Programação Competitiva

Base para código:

```
#include <bits/stdc++.h>

using namespace std;

#define _ ios_base::sync_with_stdio(0);cin.tie(0);

typedef long long ll;

const int INF = 0x3f3f3f3f;
cosnt ll LINF = 0x3f3f3f3f3f3f3f3fll;

int main(){ _

    return 0;
}
```

Busca Binária

binary_search: Verifica se um elemento está presente em um intervalo ordenado.

lower_bound: Retorna um iterador para o primeiro elemento não menor que um valor especificado.

upper_bound: Retorna um iterador para o primeiro elemento maior que um valor especificado.

equal_range: Retorna um par de iteradores que delimitam o intervalo de elementos iguais a um valor especificado.

```
int main() {
    vector<int> v = {1, 3, 4, 4, 5, 7, 9};

    // Verificar se o elemento 4 está presente usando
binary_search
    if (binary_search(v.begin(), v.end(), 4)) {
        cout << "O elemento 4 está presente no vetor.\n";
    } else {
        cout << "O elemento 4 não está presente no vetor.\n";
```

```
    }

    // Encontrar o primeiro elemento >= 4 usando lower_bound
    auto lb = lower_bound(v.begin(), v.end(), 4);
    cout << "O primeiro elemento >= 4 está na posição: " << (lb -
v.begin()) << "\n";

    // Encontrar o primeiro elemento > 4 usando upper_bound
    auto ub = upper_bound(v.begin(), v.end(), 4);
    cout << "O primeiro elemento > 4 está na posição: " << (ub -
v.begin()) << "\n";

    // Encontrar o intervalo de elementos iguais a 4 usando
equal_range
    auto range = equal_range(v.begin(), v.end(), 4);
    cout << "Intervalo de elementos iguais a 4: ["
              << (range.first - v.begin()) << ", "
              << (range.second - v.begin()) << ")\n";

    return 0;
}
```

Ordenação

1- Quick Sort

Desempenho Médio: QuickSort é um dos algoritmos de ordenação mais rápidos para a maioria dos casos práticos, com uma complexidade média de O(nlogn).

In-place: O QuickSort não requer espaço adicional significativo (apenas a pilha de chamadas da recursão), o que o torna eficiente em termos de uso de memória.

Bom para Arrays Grandes e Aleatórios: Funciona bem com grandes conjuntos de dados e é eficiente para entradas aleatórias.

```
int particionar(vector<int>& vetor, int baixo, int alto) {
    int pivo = vetor[alto];
    int i = baixo - 1;

    for (int j = baixo; j < alto; ++j) {
        if (vetor[j] < pivo) {
            ++i;
            swap(vetor[i], vetor[j]);
        }
    }
    swap(vetor[i + 1], vetor[alto]);
```

```
    return i + 1;
}

void quickSort(vector<int>& vetor, int baixo, int alto) {
    if (baixo < alto) {
        int pi = particionar(vetor, baixo, alto);
        quickSort(vetor, baixo, pi - 1);
        quickSort(vetor, pi + 1, alto);
    }
}

int main() {
    vector<int> vetor = {10, 7, 8, 9, 1, 5};
    quickSort(vetor, 0, vetor.size() - 1);
    for (int numero : vetor) {
        cout << numero << " ";
    }
    return 0;
}
```

2- MergeSort

Garantia de Tempo de Execução: Tem uma complexidade garantida de O(nlogn) para todos os casos.

Estabilidade: Mantém a ordem relativa de elementos iguais, o que é útil em certas aplicações.

Aplicabilidade em Listas e Grandes Dados: Bom para listas encadeadas e grandes conjuntos de dados onde o array não cabe completamente na memória (versões externas).

```
void mesclar(vector<int>& vetor, int esquerda, int meio, int
direita) {
    int n1 = meio - esquerda + 1;
    int n2 = direita - meio;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; ++i)
        L[i] = vetor[esquerda + i];
    for (int j = 0; j < n2; ++j)
        R[j] = vetor[meio + 1 + j];

    int i = 0, j = 0, k = esquerda;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            vetor[k] = L[i];
            ++i;
        } else {
```

```
            vetor[k] = R[j];
            ++j;
        }
        ++k;
    }

    while (i < n1) {
        vetor[k] = L[i];
        ++i;
        ++k;
    }

    while (j < n2) {
        vetor[k] = R[j];
        ++j;
        ++k;
    }
}

void mergeSort(vector<int>& vetor, int esquerda, int direita) {
    if (esquerda < direita) {
        int meio = esquerda + (direita - esquerda) / 2;
        mergeSort(vetor, esquerda, meio);
        mergeSort(vetor, meio + 1, direita);
        mesclar(vetor, esquerda, meio, direita);
    }
}

int main() {
    vector<int> vetor = {12, 11, 13, 5, 6, 7};
    mergeSort(vetor, 0, vetor.size() - 1);
    for (int numero : vetor) {
        cout << numero << " ";
    }
    return 0;
}
```

