Programação Competitiva

Henrique Souza

Augustu 27, 2024

Parte I
A Mentalidade do Solucionador de Problemas: Paradigmas e Padrões

1 Desconstruindo Problemas: Uma Abordagem Metódica

1.1 Lendo nas Entrelinhas: Restrições e Pistas Implícitas

O passo mais crítico na resolução de um problema é uma análise minuciosa do enunciado, especialmente das restrições sobre as variáveis de entrada.[8] As restrições não são apenas limites; são dicas poderosas sobre a complexidade de tempo exigida para a solução. Um juiz de programação competitiva típico pode executar cerca de 10^8 operações por segundo. Isso permite inferir a complexidade esperada a partir do tamanho máximo da entrada N[26]:

- $N \leq 20$: Sugere uma complexidade exponencial, como $O(2^N \cdot N)$ ou $O(N!)$. Isso aponta para busca completa, backtracking ou DP com bitmask.[27]
- $N \leq 100$: Pode permitir $O(N^3)$ ou $O(N^4)$, comum em problemas de Floyd-Warshall ou algumas formas de DP.[28]
- $N \leq 2000 - 5000$: Frequentemente aponta para uma solução $O(N^2)$, típica de DP básica, travessias de grafos em grafos densos ou geometria ingênua.[29]
- $N \leq 10^5$: Requer uma solução quase-linear, como $O(N \log N)$ ou $O(N)$. Este é um forte indicador para ordenação, busca binária, algoritmos de linha de varredura ou estruturas de dados avançadas como árvores de segmento.[27]
- $N > 10^6$: Exige uma solução linear $O(N)$ ou logarítmica $O(\log N)$.

Identificar casos de borda e "truques"é vital. Sempre teste sua lógica contra os valores mínimos/máximos, entradas vazias ou condições especiais mencionadas no problema (por exemplo, "os números são distintos").[8]

1.2 O Papel da Complexidade Assintótica na Seleção de Algoritmos

Compreender a notação Big O é inegociável.[17] É a linguagem usada para descrever a eficiência de um algoritmo e prever se ele passará no limite de tempo. O processo de escolha de um algoritmo é muitas vezes um processo de eliminação com base na complexidade. Se N é 10^5, qualquer abordagem $O(N^2)$ é imediatamente descartada, forçando o programador a pensar em termos de soluções $O(N \log N)$.[26] Isso imediatamente traz à mente algoritmos como ordenação, busca binária ou estruturas de dados como `std::set`, `std::map` ou Árvores de Segmento.

2 Paradigmas Algorítmicos Fundamentais

Esta seção detalha as estratégias de alto nível que formam a espinha dorsal da resolução de problemas algorítmicos. O processo de resolver um problema de programação competitiva é um processo de tomada de decisão hierárquico. O primeiro passo não é escolher um algoritmo específico (por exemplo, "Dijkstra"), mas classificar o tipo de problema com base em sua estrutura e no que ele pede (por exemplo, "caminho mais curto", "valor máximo", "contar todas as maneiras").[25] Essa classificação leva a um paradigma candidato. Dominar esses padrões é a chave para reconhecer a estrutura de um problema.[17]

2.1 Busca Completa (Força-Bruta)

Conceito Explorar sistematicamente cada solução ou estado possível para encontrar a solução correta ou ótima. Isso envolve técnicas como gerar todos os subconjuntos, permutações ou combinações.[27]

Lógica Embora muitas vezes seja lento demais para a solução final, uma abordagem de força-bruta é um ponto de partida crítico. Ajuda a entender a estrutura do problema e pode ser usada para testar uma solução mais otimizada.[23] Para restrições muito pequenas (por exemplo, $N < 20$), a busca completa pode ser a solução pretendida.

Problemas Típicos Problemas envolvendo pequenos conjuntos de itens onde todas as combinações devem ser verificadas.

2.2 Algoritmos Gulosos (Greedy)

Conceito A cada passo, fazer a escolha que parece melhor no momento (uma escolha "localmente ótima") na esperança de encontrar um ótimo global.[32]

Lógica O principal desafio é provar que a estratégia gulosa está correta. Isso geralmente envolve um "argumento de troca": assumindo que existe uma solução ótima melhor que não usa a escolha gulosa, e então mostrando que se pode trocar elementos para corresponder à escolha gulosa sem piorar a solução.

