Algoritmos para programação competitiva

Henrique Souza

August 27, 2024

Abstract

Your abstract.

1 Introduction

Your introduction goes here! Simply start writing your document and use the Recompile button to view the updated PDF preview. Examples of commonly used commands and features are listed below, to help you get started.

Once you're familiar with the editor, you can find various project settings in the Overleaf menu, accessed via the button in the very top left of the editor. To view tutorials, user guides, and further documentation, please visit our help library, or head to our plans page to choose your plan.

2 Algoritmos

2.1 Searching & Sorting Algorithms

2.1.1 Binary Search

Quando usar: Use o Binary Search quando você tem um array ou uma lista já ordenada e precisa encontrar rapidamente a posição de um elemento específico.
Vantagem: Muito eficiente para buscas em dados ordenados, com complexidade de tempo O(log n).
Limitação: Requer que os dados estejam ordenados.

```
#include <iostream>
#include <vector>
#include <algorithm> // Para a fun   o std::sort, se necess rio

template <typename T>
int binarySearch(const std::vector<T>& arr, const T& key) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2; // Evitar overflow

        // Verifica se o elemento no meio    igual    chave
        if (arr[mid] == key)
            return mid;

        // Se a chave for maior, ignora a metade esquerda
        if (arr[mid] < key)
            left = mid + 1;
        // Se a chave for menor, ignora a metade direita
        else
            right = mid - 1;
    }

    // Retorna -1 se o elemento  n  o  for encontrado
    return -1;
}

int main() {
```

```
    std::vector<int> arr = {2, 3, 4, 10, 40};
    int key = 10;

    // Se o array n o estiver ordenado, voc  pode orden -lo
    // std::sort(arr.begin(), arr.end());

    int result = binarySearch(arr, key);
    if (result != -1)
        std::cout << "Elemento encontrado no  ndice : " << result << std::endl;
    else
        std::cout << "Elemento n o encontrado no array." << std::endl;

    return 0;
}
```

Listing 1: binary search

2.1.2 QuickSort

Quando usar: Use o QuickSort quando precisar ordenar um array ou lista. É um dos algoritmos de ordenação mais eficientes e amplamente utilizados.
Vantagem: Geralmente mais rápido que outros algoritmos de ordenação, com complexidade média de O(n log n).
Limitação: No pior caso, pode ter complexidade $O(n^2)$, mas isso é raro e pode ser mitigado com boas escolhas de pivô.

```
#include <iostream>
#include <vector>
#include <algorithm> // Para a fun  o std::swap

template <typename T>
int partition(std::vector<T>& arr, int low, int high) {
    T pivot = arr[high]; // Escolhe o  ltimo  elemento como piv
    int i = low - 1;      //  ndice  do menor elemento

    for (int j = low; j < high; ++j) {
        if (arr[j] < pivot) {
            ++i;
            std::swap(arr[i], arr[j]); // Coloca o elemento menor que o piv  na
posi  o correta
        }
    }
    std::swap(arr[i + 1], arr[high]); // Coloca o piv  na posi  o correta
    return i + 1;
}

template <typename T>
void quickSort(std::vector<T>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); //  ndice  de parti  o

        // Ordena os elementos antes e depois da parti  o
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

template <typename T>
void quickSort(std::vector<T>& arr) {
    if (!arr.empty()) {
        quickSort(arr, 0, arr.size() - 1);
    }
}

int main() {
    std::vector<int> arr = {10, 7, 8, 9, 1, 5};

    quickSort(arr);
```

```
    std::cout << "Array ordenado: ";
    for (const auto& elem : arr) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 2: quick sort

2.1.3 MergeSort

Quando usar: Use o MergeSort quando precisar de um algoritmo de ordenação estável ou ao trabalhar com grandes quantidades de dados. É útil quando você precisa garantir a complexidade de tempo O(n log n) no pior caso.
Vantagem: Sempre tem complexidade O(n log n) e é estável (não altera a ordem relativa dos elementos iguais).
Limitação: Requer espaço adicional para o array temporário, o que pode ser um problema em ambientes com memória limitada.

