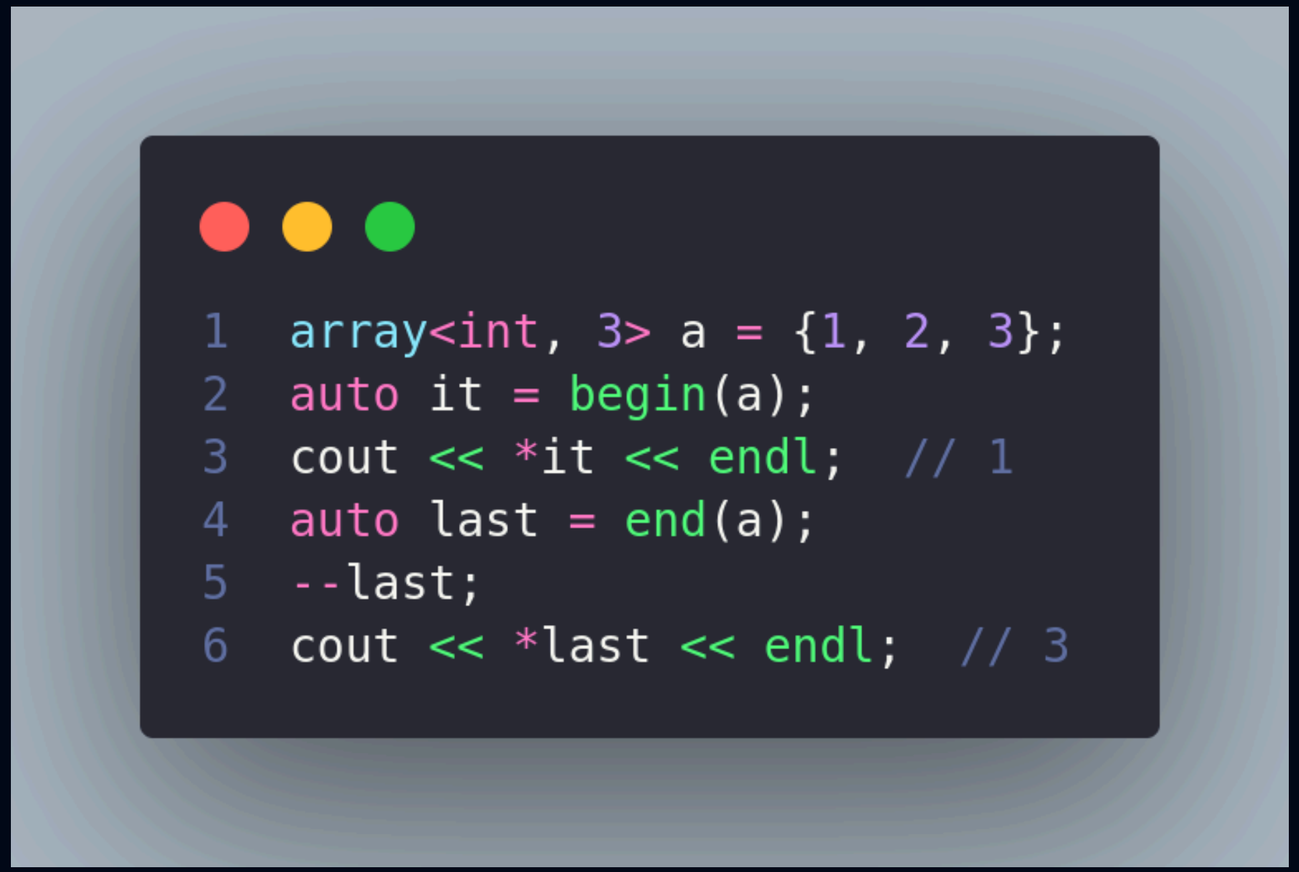
- distance(first, last)



```
1  vector<int> v = {10, 20, 30, 40};
2  int d = distance(v.begin(), v.end());
3  cout << d << endl;  // 4
4
```