3- HeapSort

Tempo de Execução Garantido: Sempre O(nlogn), independentemente da disposição inicial dos elementos.

In-place: Requer apenas uma quantidade constante de espaço extra além do array original.

Não Recursivo: Implementação baseada em heap pode evitar a sobrecarga de chamada de função recursiva.

```
void heapify(vector<int>& vetor, int tamanho, int i) {
```

```
    int maior = i;
    int esquerda = 2 * i + 1;
    int direita = 2 * i + 2;

    if (esquerda < tamanho && vetor[esquerda] > vetor[maior])
        maior = esquerda;

    if (direita < tamanho && vetor[direita] > vetor[maior])
        maior = direita;

    if (maior != i) {
        swap(vetor[i], vetor[maior]);
        heapify(vetor, tamanho, maior);
    }
}

void heapSort(vector<int>& vetor) {
    int tamanho = vetor.size();

    for (int i = tamanho / 2 - 1; i >= 0; --i)
        heapify(vetor, tamanho, i);

    for (int i = tamanho - 1; i > 0; --i) {
        swap(vetor[0], vetor[i]);
        heapify(vetor, i, 0);
    }
}

int main() {
    vector<int> vetor = {4, 10, 3, 5, 1};
    heapSort(vetor);
    for (int numero : vetor) {
        cout << numero << " ";
    }
    return 0;
}
```

4- BubbleSort

Simplicidade: Muito fácil de entender e implementar.

Detecção de Lista Ordenada: Se a lista já está ordenada ou quase ordenada, BubbleSort pode ser eficiente com uma pequena modificação para detectar se houve trocas em uma passagem.

```
void bubbleSort(vector<int>& vetor) {
    int tamanho = vetor.size();
    for (int i = 0; i < tamanho - 1; ++i) {
        bool trocado = false;
```

```
        for (int j = 0; j < tamanho - i - 1; ++j) {
            if (vetor[j] > vetor[j + 1]) {
                swap(vetor[j], vetor[j + 1]);
                trocado = true;
            }
        }
        if (!trocado) break; // Se não houve trocas, a lista já
está ordenada
    }
}

int main() {
    vector<int> vetor = {64, 34, 25, 12, 22, 11, 90};
    bubbleSort(vetor);
    for (int numero : vetor) {
        cout << numero << " ";
    }
    return 0;
}
```

5- InsertionSort

Eficiente para Pequenos Conjuntos de Dados: Muito rápido para listas pequenas ou parcialmente ordenadas, com complexidade O(n) no melhor caso.

Estável: Preserva a ordem relativa de elementos iguais.

Simplicidade: Fácil de implementar e entender.

```
void insertionSort(vector<int>& vetor) {
    int tamanho = vetor.size();
    for (int i = 1; i < tamanho; ++i) {
        int chave = vetor[i];
        int j = i - 1;
        while (j >= 0 && vetor[j] > chave) {
            vetor[j + 1] = vetor[j];
            --j;
        }
        vetor[j + 1] = chave;
    }
}

int main() {
    vector<int> vetor = {12, 11, 13, 5, 6};
    insertionSort(vetor);
    for (int numero : vetor) {
        cout << numero << " ";
    }
```

```
    return 0;
}
```

6- SelectionSort

Simplicidade: Muito fácil de entender e implementar.

Número Fixo de Trocas: Executa no máximo n−1 trocas, o que pode ser útil em situações onde a troca é cara em termos de tempo.

```
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        swap(arr[i], arr[minIndex]);
    }
}

int main() {
    vector<int> arr = {64, 25, 12, 22, 11};
    selectionSort(arr);
    for (int num : arr) {
        cout << num << " ";
    }
    return 0;
}
```

Grafos

1. **Busca em Profundidade (DFS)**

Exploração de Grafos: DFS é usado para explorar todos os vértices e arestas de um grafo, começando por um vértice e explorando tão longe quanto possível antes de retroceder.