```
#include <iostream>
#include <vector>

template <typename T>
void merge(std::vector<T>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Vetores tempor rios para armazenar as metades
    std::vector<T> leftArr(n1);
    std::vector<T> rightArr(n2);

    // Copia os dados para os vetores tempor rios
    for (int i = 0; i < n1; ++i)
        leftArr[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        rightArr[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    // Mescla os vetores tempor rios de volta ao vetor original
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            ++i;
        } else {
            arr[k] = rightArr[j];
            ++j;
        }
        ++k;
    }

    // Copia os elementos restantes de leftArr, se houver
    while (i < n1) {
        arr[k] = leftArr[i];
        ++i;
        ++k;
    }

    // Copia os elementos restantes de rightArr, se houver
    while (j < n2) {
        arr[k] = rightArr[j];
        ++j;
        ++k;
    }
}

template <typename T>
```

```
void mergeSort(std::vector<T>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Ordena a primeira e a segunda metade
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Mescla as metades ordenadas
        merge(arr, left, mid, right);
    }
}

template <typename T>
void mergeSort(std::vector<T>& arr) {
    if (!arr.empty()) {
        mergeSort(arr, 0, arr.size() - 1);
    }
}

int main() {
    std::vector<int> arr = {12, 11, 13, 5, 6, 7};

    mergeSort(arr);

    std::cout << "Array ordenado: ";
    for (const auto& elem : arr) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 3: merge sort

2.1.4 Counting Sort

Quando usar: Use o Counting Sort quando você tem uma faixa limitada de valores inteiros e precisa de uma ordenação muito rápida.
Vantagem: Extremamente eficiente para dados com um intervalo pequeno e conhecido, com complexidade O(n + k), onde k é o valor máximo no array.
Limitação: Não é comparativo, e o espaço necessário pode ser grande se os dados tiverem um intervalo amplo.

```
#include <iostream>
#include <vector>
#include <algorithm> // Para a fun   o std::max e std::min

template <typename T>
void countingSort(std::vector<T>& arr) {
    if (arr.empty()) return;

    // Encontra o valor m ximo e m nimo no array
    T minVal = *std::min_element(arr.begin(), arr.end());
    T maxVal = *std::max_element(arr.begin(), arr.end());

    // Calcula o tamanho do array de contagem
    int range = maxVal - minVal + 1;
    std::vector<int> count(range, 0);

    // Conta a ocorr ncia de cada elemento
    for (const T& num : arr) {
        ++count[num - minVal];
    }

    // Modifica o array original com base nas contagens
    int index = 0;
    for (int i = 0; i < range; ++i) {
```

```
            while (count[i]-- > 0) {
                arr[index++] = i + minVal;
            }
        }
    }

    int main() {
        std::vector<int> arr = {4, 2, 2, 8, 3, 3, 1};

        countingSort(arr);

        std::cout << "Array ordenado: ";
        for (const auto& elem : arr) {
            std::cout << elem << " ";
        }
        std::cout << std::endl;

        return 0;
    }
```

Listing 4: counting sort

2.1.5 KMP

Quando usar: Use o algoritmo KMP para buscar padrões em strings, especialmente quando você precisa fazer múltiplas buscas no mesmo texto ou quando o padrão tem partes repetitivas.
Vantagem: Eficiência de O(n + m), que é melhor do que a busca ingênua, especialmente em textos grandes com padrões complexos.
Limitação: Pode ser mais complexo de implementar e entender em comparação com a busca de padrão ingênua.

```
#include <iostream>
#include <vector>
#include <string>

// Fun  o para calcular o array de prefixo, tamb m conhecido como LPS (Longest
Prefix Suffix)
std::vector<int> computeLPSArray(const std::string& pattern) {
    int m = pattern.length();
    std::vector<int> lps(m, 0); // Array de LPS
    int length = 0; // Comprimento do maior prefixo que tamb m    sufixo
    int i = 1;

    while (i < m) {
        if (pattern[i] == pattern[length]) {
            ++length;
            lps[i] = length;
            ++i;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                ++i;
            }
        }
    }
    return lps;
}

// Fun  o KMP para buscar um padr o em um texto
void KMPSearch(const std::string& pattern, const std::string& text) {
    int m = pattern.length();
    int n = text.length();

    // Calcula o array de LPS
    std::vector<int> lps = computeLPSArray(pattern);

    int i = 0; //  ndice  para o texto
```

```
    int j = 0; //  ndice  para o padr o

    while (i < n) {
        if (pattern[j] == text[i]) {
            ++i;
            ++j;
        }

        if (j == m) {
            std::cout << "Padr o encontrado no  ndice  " << (i - j) << std::endl;
            j = lps[j - 1];
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                ++i;
            }
        }
    }
}

int main() {
    std::string text = "ABABDABACDABABCABAB";
    std::string pattern = "ABABCABAB";

    KMPSearch(pattern, text);

    return 0;
}
```