Template <iterator>

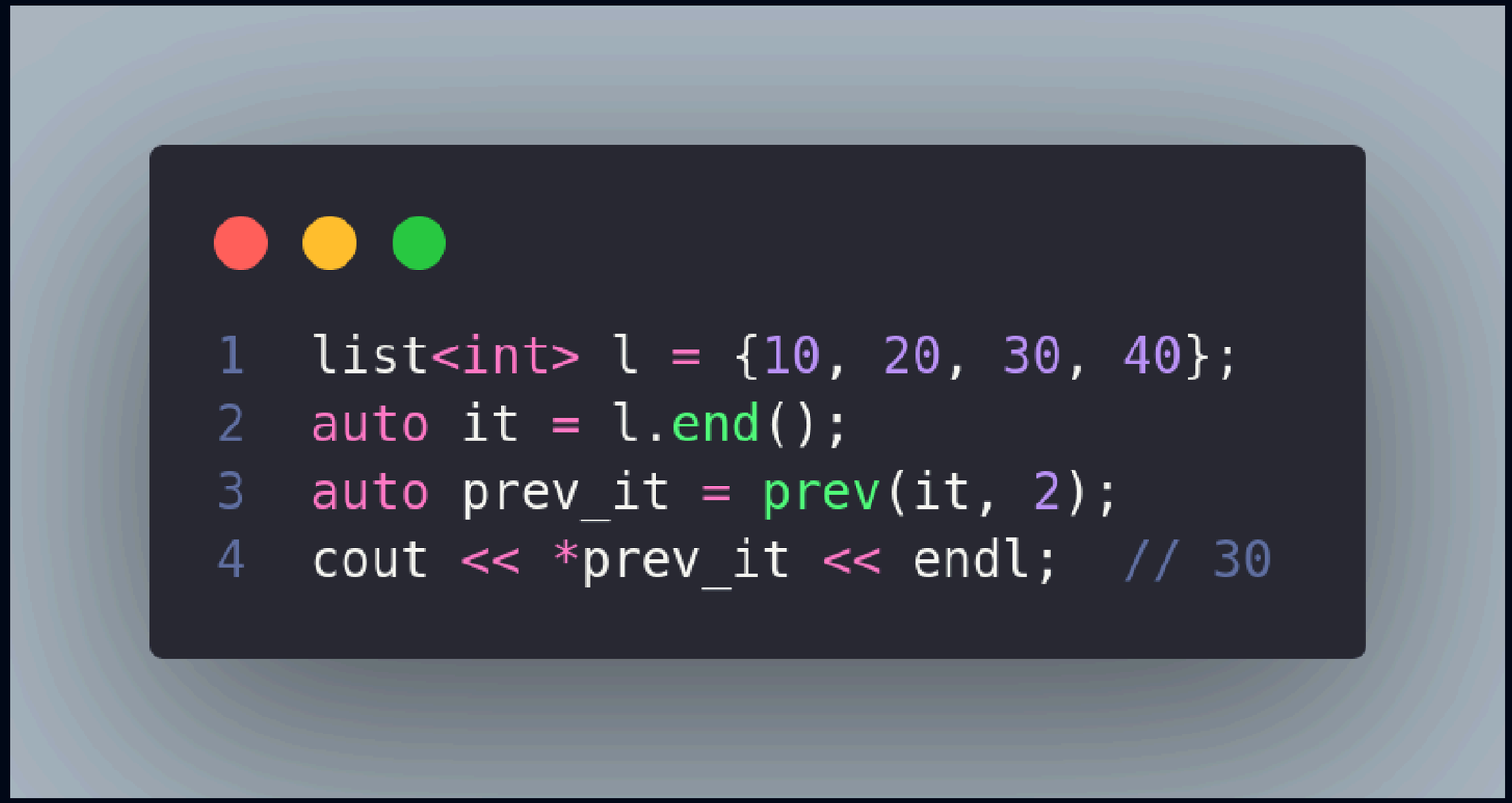
- **begin() e end()**



```cpp
array<int, 3> a = {1, 2, 3};
auto it = begin(a);
cout << *it << endl;  // 1
auto last = end(a);
--last;
cout << *last << endl;  // 3
```