Caminhos e Componentes Conexos: Pode ser usado para encontrar componentes conexos, detectar ciclos e encontrar caminhos em grafos.

```
void dfsUtil(int vertice, vector<bool>& visitado, const
vector<vector<int>>& adj) {
    visitado[vertice] = true;
    cout << vertice << " ";

    for (int i : adj[vertice]) {
        if (!visitado[i]) {
            dfsUtil(i, visitado, adj);
        }
    }
}

void dfs(int numVertices, const vector<vector<int>>& adj) {
    vector<bool> visitado(numVertices, false);
    for (int i = 0; i < numVertices; ++i) {
        if (!visitado[i]) {
            dfsUtil(i, visitado, adj);
        }
    }
}

int main() {
    int numVertices = 4;
    vector<vector<int>> adj(numVertices);
    adj[0] = {1, 2};
    adj[1] = {0, 2, 3};
    adj[2] = {0, 1};
    adj[3] = {1};

    dfs(numVertices, adj);
    return 0;
}
```

2. **Busca em Largura (BFS)**

Exploração de Grafos: BFS explora todos os vértices ao mesmo nível antes de passar para o próximo nível, útil para encontrar o caminho mais curto em um grafo não ponderado.

Níveis e Componentes: Usado para descobrir componentes conexos e níveis em um grafo.

```
void bfs(int numVertices, const vector<vector<int>>& adj) {
    vector<bool> visitado(numVertices, false);
    queue<int> fila;

    for (int i = 0; i < numVertices; ++i) {
        if (!visitado[i]) {
            visitado[i] = true;
            fila.push(i);

            while (!fila.empty()) {
                int vertice = fila.front();
                fila.pop();
                cout << vertice << " ";

                for (int vizinho : adj[vertice]) {
                    if (!visitado[vizinho]) {
                        visitado[vizinho] = true;
                        fila.push(vizinho);
                    }
                }
            }
        }
    }
}

int main() {
    int numVertices = 4;
    vector<vector<int>> adj(numVertices);
    adj[0] = {1, 2};
    adj[1] = {0, 2, 3};
    adj[2] = {0, 1};
    adj[3] = {1};

    bfs(numVertices, adj);
    return 0;
}
```

3. **Algoritmo de Dijkstra**

Caminho Mínimo: Dijkstra é utilizado para encontrar o caminho mais curto a partir de um vértice de origem para todos os outros vértices em um grafo ponderado com arestas de peso não negativo.

```
void dijkstra(int numVertices, const vector<vector<pair<int, int>>>&
adj, int origem) {
    vector<int> distancia(numVertices, INT_MAX);
    set<pair<int, int>> conjunto; // (distância, vértice)

    distancia[origem] = 0;
    conjunto.insert({0, origem});

    while (!conjunto.empty()) {
        int u = conjunto.begin()->second;
        conjunto.erase(conjunto.begin());

        for (const auto& vizinho : adj[u]) {
            int v = vizinho.first;
            int peso = vizinho.second;

            if (distancia[u] + peso < distancia[v]) {
                conjunto.erase({distancia[v], v});
                distancia[v] = distancia[u] + peso;
                conjunto.insert({distancia[v], v});
            }
        }
    }

    cout << "Distâncias a partir do vértice " << origem << ":\n";
    for (int i = 0; i < numVertices; ++i) {
        if (distancia[i] == INT_MAX) {
            cout << i << ": infinito\n";
        } else {
            cout << i << ": " << distancia[i] << "\n";
        }
    }
}

int main() {
    int numVertices = 5;
    vector<vector<pair<int, int>>> adj(numVertices);

    adj[0].emplace_back(1, 10);
    adj[0].emplace_back(4, 3);
    adj[1].emplace_back(2, 2);
    adj[1].emplace_back(4, 4);
    adj[2].emplace_back(3, 9);
    adj[3].emplace_back(4, 7);
    adj[4].emplace_back(1, 1);
```

```
    adj[4].emplace_back(2, 8);

    dijkstra(numVertices, adj, 0);
    return 0;
}
```