Listing 5: KMP

2.2 Dynamic Programming

2.2.1 Longest Common Subsequence

Quando usar: Use o LCS quando precisar encontrar a subsequência comum mais longa entre duas sequências. Isso é útil em várias aplicações, como comparação de strings (por exemplo, para análise de texto ou alinhamento de sequências em bioinformática), e em comparação de versões de arquivos. Exemplo de uso: Encontrar a similaridade entre dois documentos de texto ou comparar versões de código fonte.

```
#include <iostream>
#include <vector>
#include <algorithm>

// Fun  o para calcular o Longest Common Subsequence (LCS)
template<typename T>
std::vector<T> longestCommonSubsequence(const std::vector<T>& seq1, const std::
vector<T>& seq2) {
    int m = seq1.size();
    int n = seq2.size();

    // Criando uma tabela para armazenar os resultados das subsolu  es
    std::vector<std::vector<int>> dp(m + 1, std::vector<int>(n + 1, 0));

    // Construindo a tabela dp em forma bottom-up
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (seq1[i - 1] == seq2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = std::max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    // Reconstituindo a subsequ ncia comum
    std::vector<T> lcs;
```

```
    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (seq1[i - 1] == seq2[j - 1]) {
            lcs.push_back(seq1[i - 1]);
            --i;
            --j;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            --i;
        } else {
            --j;
        }
    }

    // A subsequ ncia foi constru da de tr s para frente, ent o invert -la
    std::reverse(lcs.begin(), lcs.end());

    return lcs;
}

int main() {
    // Exemplo com strings
    std::string str1 = "AGGTAB";
    std::string str2 = "GXTXAYB";

    std::vector<char> seq1(str1.begin(), str1.end());
    std::vector<char> seq2(str2.begin(), str2.end());

    std::vector<char> lcs = longestCommonSubsequence(seq1, seq2);

    std::cout << "Longest Common Subsequence: ";
    for (char c : lcs) {
        std::cout << c;
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 6: Longest Common Subsequence

2.2.2 Longest Increasing Subsequence

Quando usar: Use o problema de Subset Sum quando precisar determinar se há um subconjunto de números que soma exatamente um valor alvo. É um problema fundamental em otimização e é usado em contextos como alocação de recursos e planejamento financeiro.
Exemplo de uso: Verificar se é possível dividir um orçamento específico entre vários itens de forma que o total gasto seja exatamente igual ao orçamento.

```
#include <iostream>
#include <vector>
#include <algorithm>

// Fun   o para calcular o Longest Increasing Subsequence (LIS)
template<typename T>
std::vector<T> longestIncreasingSubsequence(const std::vector<T>& seq) {
    if (seq.empty()) return {};

    int n = seq.size();

    // Vetor para armazenar o  ndice  do predecessor do elemento na subsequ ncia
de LIS
    std::vector<int> prev(n, -1);

    // Vetor para armazenar os  ndices  da subsequ ncia de LIS encontrada
    std::vector<int> lis;

    // Vetor para armazenar as  ltimas  posi  es onde o valor T ocorre nas
subsequ ncias de LIS de diferentes comprimentos
    std::vector<int> lastIndex(n + 1, 0);
    int length = 1;  // Inicialmente a LIS tem comprimento 1
```

```
    lis.push_back(0);  // O primeiro elemento da sequ ncia    a primeira
subsequ ncia de comprimento 1

    for (int i = 1; i < n; ++i) {
        if (seq[i] > seq[lis.back()]) {
            // Se seq[i]    maior que o  ltimo  elemento da atual LIS, podemos
estender a LIS
            prev[i] = lis.back();
            lis.push_back(i);
            length++;
        } else {
            // Encontre a posi  o na subsequ ncia onde seq[i] pode substituir
um valor maior ou igual
            int pos = std::lower_bound(lis.begin(), lis.end(), i,
                                       [&seq](int idx1, int idx2) { return seq[
idx1] < seq[idx2]; }) - lis.begin();
            if (seq[i] < seq[lis[pos]]) {
                lis[pos] = i;
                if (pos > 0) {
                    prev[i] = lis[pos - 1];
                }
            }
        }
    }