Template <iterator>

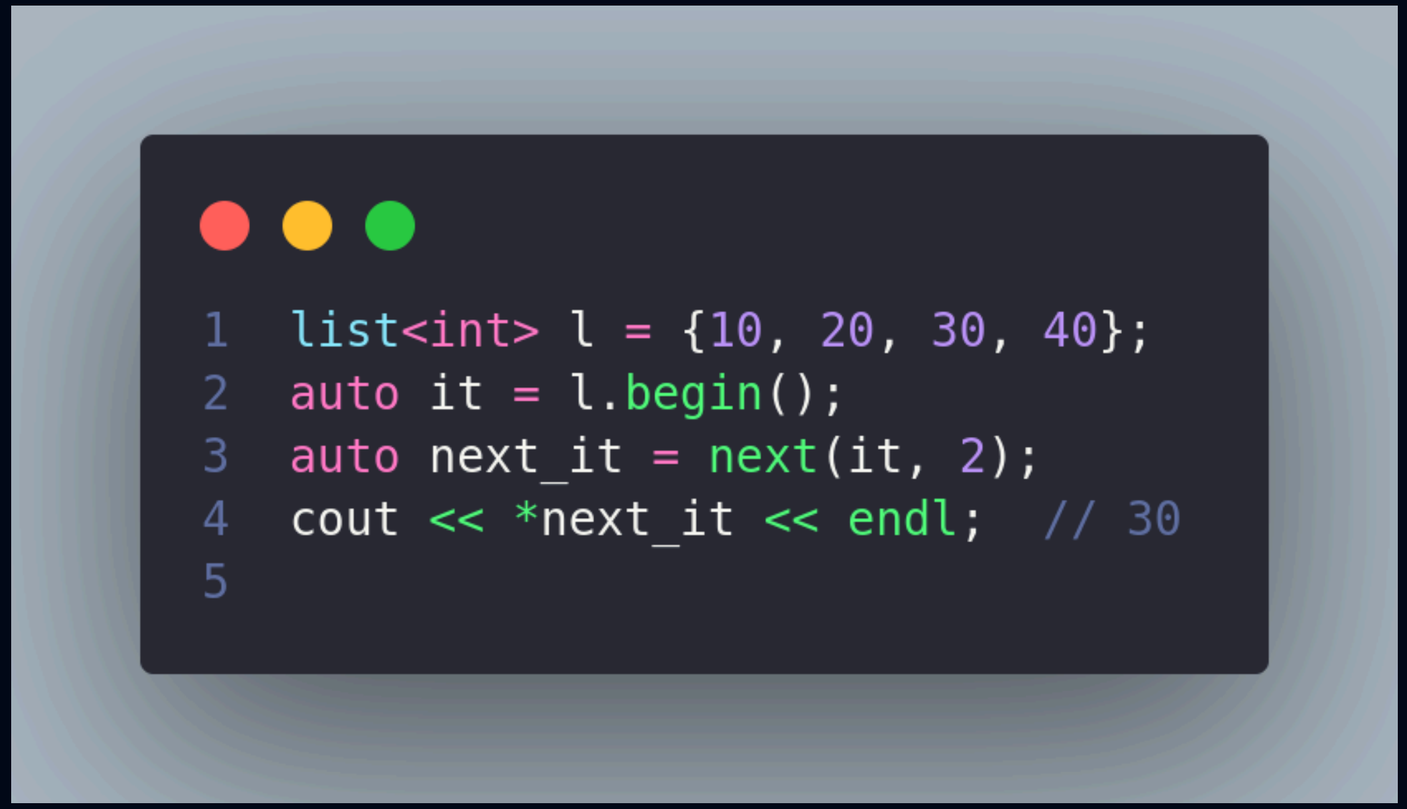
- prev(it) ou prev(it, n)



```
1  list<int> l = {10, 20, 30, 40};
2  auto it = l.end();
3  auto prev_it = prev(it, 2);
4  cout << *prev_it << endl;  // 30
```