Strings

1- Algoritmo de KMP (Knuth-Morris-Pratt) para busca de padrões em strings

O Algoritmo de KMP (Knuth-Morris-Pratt) é um método eficiente para buscar padrões em strings. Ele melhora a busca de padrões ao evitar comparações redundantes, que ocorrem em algoritmos mais simples como a busca ingênua.

```
// Função para construir a tabela de prefixos
vector<int> construirTabelaPrefixos(const string& padrao) {
    int m = padrao.length();
    vector<int> prefixo(m, 0);
    int j = 0; // comprimento do prefixo atual

    for (int i = 1; i < m; ++i) {
        while (j > 0 && padrao[i] != padrao[j]) {
            j = prefixo[j - 1];
        }
        if (padrao[i] == padrao[j]) {
            ++j;
        }
        prefixo[i] = j;
    }

    return prefixo;
}

// Função para realizar a busca KMP
void buscaKMP(const string& texto, const string& padrao) {
    int n = texto.length();
    int m = padrao.length();
    vector<int> prefixo = construirTabelaPrefixos(padrao);

    int j = 0; // índice para o padrão
    for (int i = 0; i < n; ++i) {
```

```
        while (j > 0 && texto[i] != padrao[j]) {
            j = prefixo[j - 1];
        }
        if (texto[i] == padrao[j]) {
            ++j;
        }
        if (j == m) {
            cout << "Padrão encontrado na posição " << i - m + 1 <<
endl;
            j = prefixo[j - 1];
        }
    }
}

int main() {
    string texto = "ABABDABACDABABCABAB";
    string padrao = "ABABCABAB";
    buscaKMP(texto, padrao);
    return 0;
}
```

2- Expansão ao Redor do Centro: maior palíndromo

```
string maiorPalindromo(const string& str) {
    int n = str.length();
    if (n == 0) return "";

    string maior;

    for (int i = 0; i < n; ++i) {
        // Expansão em torno do centro único (caractere)
        int inicio = i, fim = i;
        while (inicio >= 0 && fim < n && str[inicio] == str[fim]) {
            if (fim - inicio + 1 > maior.length()) {
                maior = str.substr(inicio, fim - inicio + 1);
            }
            --inicio;
            ++fim;
        }

        // Expansão em torno do centro duplo (entre caracteres)
        inicio = i, fim = i + 1;
        while (inicio >= 0 && fim < n && str[inicio] == str[fim]) {
            if (fim - inicio + 1 > maior.length()) {
                maior = str.substr(inicio, fim - inicio + 1);
            }
            --inicio;
            ++fim;
```

```
        }
    }

    return maior;
}

int main() {
    string str = "babad";
    string resultado = maiorPalindromo(str);
    cout << "O maior palíndromo em " << str << " é " << resultado <<
endl;
    return 0;
}
```

Outros

1. **Algoritmo de Euclides (MDC - Máximo Divisor Comum)**

```
int mdc(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main() {
    int a = 48;
    int b = 18;
    cout << "MDC de " << a << " e " << b << " é " << mdc(a, b) << endl;
    return 0;
}
```

2- **Algoritmo de Exponenciação Rápida (Exponentiation by Squaring)**

```
long long exponenciacaoRapida(long long base, int expoente) {
```

```
    long long resultado = 1;
    while (expoente > 0) {
        if (expoente % 2 == 1) {
            resultado *= base;
        }
        base *= base;
        expoente /= 2;
    }
    return resultado;
}

int main() {
    long long base = 2;
    int expoente = 10;
    cout << base << "^" << expoente << " é " <<
exponenciacaoRapida(base, expoente) << endl;
    return 0;
}
```

3- **Busca Ternária**

Busca em Arrays Ordenados: A busca ternária divide o intervalo de busca em três partes, comparando com dois pontos e é útil para encontrar um elemento em um array ordenado. Ela é mais eficiente que a busca binária em alguns casos específicos.

```
int buscaTernaria(const vector<int>& arr, int chave, int esquerda, int direita) {
   if (direita >= esquerda) {
      int terço1 = esquerda + (direita - esquerda) / 3;
      int terço2 = direita - (direita - esquerda) / 3;

      if (arr[terço1] == chave) return terço1;
      if (arr[terço2] == chave) return terço2;

      if (chave < arr[terço1]) {
         return buscaTernaria(arr, chave, esquerda, terço1 - 1);
      } else if (chave > arr[terço2]) {
         return buscaTernaria(arr, chave, terço2 + 1, direita);
      } else {
         return buscaTernaria(arr, chave, terço1 + 1, terço2 - 1);
      }
   }
   return -1;
}

int main() {
   vector<int> arr = {1, 4, 7, 10, 13, 16, 19, 22, 25, 28};
   int chave = 16;
```

```
    int resultado = buscaTernaria(arr, chave, 0, arr.size() - 1);
    if (resultado != -1) {
        cout << "Chave " << chave << " encontrada na posição " << resultado << endl;
    } else {
        cout << "Chave " << chave << " não encontrada" << endl;
    }
    return 0;
}
```