    // Reconstru  o da subsequ ncia crescente
    std::vector<T> result(length);
    int k = lis.back();
    for (int i = length - 1; i >= 0; --i) {
        result[i] = seq[k];
        k = prev[k];
    }

    return result;
}

int main() {
    // Exemplo com uma sequ ncia de inteiros
    std::vector<int> seq = {10, 22, 9, 33, 21, 50, 41, 60, 80};

    std::vector<int> lis = longestIncreasingSubsequence(seq);

    std::cout << "Longest Increasing Subsequence: ";
    for (int num : lis) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 7: Longest Increasing Subsequence

2.2.3 Subset Sum Problem

Quando usar: Use o problema de Subset Sum quando precisar determinar se há um subconjunto de números que soma exatamente um valor alvo. É um problema fundamental em otimização e é usado em contextos como alocação de recursos e planejamento financeiro.
Exemplo de uso: Verificar se é possível dividir um orçamento específico entre vários itens de forma que o total gasto seja exatamente igual ao orçamento

```
#include <iostream>
#include <vector>

// Fun  o para verificar se existe um subconjunto com a soma desejada
template<typename T>
bool subsetSum(const std::vector<T>& set, T targetSum) {
    int n = set.size();
```

```
    // Matriz dp onde dp[i][j] ser  true se existir um subconjunto dos primeiros
i elementos com soma j
    std::vector<std::vector<bool>> dp(n + 1, std::vector<bool>(targetSum + 1,
false));

    // Para soma 0, sempre existe um subconjunto vazio
    for (int i = 0; i <= n; ++i) {
        dp[i][0] = true;
    }

    // Preenchendo a matriz dp de forma bottom-up
    for (int i = 1; i <= n; ++i) {
        for (T j = 1; j <= targetSum; ++j) {
            if (set[i - 1] > j) {
                dp[i][j] = dp[i - 1][j];
            } else {
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
            }
        }
    }

    return dp[n][targetSum];
}

int main() {
    // Exemplo com uma sequ ncia de inteiros
    std::vector<int> set = {3, 34, 4, 12, 5, 2};
    int targetSum = 9;

    if (subsetSum(set, targetSum)) {
        std::cout << "Existe um subconjunto com a soma " << targetSum << std::endl
;
    } else {
        std::cout << "N o existe um subconjunto com a soma " << targetSum << std
::endl;
    }

    return 0;
}
```

Listing 8: Subset Sum Problem

2.2.4 0–1 Knapsack Problem

Quando usar: Use o problema da mochila 0–1 quando precisar escolher um subconjunto de itens com valores e pesos específicos para maximizar o valor total sem exceder uma capacidade limitada. É amplamente usado em problemas de otimização onde há restrições de capacidade ou orçamento.
Exemplo de uso: Determinar quais itens levar em uma mochila de capacidade limitada para maximizar o valor total, como em problemas de otimização de recursos em logística ou planejamento de projetos.

```
#include <iostream>
#include <vector>
#include <algorithm>

// Fun  o para resolver o 0 1  Knapsack Problem
template<typename T>
T knapsack(const std::vector<T>& weights, const std::vector<T>& values, T capacity
) {
    int n = weights.size();

    // Matriz dp onde dp[i][w] ser  o valor m ximo que pode ser obtido com os
primeiros i itens e capacidade w
    std::vector<std::vector<T>> dp(n + 1, std::vector<T>(capacity + 1, 0));

    // Preenchendo a matriz dp de forma bottom-up
    for (int i = 1; i <= n; ++i) {
        for (T w = 1; w <= capacity; ++w) {
            if (weights[i - 1] <= w) {
                dp[i][w] = std::max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] +
values[i - 1]);
```

```
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    // O valor m ximo que pode ser obtido com a capacidade dada    armazenado em
dp[n][capacity]
    return dp[n][capacity];
}

int main() {
    // Exemplo com pesos e valores inteiros
    std::vector<int> weights = {10, 20, 30};
    std::vector<int> values = {60, 100, 120};
    int capacity = 50;

    int max_value = knapsack(weights, values, capacity);

    std::cout << "O valor m ximo que pode ser obtido    " << max_value << std::
endl;

    return 0;
}
```