Template <iterator>

- next(it) ou next(it, n)



```cpp
list<int> l = {10, 20, 30, 40};
auto it = l.begin();
auto next_it = next(it, 2);
cout << *next_it << endl;  // 30

```

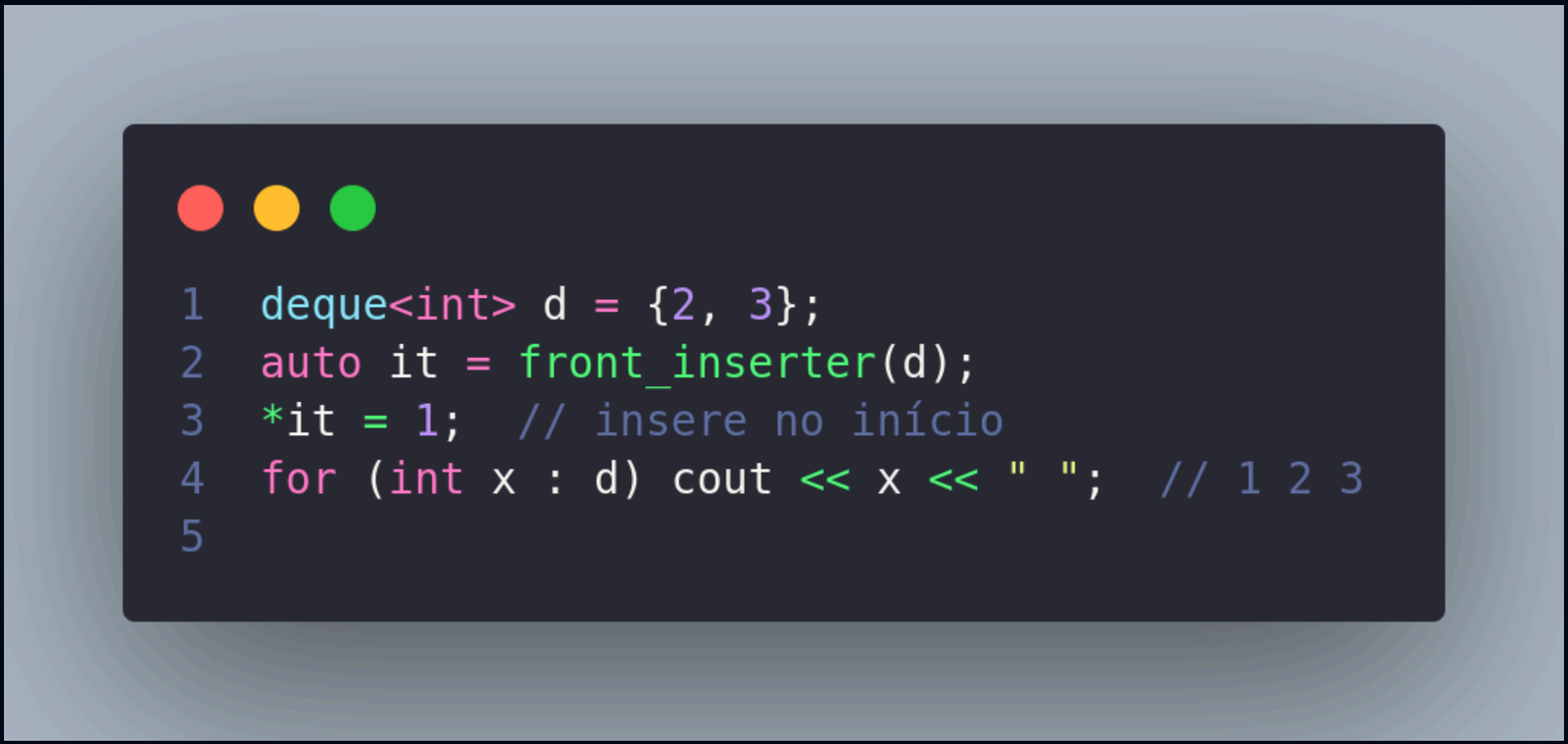
Template <iterator>

- back_inserter(container)

```cpp
vector<int> v = {1, 2};
auto it = back_inserter(v);