4- Janela deslizante

O algoritmo de janela deslizante é usado para encontrar uma solução para problemas que envolvem subarrays ou substrings de tamanho fixo ou variável. Ele usa uma "janela" que se move através da estrutura de dados para realizar cálculos ou buscar informações sem precisar recalcular para cada possível subarray ou substring do início ao fim.

Como Funciona?

- **Janela Fixa**: Quando o tamanho da janela é fixo, você começa com uma janela de tamanho fixo e a move através da estrutura de dados. Você atualiza a solução (como a soma ou o máximo) conforme a janela se move.
- **Janela Variável**: Quando o tamanho da janela é variável, a janela é ajustada dinamicamente para encontrar a solução ótima. A janela pode crescer ou encolher com base em certas condições.

```
vector<int> maximoJanelaDeslizante(const vector<int>& numeros, int
tamanhoJanela) {
    vector<int> valoresMaximos;
    deque<int> filaDupla;

    for (int i = 0; i < numeros.size(); ++i) {
        // Remove elementos fora da janela
        while (!filaDupla.empty() && filaDupla.front() <= i -
tamanhoJanela) {
            filaDupla.pop_front();
        }
        // Remove elementos menores do que o elemento atual
        while (!filaDupla.empty() && numeros[filaDupla.back()] <=
numeros[i]) {
            filaDupla.pop_back();
        }
        filaDupla.push_back(i);
        // O máximo da janela é o primeiro elemento da fila dupla
        if (i >= tamanhoJanela - 1) {
```

```
            valoresMaximos.push_back(numeros[filaDupla.front()]);
        }
    }

    return valoresMaximos;
}

int main() {
    vector<int> numeros = {1, 3, -1, -3, 5, 3, 6, 7};
    int tamanhoJanela = 3;
    vector<int> resultado = maximoJanelaDeslizante(numeros,
tamanhoJanela);

    cout << "Máximos de subarrays de tamanho " << tamanhoJanela << ": ";
    for (int valor : resultado) {
        cout << valor << " ";
    }
    cout << endl;

    return 0;
}
```

```
int menorSubarraySomaMaiorOuIgual(const vector<int>& numeros, int alvo)
{
    int inicio = 0;
    int somaAtual = 0;
    int comprimentoMinimo = INT_MAX;

    for (int fim = 0; fim < numeros.size(); ++fim) {
        somaAtual += numeros[fim];

        while (somaAtual >= alvo) {
            comprimentoMinimo = min(comprimentoMinimo, fim - inicio +
1);
            somaAtual -= numeros[inicio++];
        }
    }

    return comprimentoMinimo == INT_MAX ? 0 : comprimentoMinimo;
}

int main() {
    vector<int> numeros = {2, 3, 1, 2, 4, 3};
    int alvo = 7;
    int resultado = menorSubarraySomaMaiorOuIgual(numeros, alvo);

    if (resultado > 0) {
        cout << "O comprimento do menor subarray cuja soma é pelo menos
" << alvo << " é " << resultado << endl;
```

```
    } else {
        cout << "Não há subarray cuja soma seja pelo menos " << alvo <<
endl;
    }

    return 0;
}
```

Exemplo: Encontrar a sub-string com maior soma em uma string.
cpp
Copiar código

```cpp
int max_sum = 0, window_sum = 0, l = 0;
for (int r = 0; r < n; r++) {
    window_sum += arr[r];
    while (window_sum > max_sum) {
        max_sum = window_sum;
        window_sum -= arr[l];
        l++;
    }
}
```

Teoria dos números

1. **Algoritmo de Exponenciação Modular**

```
long long exponenciacaoModular(long long base, long long expoente, long
long modulo) {
    long long resultado = 1;
    base = base % modulo;
    while (expoente > 0) {
        if (expoente % 2 == 1) {
            resultado = (resultado * base) % modulo;
        }
        expoente = expoente >> 1;
        base = (base * base) % modulo;
    }
    return resultado;
}

int main() {
```

```
    long long base = 5;
    long long expoente = 3;
    long long modulo = 13;
    cout << base << "^" << expoente << " mod " << modulo << " é " <<
exponenciacaoModular(base, expoente, modulo) << endl;
    return 0;
}
```