Listing 9: 0–1 Knapsack Problem

2.2.5 Minimum Partition

Quando usar: Use o problema da partição mínima quando precisar dividir um conjunto de números em dois subconjuntos de forma que a diferença entre suas somas seja mínima. É útil em problemas de balanceamento e divisão de recursos.
Exemplo de uso: Dividir um conjunto de itens em duas partes com a menor diferença possível em termos de peso ou valor, como em problemas de carga balanceada ou alocação equitativa de recursos.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

// Fun   o para encontrar a parti   o m nima
template<typename T>
T minimumPartition(const std::vector<T>& set) {
    T totalSum = std::accumulate(set.begin(), set.end(), static_cast<T>(0));
    int n = set.size();

    // A metade da soma total    o m ximo valor de soma que precisamos considerar
    T target = totalSum / 2;

    // Matriz dp onde dp[i][j] ser  true se uma soma j pode ser obtida usando os
primeiros i itens
    std::vector<std::vector<bool>> dp(n + 1, std::vector<bool>(target + 1, false))
;

    // Inicializa  o:    sempre poss vel ter uma soma de 0 com um subconjunto
vazio
    for (int i = 0; i <= n; ++i) {
        dp[i][0] = true;
    }

    // Preenchendo a matriz dp de forma bottom-up
    for (int i = 1; i <= n; ++i) {
        for (T j = 1; j <= target; ++j) {
            if (set[i - 1] <= j) {
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
```

```
    // Encontrar o maior valor de soma que pode ser obtido que    menor ou igual a
 target
    T sum1 = 0;
    for (T j = target; j >= 0; --j) {
        if (dp[n][j]) {
            sum1 = j;
            break;
        }
    }

    // A soma m nima de parti   o    a diferen a entre a soma total e duas
vezes a maior soma poss vel que    <= target
    T sum2 = totalSum - sum1;
    return std::abs(sum2 - sum1);
}

int main() {
    // Exemplo com uma sequ ncia de inteiros
    std::vector<int> set = {1, 6, 11, 5};

    int min_partition = minimumPartition(set);

    std::cout << "A diferen a m nima de parti   o    " << min_partition << std
::endl;

    return 0;
}
```

Listing 10: Minimum Partition

2.3 Algoritmos importantes e Programação dinamica

1. Memoização (Cache) Memoização é uma técnica em que os resultados de subproblemas são armazenados em uma estrutura de dados (geralmente um array ou um dicionário) para que, quando o mesmo subproblema for encontrado novamente, o resultado possa ser retornado diretamente do cache, em vez de ser recalculado.

Exemplo: Fibonacci com Memoização: O problema de Fibonacci é um exemplo clássico para ilustrar a memoização. O enunciado é encontrar o enésimo número na sequência de Fibonacci, onde cada número é a soma dos dois números anteriores, e os dois primeiros números são 0 e 1.

```
#include <iostream>
#include <vector>

std::vector<int> memo; // Vetor para armazenar os resultados dos subproblemas

int fibonacci(int n) {
    if (n <= 1) return n; // Casos base
    if (memo[n] != -1) return memo[n]; // Retorna o valor se j   estiver no cache

    // Calcula e armazena o resultado no cache
    memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
    return memo[n];
}

int main() {
    int n = 50; // Tamanho m ximo desejado
    memo.resize(n + 1, -1); // Inicializa o cache com -1

    std::cout << "Fibonacci(" << n << ") = " << fibonacci(n) << std::endl;
    return 0;
}
```

Listing 11: Fibonacci com Memoização

2. Programação Dinâmica Tabular Programação dinâmica tabular utiliza uma tabela (geralmente uma matriz ou array) para armazenar os resultados dos subproblemas. A ideia é preencher a tabela de forma bottom-up, começando pelos subproblemas mais simples e subindo até o problema original.

Exemplo: Fibonacci com Programação Dinâmica Tabular: Este exemplo usa um array para armazenar os valores de Fibonacci calculados anteriormente.

```
#include <iostream>
#include <vector>

int fibonacci(int n) {
    std::vector<int> table(n + 1, 0); // Cria uma tabela para armazenar os
resultados
    table[0] = 0;
    if (n > 0) table[1] = 1;

    for (int i = 2; i <= n; ++i) {
        table[i] = table[i - 1] + table[i - 2]; // Preenche a tabela
    }

    return table[n];
}

int main() {
    int n = 50;
    std::cout << "Fibonacci(" << n << ") = " << fibonacci(n) << std::endl;
    return 0;
}
```

Listing 12: Fibonacci com Memoização

3. Programação Dinâmica com Backtracking Esta abordagem é útil quando a solução do problema envolve várias decisões, e você deseja explorar todas as possibilidades de maneira eficiente.

Exemplo: Problema da Mochila 0/1: No problema da mochila 0/1, você tem uma mochila com uma capacidade máxima e uma série de itens, cada um com um peso e um valor. O objetivo é maximizar o valor total dos itens que você pode colocar na mochila sem exceder a capacidade.