Problemas Típicos Problemas de agendamento (por exemplo, seleção de atividades), problema do troco com sistemas de moedas canônicos, encontrar Árvores Geradoras Mínimas (os algoritmos de Kruskal e Prim são fundamentalmente gulosos).[27]

2.3 Divisão e Conquista

Conceito Quebrar um problema em subproblemas menores e independentes do mesmo tipo, resolvê-los recursivamente e, em seguida, combinar suas soluções para resolver o problema original.[17]

Lógica A chave é que os subproblemas não se sobrepõem. A etapa de "combinar"é muitas vezes a parte mais complexa do algoritmo.

Problemas Típicos Merge Sort, Quick Sort, Busca Binária, Par de Pontos Mais Próximo.[36] A Busca Binária é um caso especial onde um subproblema é totalmente descartado.[38]

2.4 Programação Dinâmica (DP)

Conceito Uma técnica poderosa para problemas de otimização e contagem que decompõe um problema em subproblemas mais simples e sobrepostos. Ela resolve cada subproblema apenas uma vez e armazena sua solução, evitando computação redundante.[5]

Lógica Problemas de DP devem exibir duas propriedades:

- **Subestrutura Ótima:** A solução ótima para o problema principal pode ser construída a partir de soluções ótimas para seus subproblemas.
- **Subproblemas Sobrepostos:** A solução recursiva envolve resolver os mesmos subproblemas várias vezes.

A DP pode ser implementada de duas maneiras:

- **Top-Down (Memoização):** Uma implementação recursiva direta que armazena o resultado de cada subproblema em um cache (por exemplo, um array ou mapa) para evitar recomputação.[31]
- **Bottom-Up (Tabulação):** Uma abordagem iterativa que preenche uma tabela de DP, começando dos menores subproblemas e construindo até a solução final.[31]

Problemas Típicos Uma vasta categoria incluindo o Problema da Mochila, a Subsequência Comum Mais Longa e muitos problemas de busca de caminhos em grades.[27]

2.5 Técnicas Especializadas: Two Pointers, Sliding Window e Meet-in-the-Middle

Essas técnicas não são paradigmas fundamentais em si, mas sim otimizações poderosas do paradigma da Busca Completa. A estratégia de resolução de problemas frequentemente envolve primeiro identificar a solução de força-bruta e depois perguntar: "Qual propriedade da entrada (por exemplo, ordenada, contígua) posso explorar para podar essa busca ou evitar recomputação?". Essa linha de pensamento leva diretamente a essas técnicas de otimização especializadas.

Two Pointers Uma técnica eficiente para procurar pares ou subsequências em um array ordenado. Dois ponteiros começam em extremidades opostas (ou ambos no início) e se movem um em direção ao outro com base em condições, reduzindo uma busca $O(N^2)$ para $O(N)$.[33] É uma maneira mais inteligente de buscar, não uma abordagem fundamentalmente diferente de decomposição de problemas.

Sliding Window Usado para problemas em subarrays ou substrings contíguas. Uma "janela"de tamanho fixo ou variável desliza sobre os dados, e atualizamos eficientemente o estado da janela em vez de recalcular para cada nova posição. Isso também reduz frequentemente $O(N^2)$ para $O(N)$.[30] É uma otimização da verificação por força-bruta.

Meet-in-the-Middle Uma técnica que divide um problema em duas metades, resolve cada metade com força-bruta (busca completa) e depois combina os resultados. É eficaz quando o tamanho da entrada N é grande demais para $O(2^N)$ mas pequeno o suficiente para $O(2^{N/2})$, como $N \approx 40$.[27] A etapa de "combinar"geralmente envolve ordenar um conjunto de resultados e usar busca binária nele para cada elemento do outro conjunto. É uma otimização no estilo de divisão e conquista aplicada a uma busca por força-bruta.

Parte II

O Compêndio do Programador: Algoritmos e Estruturas de Dados

3 Dominando a Standard Template Library (STL) do C++

3.1 Contêineres Sequenciais e Associativos: Uma Análise Comparativa

A escolha da estrutura de dados correta é uma decisão estratégica fundamental. Uma escolha errada pode levar a uma solução que excede o limite de tempo (Time Limit Exceeded - TLE).

std::vector O contêiner mais fundamental. É um array dinâmico que fornece acesso aleatório $O(1)$, mas inserção/deleção $O(N)$ no meio. É a escolha padrão para a maioria das tarefas do tipo array devido ao seu desempenho e amizade com o cache.[9] A operação `push_back` tem complexidade $O(1)$ amortizada.