Funções úteis

1. Funções Úteis de `vector`

1.1. Inicialização e Acesso

Criar um `vector` com valores iniciais:

```
vector<int> v = {1, 2, 3, 4, 5};
```

- `vector<int> v(10, 5); // 10 elementos, todos com valor 5`
- **Acessar elementos:**

  `int x = v[0]; // Acesso direto`
- `int y = v.at(2); // Acesso com verificação de limites`
- `int z = v.front(); // Primeiro elemento`
- `int w = v.back(); // Último elemento`

1.2. Inserção e Remoção

Adicionar elementos:

```
v.push_back(6); // Adiciona ao final
```

- `v.insert(v.begin() + 1, 10); // Insere 10 na posição 1`

Remover elementos:

```
v.pop_back(); // Remove o último elemento
v.erase(v.begin() + 2); // Remove o elemento na posição 2
```

- `v.clear(); // Remove todos os elementos`

1.3. Ordenação e Busca

- **Ordenar `vector`:**

  ```
  sort(v.begin(), v.end()); // Ordem crescente
  ```
- `sort(v.begin(), v.end(), greater<int>()); // Ordem decrescente`

Busca binária (vector deve estar ordenado):

```
bool found = binary_search(v.begin(), v.end(), 3);
```

- `auto it = lower_bound(v.begin(), v.end(), 3); // Primeiro >= 3`
- `auto it2 = upper_bound(v.begin(), v.end(), 3); // Primeiro > 3`

1.4. Outras Operações

Obter o tamanho:

```
int size = v.size();
```

- `bool empty = v.empty(); // Verifica se está vazio`

Redimensionar:

```
v.resize(8); // Redimensiona para 8 elementos
```

- `v.resize(10, 0); // Redimensiona para 10 elementos, novos elementos são 0`

Reverter a ordem:

```
reverse(v.begin(), v.end());
```

-
- **Obter um sub-vetor:**

  ```
  vector<int> subvector(v.begin() + 2, v.begin() + 5); // Elementos 2 a 4
  ```

2. Funções Úteis de `string`

2.1. Inicialização e Acesso

Criar uma `string`:

```
string s = "Hello";
```

- `string s2(10, 'a'); // "aaaaaaaaaa"`

Acessar caracteres:

```
char c = s[1];
```

- `char d = s.at(2); // Acesso com verificação de limites`

2.2. Concatenação e Modificação

Concatenar `strings`:

```
string s3 = s + " World";
s += "!";
```

-

Inserir e Remover:

```
s.insert(5, "X"); // Insere "X" na posição 5
```

- `s.erase(1, 3); // Remove 3 caracteres a partir da posição 1`

2.3. Busca e Comparação

- **Encontrar substrings:**

  ```
  size_t pos = s.find("lo"); // Retorna posição ou string::npos
  ```
- `size_t pos2 = s.rfind("l"); // Última ocorrência`
- **Substituir substrings:**

  ```
  s.replace(0, 5, "Hi"); // Substitui os primeiros 5 caracteres por "Hi"
  ```
- **Comparar strings:**

  ```
  int cmp = s.compare("Hello"); // Retorna 0 se iguais, <0 se menor, >0 se maior
  ```

2.4. Substrings e Conversão

- **Obter substring:**

  ```
  string sub = s.substr(1, 3); // 3 caracteres a partir da posição 1
  ```

Conversão de string para número:

```
int num = stoi("123");
longlong big_num = stoll("123456789012345");
float f = stof("123.45");
double d = stod("123.456");
```

-

Conversão de número para string:

```
string num_str = to_string(123);
```

-

3. Funções Úteis de `set` e `map`

3.1. `set`

Inserção e Remoção:

```
set<int> s;
s.insert(5);
```

- `s.erase(5); // Remove o elemento 5`
- **Busca e Checagem:**

  ```
  bool exists = s.find(5) != s.end(); // Verifica se 5 está no setbool contains = s.count(5); // Retorna 1 se 5 está no set, 0 se não está
  ```

3.2. `map`

Inserção e Acesso:

```
map<string, int> m;
m["one"] = 1;
int value = m["one"];
```