```
#include <iostream>
#include <vector>
#include <algorithm>

int knapsack(const std::vector<int>& weights, const std::vector<int>& values, int
capacity) {
    int n = weights.size();
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(capacity + 1, 0));

    for (int i = 1; i <= n; ++i) {
        for (int w = 0; w <= capacity; ++w) {
            if (weights[i - 1] <= w) {
                dp[i][w] = std::max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] +
values[i - 1]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }

    return dp[n][capacity];
}

int main() {
    std::vector<int> weights = {1, 2, 3, 8, 7, 4};
    std::vector<int> values = {20, 5, 10, 40, 15, 25};
    int capacity = 10;

    std::cout << "Valor m ximo na mochila: " << knapsack(weights, values,
capacity) << std::endl;
    return 0;
}
```

Listing 13: Fibonacci com Memoização

Sieve of Eratosthenes

Descrição: O Sieve of Eratosthenes é um algoritmo eficiente para encontrar todos os números primos menores ou iguais a um dado número n. Funciona marcando os múltiplos de cada número

primo, começando pelos números primos menores, como não primos, e continua até que todos os números menores ou iguais a n sejam verificados.

Quando usar: Use o Sieve of Eratosthenes quando precisar gerar uma lista de números primos em um intervalo de forma rápida e eficiente. É particularmente útil quando você precisa encontrar todos os números primos até um certo limite ou quando o problema exige a verificação rápida de primalidade para números em um intervalo.

Exemplo de uso: Encontrar todos os números primos menores que 1.000.000 para análises matemáticas, criptografia, ou problemas de teoria dos números.

```
#include <iostream>
#include <vector>
#include <cmath>

// Fun  o para gerar todos os n meros primos at  um dado limite usando o Sieve
 of Eratosthenes
std::vector<int> sieveOfEratosthenes(int limit) {
    std::vector<bool> isPrime(limit + 1, true);
    isPrime[0] = isPrime[1] = false; // 0 e 1 n o s o primos

    for (int p = 2; p * p <= limit; ++p) {
        if (isPrime[p]) {
            for (int multiple = p * p; multiple <= limit; multiple += p) {
                isPrime[multiple] = false;
            }
        }
    }

    std::vector<int> primes;
    for (int num = 2; num <= limit; ++num) {
        if (isPrime[num]) {
            primes.push_back(num);
        }
    }

    return primes;
}

int main() {
    int limit = 50; // Limite at  onde encontrar n meros primos

    std::vector<int> primes = sieveOfEratosthenes(limit);

    std::cout << "N meros primos at  " << limit << ": ";
    for (int prime : primes) {
        std::cout << prime << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 14: Sieve of Eratosthenes

Segmented Sieve

Descrição: O Segmented Sieve é uma extensão do Sieve of Eratosthenes que é usado para encontrartodos os números primos em um intervalo específico [L,R], onde L e R são os limites inferior e superior do intervalo. É especialmente útil quando o intervalo [L,R] é grande, mas RL é pequeno em comparação com R, permitindo encontrar primos em um intervalo sem a necessidade de calcular todos os primos até R.