std::deque Uma fila de duas pontas (double-ended queue). Fornece inserção/deleção $O(1)$ tanto na frente quanto atrás, mas seus elementos não são armazenados de forma contígua, tornando o acesso aleatório um pouco mais lento que o vector.[48] É o contêiner subjacente padrão para `std::stack` e `std::queue`.[50]

std::set / std::map São contêineres associativos ordenados, tipicamente implementados como Árvores Rubro-Negras. Eles armazenam chaves únicas (set) ou pares chave-valor (map) e os mantêm ordenados. Operações como inserção, deleção e busca são $O(\log N)$.[11] São essenciais quando a ordem ou a localização do próximo/anterior elemento (`lower_bound`, `upper_bound`) é necessária.[52]

std::unordered_set / std::unordered_map São contêineres associativos baseados em hash (tabelas de hash). Eles fornecem complexidade de tempo média amortizada de $O(1)$ para inserção, deleção e busca, mas têm uma complexidade de pior caso de $O(N)$ em caso de muitas colisões de hash.[54] Eles não mantêm nenhuma ordem. Em programação competitiva, `unordered_map` é geralmente preferido em relação ao `map` para contagem de frequência e buscas quando a ordem não é necessária, devido ao seu desempenho superior no caso médio.[34] No entanto, é preciso ter cuidado com casos de teste projetados para causar colisões de hash, o que pode levar a um veredito de TLE. Usar uma função de hash personalizada pode mitigar esse risco. A escolha entre contêineres ordenados e não ordenados é, portanto, uma decisão estratégica que equilibra desempenho garantido com velocidade no caso médio.

Tabela 1: Comparação dos Principais Contêineres STL do C++

Contêiner	**Estrutura Interna**	**Acesso**	**Busca**	**Inserção/Deleção**	**Ordenado?**	**Caso de Uso Chave em PC**
`std::vector`	Array Dinâmico	$O(1)$	$O(N)$	$O(N)$ (meio), $O(1)$ amort. (fim)	Não	Array de uso geral, lista de adjacência.
`std::deque`	Lista de blocos de array	$O(1)$	$O(N)$	$O(1)$ (início/fim), $O(N)$ (meio)	Não	Fila de duas pontas, base para queue/stack.
`std::set`	Árvore Rubro-Negra	N/A	$O(\log N)$	$O(\log N)$	Sim	Manter elementos únicos e ordenados, `lower_bound`.
`std::map`	Árvore Rubro-Negra	$O(\log N)$	$O(\log N)$	$O(\log N)$	Sim	Mapeamento chave-valor ordenado.
`std::unordered_set`	Tabela de Hash	N/A	$O(1)$ avg.	$O(1)$ avg.	Não	Verificação de existência de elementos em $O(1)$.
`std::unordered_map`	Tabela de Hash	$O(1)$ avg.	$O(1)$ avg.	$O(1)$ avg.	Não	Contagem de frequência, mapeamento rápido.

3.2 Adaptadores e Filas de Prioridade: stack, queue, priority_queue

std::stack Um adaptador de contêiner Last-In, First-Out (LIFO), construído sobre `std::deque` por padrão.[50] Fornece operações `push()`, `pop()` e `top()`, todas em $O(1)$. Essencial para problemas que envolvem simulação de recursão, avaliação de expressões e pilhas monotônicas.[58]

std::queue Um adaptador de contêiner First-In, First-Out (FIFO), também construído sobre `std::deque` por padrão.[51] Fornece `push()`, `pop()` e `front()`. É a estrutura de dados central para a Busca em Largura (BFS) e algoritmos relacionados.[61]

std::priority_queue Um adaptador de contêiner que fornece acesso semelhante a uma fila, mas sempre retorna o elemento com a maior prioridade. Por padrão, é uma max-heap.[63] É crucial para algoritmos como Dijkstra, Prim e qualquer problema que exija a recuperação eficiente do elemento máximo/mínimo de uma coleção dinâmica.[66] Para criar uma min-heap, pode-se usar `priority_queue<int, vector<int>, greater<int»` ou o truque comum em PC de inserir valores negativos em uma max-heap.[63]

4 Algoritmos em Teoria dos Números

A teoria dos números é um tópico frequente em PC, especialmente em problemas de plataformas como Codeforces.[6] Os tópicos de primos, MDC e aritmética modular formam um kit de ferramentas interdependente. Dominar um geralmente requer conhecimento dos outros, e juntos eles desbloqueiam uma grande classe de problemas matemáticos. Por exemplo, a divisão modular requer o inverso multiplicativo modular, que por sua vez requer o Algoritmo de Euclides Estendido (baseado em MDC) ou o Pequeno Teorema de Fermat (que requer teste de primalidade).