-

Remover Elementos:

```
m.erase("one");
```

-

Busca:

```
auto it = m.find("two");
if (it != m.end()) {
    int val = it->second;
}
```

-

4. Funções Úteis de `algorithm`

4.1. Operações em Containers

Encontrar o mínimo/máximo:

```
int min_val = *min_element(v.begin(), v.end());
int max_val = *max_element(v.begin(), v.end());
```

-

Soma de todos os elementos:

```
int sum = accumulate(v.begin(), v.end(), 0);
```

-

Contar elementos com certa condição:

```
int count_fives = count(v.begin(), v.end(), 5);
```

-

Remover duplicatas em `vector`:

```
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

-

4.2. Funções Matemáticas

Cálculo de GCD e LCM:

```
int gcd_val = __gcd(24, 18); // 6int lcm_val = (a * b) /
__gcd(a, b);
```

- **Potência e Raiz Quadrada:**
  cpp
  Copiar código
  `int pow_val = pow(2, 3); // 2^3 = 8`
- `double sqrt_val = sqrt(16.0); // 4.0`

5. Funções de Entrada e Saída Eficientes

5.1. Entrada e Saída Rápida

Otimizar IO para problemas de grandes volumes de dados:

```
ios::sync_with_stdio(false);
cin.tie(0);
cout.tie(0);
```

-

5.2. Entrada de Linhas Inteiras

Entrada de uma linha inteira de dados:

```
string line;
```

- `getline(cin, line);`

1. `deque` (Double-Ended Queue)

O `deque` é um contêiner de sequência que permite inserção e remoção eficientes em ambas as extremidades (frente e traseira).

Inicialização e Acesso

Criar e inicializar:

```
deque<int> d = {1, 2, 3, 4, 5}; // Inicializa o deque com valores
```

-

Acessar elementos:

```
int x = d[0]; // Acesso direto ao primeiro elemento

int y = d.at(2); // Acesso com verificação de limites

int z = d.front(); // Acesso ao primeiro elemento

int w = d.back(); // Acesso ao último elemento
```

-

Inserção e Remoção

Adicionar e remover elementos:

```
d.push_back(6); // Adiciona 6 ao final

d.push_front(0); // Adiciona 0 ao início

d.pop_back(); // Remove o último elemento

d.pop_front(); // Remove o primeiro elemento
```

-

Inserir e remover em posições específicas:

```
d.insert(d.begin() + 2, 10); // Insere 10 na posição 2

d.erase(d.begin() + 3); // Remove o elemento na posição 3
```

-

Outras Operações

Tamanho e capacidade:

```
int size = d.size(); // Retorna o número de elementos

bool empty = d.empty(); // Verifica se o deque está vazio
```

-

Redimensionar:

```
d.resize(8); // Redimensiona o deque para 8 elementos

d.resize(5, 0); // Redimensiona para 5 elementos, novos elementos são 0
```

-

2. `stack`

O `stack` é um contêiner adaptador que implementa uma pilha (LIFO - Last In First Out). Ele utiliza por padrão um `deque` ou `vector` como base.

Inicialização

Criar e inicializar:

```
stack<int> st; // Pilha vazia
```

-

Operações Básicas

Adicionar e remover elementos:

```
st.push(1); // Adiciona 1 ao topo

st.push(2); // Adiciona 2 ao topo

st.pop(); // Remove o elemento do topo (2)
```

-

Acesso ao topo:

```
int top = st.top(); // Acessa o elemento do topo (1 após o último pop)
```

-

Tamanho e verificação:

```
int size = st.size(); // Retorna o número de elementos na pilha

bool empty = st.empty(); // Verifica se a pilha está vazia
```

-

3. queue

O `queue` é um contêiner adaptador que implementa uma fila (FIFO - First In First Out). Ele utiliza por padrão um `deque` como base.

Inicialização

Criar e inicializar:

```
queue<int> q; // Fila vazia
```

-

Operações Básicas

Adicionar e remover elementos:

```
q.push(1); // Adiciona 1 ao final

q.push(2); // Adiciona 2 ao final

q.pop(); // Remove o elemento do início (1)
```

-

Acesso ao início e final:

```
int front = q.front(); // Acessa o elemento do início (2 após o
último pop)

int back = q.back(); // Acessa o elemento do final (2)
```

-

Tamanho e verificação:

```
int size = q.size(); // Retorna o número de elementos na fila
```

- `bool empty = q.empty(); // Verifica se a fila está vazia`