Use o Segmented Sieve quando precisar encontrar números primos em um intervalo grande [L,R] onde L e R são grandes, mas a diferença RL é relativamente pequena. É eficiente para intervalos específicos e evita a necessidade de gerar todos os números primos até R diretamente.

```
#include <iostream>
#include <vector>
#include <cmath>

// Fun  o para gerar todos os n meros primos at  a raiz quadrada de um limite
usando o Sieve of Eratosthenes
```

```
std::vector<int> simpleSieve(int limit) {
    std::vector<bool> isPrime(limit + 1, true);
    isPrime[0] = isPrime[1] = false; // 0 e 1 n o  s  o primos

    for (int p = 2; p * p <= limit; ++p) {
        if (isPrime[p]) {
            for (int multiple = p * p; multiple <= limit; multiple += p) {
                isPrime[multiple] = false;
            }
        }
    }

    std::vector<int> primes;
    for (int num = 2; num <= limit; ++num) {
        if (isPrime[num]) {
            primes.push_back(num);
        }
    }

    return primes;
}

// Fun   o para encontrar todos os n meros primos no intervalo [L, R] usando o
Segmented Sieve
void segmentedSieve(int L, int R) {
    int limit = std::sqrt(R) + 1; // Raiz quadrada do limite superior
    std::vector<int> primes = simpleSieve(limit);

    // Inicializa o intervalo [L, R] como todos primos
    std::vector<bool> isPrime(R - L + 1, true);

    // Marca n o primos no intervalo [L, R]
    for (int prime : primes) {
        int start = std::max(prime * prime, (L + prime - 1) / prime * prime); //
Maior m ltiplo de 'prime' >= L
        for (int j = start; j <= R; j += prime) {
            isPrime[j - L] = false;
        }
    }

    // Imprime os n meros primos no intervalo [L, R]
    if (L == 1) isPrime[0] = false; // 1 n o    primo

    std::cout << "N meros primos no intervalo [" << L << ", " << R << "]: ";
    for (int i = 0; i <= R - L; ++i) {
        if (isPrime[i]) {
            std::cout << (L + i) << " ";
        }
    }
    std::cout << std::endl;
}

int main() {
    int L = 10; // Limite inferior do intervalo
    int R = 50; // Limite superior do intervalo

    segmentedSieve(L, R);

    return 0;
}
```

Listing 15: Segmented Sieve

2.4 BITWISE

Qualquer tipo de dado, seja um número inteiro, número racional ou um caractere, pode ser representado por bits. Exemplos de representação binária de números inteiros (sem sinal):

- **1)** $14 = \{1110\}_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 14$. **2)** $20 = \{10100\}_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 20$.

3 Operadores Bit-a-Bit

3.1 NOT ($\tilde{}$)

Bitwise NOT é um operador unário que flipa os bits de um número (se o bit é 0, vira 1 e vice-versa). Bitwise NOT é apenas o complemento de 1 de um número.

$$N = 5 = (101)_2$$
$$\sim N = \sim 5 = \sim (101)_2 = (010)_2 = 2$$

3.2 AND (&)

Bitwise AND é um operador binário que opera em duas palavras de bits de mesmo tamanho. Se ambos os bits na posição comparada das palavras forem 1, então o valor do bit resultante nessa posição na palavra final será 1; qualquer outro caso o bit resultante será 0.

$$A = 5 = (101)_2$$
$$B = 3 = (011)_2$$
$$A\&B = (101)_2\&(011)_2 = (001)_2 = 1$$

3.3 OR (—)

Bitwise OR, similarmente ao bitwise AND, é um operador binário que opera em duas palavras de bits de mesmo tamanho. Se ambos os bits na posição comparada forem 0, o valor do bit resultante nessa posição na palavra final será 0; qualquer outro caso o bit resultante será 1.

$$A = 5 = (101)_2$$
$$B = 3 = (011)_2$$
$$A|B = (101)_2|(011)_2 = (111)_2 = 7$$

3.4 XOR ($^{}$)

Bitwise XOR também é um operador binário que opera em duas palavras de bits de mesmo tamanho. Se ambos os bits na posição comparada forem iguais (0 ou 1), o valor do bit resultante nessa posição na palavra final será 0; se os bits forem diferentes (um 0 e outro 1), o bit resultante será 1.

$$A = 5 = (101)_2$$
$$B = 3 = (011)_2$$
$$A \oplus B = (101)_2 \oplus (011)_2 = (110)_2 = 6$$

3.5 Left Shift (¡¡)

O operador de deslocamento à esquerda desloca os bits de uma palavra X vezes para a esquerda e preenche a palavra com X zeros à direita. O deslocamento à esquerda de X bits em um número inteiro é equivalente a multiplicá-lo por 2^X.

$$1 << 1 = 2$$
$$1 << 2 = 4$$
$$2 << 2 = \{00010\}_2 << 2 = \{01000\}_2 = 8$$
$$1 << n = 2^n$$

3.6 Right Shift (¿¿)

O operador de deslocamento à direita desloca os bits de uma palavra X vezes para a direita e preenche a palavra com X zeros à esquerda. O deslocamento à direita de X bits em um número inteiro é

equivalente a dividi-lo por 2^X.