4.1 Números Primos: Crivo de Eratóstenes e Teste de Primalidade

Crivo de Eratóstenes Um algoritmo eficiente para encontrar todos os números primos até um determinado inteiro n. Ele funciona marcando iterativamente como compostos os múltiplos de cada primo, começando com 2.[69] A implementação padrão tem uma complexidade de tempo de $O(n \log \log n)$. É usado para pré-computação quando muitas consultas relacionadas a primos são necessárias dentro de um certo intervalo.[72]

```cpp
void sieve(int n, vector<bool>& is_prime) {
    is_prime.assign(n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int p = 2; p * p <= n; p++) {
        if (is_prime[p]) {
            for (int i = p * p; i <= n; i += p)
                is_prime[i] = false;
        }
    }
}
```

Listing 1: Crivo de Eratóstenes (Adaptado de [69])

Teste de Primalidade Para testar se um único número n, potencialmente grande, é primo.

- **Divisão por Tentativa:** O método mais simples é verificar divisores até $\sqrt{n}$. Complexidade: $O(\sqrt{n})$.[73]
- **Teste de Miller-Rabin:** Um teste probabilístico que é extremamente rápido e confiável para números grandes. É baseado em propriedades derivadas do Pequeno Teorema de Fermat. Para um inteiro de 64 bits, o uso de um conjunto específico de 7 a 12 bases pré-determinadas torna o teste determinístico e completamente preciso.[73] Este é o padrão para testes de primalidade de números grandes em PC.

4.2 O Algoritmo de Euclides: MDC, MMC e a Forma Estendida

Máximo Divisor Comum (MDC) O algoritmo de Euclides é o método padrão e eficiente para encontrar o MDC de dois inteiros a e b.[76] É baseado no princípio de que $\gcd(a, b) = \gcd(b, a \pmod b)$. A biblioteca padrão do C++17 inclui `std::gcd`.[79]

```cpp
long long gcd(long long a, long long b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

Listing 2: MDC Recursivo (Adaptado de [76])

Mínimo Múltiplo Comum (MMC) Facilmente calculado usando o MDC: $\operatorname{lcm}(a, b) = (a \cdot b)/\gcd(a, b)$.

Algoritmo de Euclides Estendido Encontra inteiros x e y tais que $a \cdot x + b \cdot y = \gcd(a, b)$.[80] Isso é crucial para calcular inversos multiplicativos modulares.[76]

```
long long extendedGcd(long long a, long long b, long long &x, long long &y) {
    if (a == 0) {
        x = 0;
        y = 1;
        return b;
    }
    long long x1, y1;
    long long d = extendedGcd(b % a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

Listing 3: Euclides Estendido (Adaptado de [76])

4.3 Aritmética Modular: Exponenciação, Inverso e o Teorema Chinês do Resto

Exponenciação Modular (Exponenciação Binária) Um algoritmo para calcular $(a^b) \pmod m$ em tempo $O(\log b)$, essencial para cálculos com grandes expoentes que, de outra forma, causariam overflow em tipos de dados padrão.[84] Ele funciona elevando a base ao quadrado repetidamente e multiplicando no resultado com base na representação binária do expoente.

```
long long power(long long base, long long exp, long long mod) {
    long long res = 1;
    base %= mod;
    while (exp > 0) {
        if (exp % 2 == 1) res = (res * base) % mod;
        base = (base * base) % mod;
        exp /= 2;
    }
    return res;
}
```

Listing 4: Exponenciação Modular (Adaptado de [85])

Inverso Multiplicativo Modular Dado a e m, encontrar um inteiro x tal que $(a \cdot x) \pmod m = 1$. O inverso existe se e somente se a e m são coprimos ($\gcd(a, m) = 1$).[88] É o equivalente da divisão em aritmética modular. Pode ser calculado de duas maneiras principais:

- **Usando o Algoritmo de Euclides Estendido:** Se $\gcd(a, m) = 1$, então $a \cdot x + m \cdot y = 1$. Tomando o módulo m, obtemos $a \cdot x \equiv 1 \pmod m$. O x encontrado é o inverso modular.[88]
- **Usando o Pequeno Teorema de Fermat:** Se m é um número primo, então $a^{m-1} \equiv 1 \pmod m$. Isso implica que $a \cdot a^{m-2} \equiv 1 \pmod m$, então a^{m-2} é o inverso. Isso pode ser calculado eficientemente com exponenciação modular.[84]