*it = 3;  // equivale a v.push_back(3);
*it = 4;
for (int x : v) cout << x << " ";  // 1 2 3 4
```

Template <iterator>

- front_inserter(container)

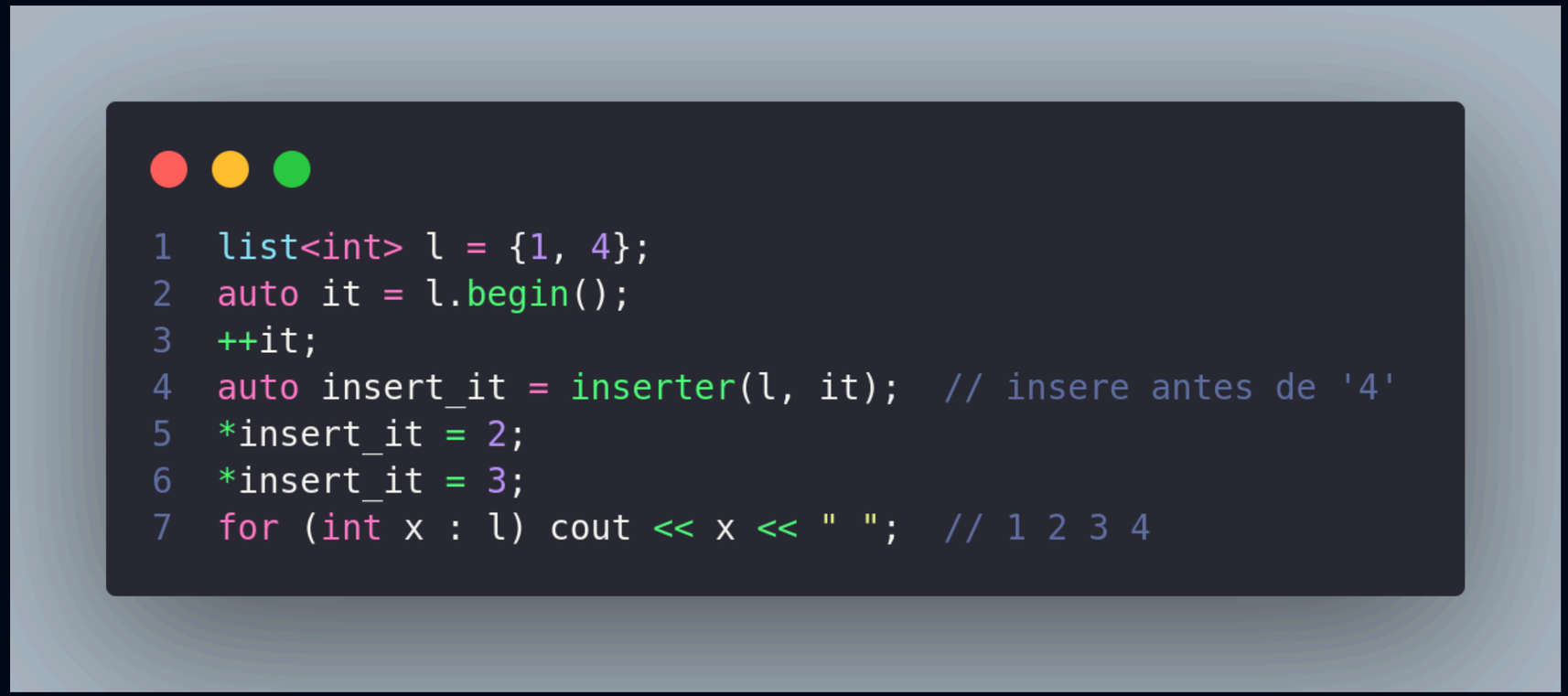


```cpp
deque<int> d = {2, 3};
auto it = front_inserter(d);
*it = 1;  // insere no início
for (int x : d) cout << x << " ";  // 1 2 3