$$4 >> 1 = 2$$
$$6 >> 1 = 3$$
$$5 >> 1 = 2$$
$$16 >> 4 = 1$$

3.7 Tabela-Verdade

X	Y	X & Y	X $\|\|Y$	X $\oplus Y$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

4 Manipulando Bits

4.1 Checando se um determinado bit está ligado

Para checar se o i-ésimo bit de um número N está ligado, basta checar se o `AND` de 2^i e N é diferente de 0. Como visto anteriormente, o número 2^i é simplesmente o número 1 deslocado de i bits ($1 << i$).

- ```
  bool isSet(int bitPosition, int number) {
    bool ret = ((number & (1 << bitPosition)) != 0);
    return ret;
  }
  ```

4.2 Ligando um determinado bit em um número

Para ligar o i-ésimo bit de um número N, basta fazer o `OR` de 2^i com N.

```
int setBit(int bitPosition, int number) {
  return (number | (1 << bitPosition) );
}
```

5 Representando Conjunto com Bits - Bitmasks

Suponha que tenhamos um conjunto universo com 8 elementos, $U = \{a, b, c, d, e, f, g, h\}$. Vamos associar cada elemento de U a um bit:

- a $\rightarrow bit7b \rightarrow bit6$
- c $\rightarrow bit5d \rightarrow bit4$
- e $\rightarrow bit3f \rightarrow bit2$
- g $\rightarrow bit1h \rightarrow bit0$

Com essa associação, podemos representar qualquer subconjunto de U como uma máscara de 8 bits. Exemplos:

Conjunto	Bitmask
{b, c, f, h}	01100101
{a}	10000000
{}	00000000
```

5.1 Adicionar um elemento ao conjunto

Para adicionar um elemento a um conjunto representado como uma bitmask, basta setar o bit correspondente ao elemento na bitmask do conjunto.

- ```
  int addElement(int bitmask, int elementPosition) {
    bitmask = bitmask | (1 << elementPosition);
    return bitmask;
  }
  ```

5.2 Checar se um conjunto contém um elemento

Essa operação é a mesma de checar se um dado bit está setado na bitmask.

```
bool hasElement(int bitmask, int elementPosition){
  bool ret = ((bitmask & (1 << elementPosition)) != 0);
  return ret;
}
```

5.3 União de 2 Conjuntos

Um elemento estará presente na união de 2 conjuntos se e somente se pelo menos um dos conjuntos contiver este elemento. Com base nisso e na tabela-verdade, a máscara que representa a união de duas máscaras é o `OR` delas.

```
int union(int bitmaskA, int bitmaskB){
  return (bitmaskA | bitmaskB);
}
```

5.4 Interseção de 2 Conjuntos

Um elemento estará presente na interseção de 2 conjuntos se e somente se os 2 conjuntos contiverem este elemento. Logo, a máscara que representa a interseção de duas máscaras é o `AND` delas.

```
int intersection(int bitmaskA, int bitmaskB){
  return (bitmaskA & bitmaskB);
}
```

5.5 Gerando Todos os Subconjuntos de um Conjunto

Suponha que você tenha um conjunto $S = \{p, q, r\}$. Para formar um subconjunto de S, podemos escolher ou não o elemento p (2 opções), escolher ou não o elemento q (2 opções de novo) e escolher ou não o elemento r (2 opções novamente). Logo, podemos formar um subconjunto de S de $2 \times 2 \times 2$ maneiras diferentes. Um conjunto de N elementos possui 2^N subconjuntos. S possui $2^3 = 8$ subconjuntos.

Como S tem 3 elementos, precisamos de 3 bits para representar cada subconjunto. A associação é:

- p = bit 2
- q = bit 1
- r = bit 0

Os subconjuntos são representados por números de 3 bits:
```

Número	Subconjunto
0	{ }
1	{r}
2	{q}
3	{q, r}
4	{p}
5	{p, r}
6	{p, q}
7	{p, q, r}

Código para imprimir todos os subconjuntos de um conjunto:

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

void possibleSubsets(char S[], int N) {
    for(int i = 0; i < (1 << N); ++i) {  // i = [0, 2^N - 1]
        for(int j = 0; j < N; ++j)
            if(i & (1 << j))  // se o j-ésimo bit de i está setado, printamos S[j]
                cout << S[j] << ' ';
        cout << endl;
    }
}
```