Teorema Chinês do Resto (TCR) Resolve um sistema de congruências simultâneas. Dados módulos coprimos dois a dois $n_1, n_2, \ldots, n_k$ e os restos correspondentes $a_1, a_2, \ldots, a_k$, ele encontra uma solução única x (módulo $N = n_1 \cdot n_2 \cdot \cdots \cdot n_k$) tal que $x \equiv a_i \pmod{n_i}$ para todo i.[90] É uma ferramenta poderosa em problemas que envolvem a decomposição de um grande cálculo modular em cálculos menores e gerenciáveis.[93]

5 Algoritmos de Grafos

Grafos são um dos tópicos mais importantes e frequentes na programação competitiva.[5] Muitos algoritmos de grafos "avançados"são, na verdade, aplicações inteligentes ou combinações das duas travessias fundamentais: BFS e DFS. Compreender as propriedades centrais dessas travessias é a chave para entender os algoritmos mais complexos.

5.1 Representação e Travessia de Grafos

Representação A maneira mais comum de representar um grafo em PC é usando uma lista de adjacência (`vector<vector<int» adj`), onde `adj[i]` armazena uma lista de vértices conectados ao vértice i.[69] Para grafos densos, uma matriz de adjacência pode ser usada, mas é menos comum devido à sua complexidade de espaço $O(V^2)$.

Busca em Largura (BFS) Um algoritmo de travessia que explora os vértices nível por nível. Ele usa uma `queue` para gerenciar os nós a serem visitados.[62] A BFS garante encontrar o caminho mais curto em termos de número de arestas em um grafo não ponderado.[95] A propriedade de "camada"da BFS é a base do algoritmo de Dijkstra, que é essencialmente uma BFS em um grafo ponderado onde a fila é priorizada pela distância.

```cpp
vector<int> bfs(int start_node, int n, const vector<vector<int>>& adj) {
    queue<int> q;
    vector<bool> visited(n + 1, false);
    vector<int> dist(n + 1, -1);
    vector<int> path;

    q.push(start_node);
    visited[start_node] = true;
    dist[start_node] = 0;

    while (!q.empty()) {
        int v = q.front();
        q.pop();
        path.push_back(v);

        for (int u : adj[v]) {
            if (!visited[u]) {
                visited[u] = true;
                q.push(u);
                dist[u] = dist[v] + 1;
            }
        }
    }
    return path; // Retorna a ordem de travessia
}
```

Listing 5: BFS (Adaptado de [95])

Busca em Profundidade (DFS) Um algoritmo de travessia que explora o mais longe possível ao longo de cada ramo antes de retroceder. É naturalmente implementado recursivamente ou iterativamente com uma `stack`.[94] A DFS é um bloco de construção fundamental para muitos outros algoritmos de grafos, como ordenação topológica e encontrar componentes fortemente conectados.

```cpp
vector<bool> visited;
vector<vector<int>> adj;

void dfs(int u) {
    visited[u] = true;
    // Process node u
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs(v);
        }
    }
}

void traverse_graph(int n) {
    visited.assign(n + 1, false);
    for (int i = 1; i <= n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
```

```
}
```

Listing 6: DFS (Adaptado de [94])

5.2 Problemas de Caminho Mais Curto

A escolha do algoritmo depende inteiramente das propriedades do grafo (pesos das arestas, presença de ciclos). Um erro na escolha pode levar a uma resposta errada ou a um TLE.

Tabela 2: Guia de Seleção de Algoritmo de Caminho Mais Curto

Algoritmo	**Complexidade**	**Pesos Negativos?**	**Ciclos Negativos?**	**Caso de Uso Chave**
BFS	$O(V+E)$	Não	Não	Grafos não ponderados.
Dijkstra	$O(E \log V)$	Não	Não	Grafos com pesos não-negativos.
Bellman-Ford	$O(V \cdot E)$	Sim	Detecta	Grafos com pesos negativos.
Floyd-Warshall	$O(V^3)$	Sim	Detecta	Todos os pares de caminhos mais cur

Algoritmo de Dijkstra Encontra os caminhos mais curtos de uma única fonte em um grafo ponderado com pesos de aresta não-negativos. A abordagem gulosa de sempre visitar o nó não visitado mais próximo é implementada eficientemente usando uma `std::priority_queue` (min-heap).[99]

Algoritmo de Bellman-Ford Encontra os caminhos mais curtos de uma única fonte em um grafo ponderado que pode conter arestas de peso negativo. É mais lento que o de Dijkstra ($O(V \cdot E)$), mas mais versátil. Sua ideia central é relaxar todas as arestas $V-1$ vezes. Um V-ésimo relaxamento que ainda melhora um caminho indica a presença de um ciclo de peso negativo.[102]