```

Template <iterator>

- inserter(container, it)



```cpp
list<int> l = {1, 4};
auto it = l.begin();
++it;
auto insert_it = inserter(l, it);  // insere antes de '4'
*insert_it = 2;
*insert_it = 3;
for (int x : l) cout << x << " ";  // 1 2 3 4
```

Template <iterator>

- make_move_iterator(it)

```cpp
vector<string> from = {"A", "B", "C"};
vector<string> to;

copy(make_move_iterator(from.begin()), make_move_iterator(from.end()), back_inserter(to));

cout << "to: ";
for (auto& s : to) cout << s << " ";  // A B C
cout << "\nfrom: ";
for (auto& s : from) cout << s << " ";  // vazio: moved-from
```

Template <iterator>

Vantagens

- Abstração do acesso: Iterators permitem percorrer estruturas de dados (como vector, list, set) sem se preocupar com a implementação interna.
- Generalização: Muitos algoritmos da STL (como std::sort, std::find, std::copy) funcionam com qualquer tipo de container que exponha iteradores.

Desvantagens

- Maior risco de erros em iteradores inválidos
- Nem todos iteradores têm as mesmas operações
- Menor desempenho que ponteiros simples em arrays


Tuple

Template <tuple>

Operações

Principais Operações Usadas:

- **make_tuple(1, 'a', 3.14)** - Cria uma tupla com os valores passados, inferindo os tipos. - **O(1)**
- **get<1>(t)** - Acessa o elemento de índice 1 da tupla t. - **O(1)**
- **tie(a, b, c) = t** - Desempacota a tupla t em variáveis individuais a, b, e c. - **O(1)**
- **tuple_size<decltype(t)>::value** - Retorna a quantidade de elementos da tupla t (em tempo de compilação). - **O(1)**

Template <tuple>

Operações

Principais Operações Usadas:

- **tuple_element<0, decltype(t)>::type** - Obtém o tipo do elemento na posição 0 da tupla t (em tempo de compilação). - **O(1)**
- **forward_as_tuple(a, b)** - Cria uma tupla de referências com a e b, útil para passagem de argumentos. - **O(1)**
- **apply(f, t)** - Aplica a função f aos elementos da tupla t como argumentos. - **O(1) + custo da chamada de f**
- **tuple_cat(t1, t2)** - Concatena as tuplas t1 e t2 em uma única nova tupla. - **O(n)**

Lista de Exercícios

1088 - Bolhas e Baldes
1244 - Ordenação por Tamanho

Se tiver alguma dúvida ou dificuldade na resolução de algum exercício, sinta-se à vontade para perguntar! 😊

Referências

[1] CPLUSPLUS.COM. C++ list. Disponível em: https://cplusplus.com/reference/list/list/. Acesso em: 08 mai. 2025.

[1] CPLUSPLUS.COM. C++ forward_list. Disponível em: https://cplusplus.com/reference/forward_list/forward_list/. Acesso em: 08 mai. 2025.

GEEKSFORGEEKS. Heap em C++ STL. GeeksforGeeks, [S. l.], [s. d.]. Disponível em: https://www-geeksforgeeks-org.translate.goog/cpp-stl-heap/?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em: 8 maio 2025.