Algoritmo de Floyd-Warshall Um algoritmo de caminhos mais curtos para todos os pares. Ele encontra os caminhos mais curtos entre cada par de vértices em $O(V^3)$. É uma abordagem baseada em DP que considera iterativamente cada vértice k como um ponto intermediário nos caminhos entre i e j.[28]

5.3 Árvores Geradoras Mínimas (MST)

Uma MST de um grafo conectado, não direcionado e ponderado é um subgrafo que conecta todos os vértices com o menor peso total de arestas possível.[107]

Algoritmo de Kruskal Um algoritmo guloso que ordena todas as arestas por peso e as adiciona à MST se não formarem um ciclo. Ele usa uma estrutura de dados Disjoint Set Union (DSU) para verificar eficientemente a existência de ciclos.[107] Complexidade: $O(E \log E)$ para ordenação.

Algoritmo de Prim Um algoritmo guloso que "cresce"a MST a partir de um vértice de partida arbitrário. Ele adiciona repetidamente a aresta mais barata que conecta um vértice na MST a um vértice fora da MST. É tipicamente implementado com uma `priority_queue`.[110] Complexidade: $O(E \log V)$.

5.4 Fluxos e Conectividade

Ordenação Topológica Uma ordenação linear de vértices em um Grafo Acíclico Dirigido (DAG) tal que para cada aresta dirigida $u \to v$, u vem antes de v na ordenação.[113] É essencial para problemas de agendamento e como um precursor para DP em DAGs. Pode ser implementada com o Algoritmo de Kahn (uma abordagem baseada em BFS usando graus de entrada) ou com DFS (uma travessia em pós-ordem reversa).[114]

Componentes Fortemente Conectados (SCCs) Em um grafo dirigido, um SCC é um subgrafo maximal onde cada vértice é alcançável de todos os outros vértices dentro desse subgrafo.[117] Encontrar SCCs permite simplificar um grafo em um DAG de seus componentes. Os algoritmos padrão são o Algoritmo de Kosaraju (duas passadas de DFS, uma no grafo reverso) e o Algoritmo de Tarjan (uma única passada de DFS, mais complexa).[117] O algoritmo de Kosaraju é uma aplicação brilhante da propriedade dos tempos de finalização da DFS para identificar corretamente os SCCs.

6 Processamento e Manipulação de Strings

Problemas de string são comuns e possuem um conjunto dedicado de algoritmos poderosos.[122] A escolha do algoritmo de string depende do tipo de consulta. Para comparações de substrings arbitrárias, o hashing é ideal. Para procurar um padrão fixo, KMP é a escolha. Para consultas baseadas em prefixos, uma Trie é a melhor solução.

6.1 Hashing de String para Comparação Eficiente

Conceito Uma técnica para converter uma string em um único inteiro (seu "hash") para que as comparações possam ser feitas em tempo $O(1)$ em vez de $O(L)$, onde L é o comprimento da string. O método mais comum é o Polynomial Rolling Hash.[124]

Lógica Uma string s de comprimento L é tratada como um número na base p, onde p é um primo maior que o tamanho do alfabeto. O hash é $(\sum_{i=0}^{L-1} s[i] \cdot p^i) \pmod{m}$. Para evitar colisões, um módulo primo grande m é usado. O uso de dois pares diferentes de (p, m) (hashing duplo) torna as colisões virtualmente impossíveis em um contexto de competição.[125]

Aplicação Encontrar eficientemente substrings duplicadas, comparar substrings e como um bloco de construção em outros algoritmos. Com hashes de prefixo pré-calculados, o hash de qualquer substring pode ser encontrado em $O(1)$.

6.2 O Algoritmo Knuth-Morris-Pratt (KMP)

Conceito Um algoritmo de $O(N + M)$ para encontrar todas as ocorrências de um padrão P de comprimento M em um texto T de comprimento N.[126]

Lógica Ele evita comparações redundantes pré-computando um array "Longest Proper Prefix which is also Suffix"(LPS) para o padrão. Este array `lps` informa ao algoritmo quantos caracteres deslocar o padrão em caso de uma incompatibilidade, usando o conhecimento adquirido do prefixo já correspondido.[127]

```
void computeLPS(const string& pat, vector<int>& lps) {
    int m = pat.length();
    lps.assign(m, 0);
    int length = 0; // Comprimento do lps anterior
    int i = 1;
    while (i < m) {
        if (pat[i] == pat[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

Listing 7: Construção do array LPS (Adaptado de [126])

6.3 A Estrutura de Dados Trie (Árvore de Prefixos)

Conceito Uma estrutura de dados em forma de árvore que armazena um conjunto de strings de forma eficiente, onde os caminhos da raiz até um nó representam prefixos.[130] Cada nó normalmente contém um array de ponteiros (um para cada caractere no alfabeto) e uma flag booleana indicando se é o fim de uma palavra.

Lógica Para inserir ou procurar uma string, percorre-se a trie a partir da raiz de acordo com os caracteres da string. Isso permite operações em $O(L)$, onde L é o comprimento da string, independentemente do número de strings na trie.[130]

Aplicação Autocompletar, buscas em dicionários e problemas envolvendo correspondência de prefixos. É a base para algoritmos mais avançados como Aho-Corasick.[132]

```
const int ALPHABET_SIZE = 26;
struct TrieNode {
    TrieNode *children[ALPHABET_SIZE];
    bool isEndOfWord;

    TrieNode() {
        isEndOfWord = false;
        for (int i = 0; i < ALPHABET_SIZE; i++)
            children[i] = nullptr;
    }
};

void insert(TrieNode *root, const string& key) {
    TrieNode *pCrawl = root;
    for (char c : key) {
        int index = c - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = new TrieNode();
        pCrawl = pCrawl->children[index];
    }
    pCrawl->isEndOfWord = true;
}
```

Listing 8: Nó da Trie e Inserção (Adaptado de [130])

7 Estruturas de Dados Avançadas para Consultas de Intervalo

Essas estruturas de dados são essenciais para problemas em competições de nível mais alto, geralmente envolvendo consultas em segmentos de array.[27] Elas representam um salto conceitual de simplesmente armazenar dados para armazenar informações pré-computadas sobre intervalos de uma maneira que permite atualizações e consultas eficientes.

7.1 Árvore de Segmentos: A Ferramenta Definitiva para Consultas de Intervalo

Conceito Uma estrutura de dados em árvore binária usada para armazenar informações sobre intervalos ou segmentos. Cada nó representa um intervalo, e o valor no nó é um agregado desse intervalo (por exemplo, soma, mínimo, máximo).[135]

Lógica Segue uma abordagem de divisão e conquista. A raiz representa todo o array $[0, N-1]$. Um nó para o intervalo $[L, R]$ tem filhos para $[L, M]$ e $[M+1, R]$, onde $M = (L+R)/2$. Essa estrutura permite tanto consultas de intervalo quanto atualizações de ponto em tempo $O(\log N)$.

Propagação Lenta (Lazy Propagation) Uma otimização crucial para atualizações de intervalo. Em vez de atualizar todos os elementos em um intervalo (o que seria lento), as atualizações são "preguiçosamente"armazenadas em nós pais e propagadas para os filhos apenas quando necessário. Isso mantém a complexidade das atualizações de intervalo em $O(\log N)$.[139]

```cpp
vector<int> arr;
vector<int> tree;

void build(int node, int start, int end) {
    if (start == end) {
        tree[node] = arr[start];
        return;
    }
    int mid = start + (end - start) / 2;
    build(2 * node + 1, start, mid);
    build(2 * node + 2, mid + 1, end);
    tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}

int query(int node, int start, int end, int l, int r) {
    if (r < start || end < l) {
        return 0; // Fora do intervalo
    }
    if (l <= start && end <= r) {
        return tree[node]; // Totalmente dentro do intervalo
    }
    int mid = start + (end - start) / 2;
    int p1 = query(2 * node + 1, start, mid, l, r);
    int p2 = query(2 * node + 2, mid + 1, end, l, r);
    return p1 + p2;
}
```

Listing 9: Construção e Consulta para Soma de Árvore de Segmentos (Adaptado de [135])

7.2 Árvore de Fenwick (Binary Indexed Tree - BIT)

Conceito Uma estrutura de dados que pode atualizar eficientemente elementos e calcular somas de prefixo em uma tabela de números.[141]

Lógica É mais eficiente em termos de espaço ($O(N)$) e muitas vezes mais simples de codificar do que uma árvore de segmentos. Cada índice i na BIT armazena a soma de um intervalo específico de elementos do array original, determinado pela representação binária de i (especificamente, o bit menos significativo). Uma consulta para uma soma de prefixo $[0, r]$ ou uma atualização em um ponto i pode ser feita em tempo $O(\log N)$. A soma do intervalo $[l, r]$ é calculada como $\text{soma}(r) - \text{soma}(l-1)$.[143]

Comparação Embora uma Árvore de Segmentos possa lidar com uma variedade maior de consultas de intervalo (como mínimo/máximo de intervalo), uma Árvore de Fenwick é frequentemente mais rápida e suficiente para consultas de soma de intervalo.[144]

```cpp
vector<int> bit;
int n;

void update(int idx, int delta) {
    for (; idx <= n; idx += idx & -idx)
        bit[idx] += delta;
}

int query(int idx) {
    int sum = 0;
    for (; idx > 0; idx -= idx & -idx)
        sum += bit[idx];
    return sum;
}
```

Listing 10: BIT - Atualização e Consulta (Adaptado de [141])

7.3 Disjoint Set Union (DSU) / Union-Find

Conceito Uma estrutura de dados que rastreia uma partição de um conjunto em subconjuntos disjuntos.[145] Possui duas operações primárias:

- **find:** Determina a qual subconjunto um elemento pertence (encontrando o representante do conjunto).
- **union:** Mescla dois subconjuntos em um único subconjunto.

Lógica É tipicamente implementada como uma floresta, onde cada conjunto é uma árvore e a raiz é o representante. Duas otimizações cruciais, compressão de caminho (achatando a árvore durante as operações `find`) e união por tamanho/rank (anexando a árvore menor à raiz da árvore maior), tornam suas operações quase constantes em média (a complexidade amortizada é $O(\alpha(N))$, onde α é a função inversa de Ackermann, que cresce extremamente devagar).[145]

Aplicação Algoritmo de Kruskal para MST, encontrar componentes conectados em um grafo e qualquer problema envolvendo particionamento ou classes de equivalência.[146]

```cpp
struct DSU {
    vector<int> parent;
    vector<int> sz;
    DSU(int n) {
        parent.resize(n);
        iota(parent.begin(), parent.end(), 0);
        sz.assign(n, 1);
    }

    int find_set(int v) {
        if (v == parent[v])
            return v;
        return parent[v] = find_set(parent[v]); // Compressao de caminho
    }

    void union_sets(int a, int b) {
        a = find_set(a);
        b = find_set(b);
        if (a != b) {
            if (sz[a] < sz[b]) // Uniao por tamanho
                swap(a, b);
            parent[b] = a;
            sz[a] += sz[b];
        }
    }
};
```

Listing 11: DSU com otimizações (Adaptado de [145])

8 Um Mergulho Profundo nos Padrões de Programação Dinâmica

DP é uma área vasta e crítica em PC.[5] Reconhecer o padrão subjacente é a chave para resolver problemas de DP.[31] A dificuldade da DP reside em modelar o estado do problema; as restrições (N pequeno, entrada é uma árvore, etc.) são as principais pistas para estruturar esse modelo.

8.1 Problemas da Mochila (Knapsack)

Conceito Uma classe de problemas que envolve a seleção de itens com pesos e valores dados para maximizar o valor total dentro de uma capacidade limitada.[157]

- **Mochila 0/1:** Cada item pode ser pego uma vez ou não. O estado é `dp[i][w]`: valor máximo usando um subconjunto dos primeiros i itens com capacidade w. Recorrência: $\texttt{dp[i][w]} = \max(\texttt{dp[i-1][w]}, \texttt{valor[i]} + \texttt{dp[i-1][w-peso[i]]})$.[29]
- **Mochila Ilimitada:** Cada item pode ser pego um número ilimitado de vezes. Recorrência: $\texttt{dp[i][w]} = \max(\texttt{dp[i-1][w]}, \texttt{valor[i]} + \texttt{dp[i][w-peso[i]]})$.[151]
- **Mochila Fracionária:** Os itens podem ser quebrados. Este é um caso especial que pode ser resolvido de forma gulosa, pegando primeiro os itens com a maior razão valor/peso.[158]

Tabela 3: Identificação de Padrões de Programação Dinâmica

Padrão	**Pistas para a Definição do Estado**	**Recorrência Típica**	**Exemplos de Problemas**
Mochila (Knapsack)	Escolher um subconjunto de itens com pesos/valores para caber em uma capacidade.	`dp[i][w]` = valor máximo usando os primeiros i itens com capacidade w.	0/1 Knapsack, Unbounded Knapsack.[29]
Sequência (LIS/LCS)	Encontrar a subsequência mais longa/ótima em uma ou duas sequências.	`dp[i]` ou `dp[i][j]`.	Longest Increasing Subsequence, Longest Common Subsequence.[152]
DP em Grades	Problemas de caminho em uma matriz 2D, movimentos restritos (apenas para baixo/-direita).	`dp[i][j]` = resposta para a subgrade terminando em (i, j).	Caminhos Únicos, Soma Mínima do Caminho.[27]
DP com Bitmask	N é pequeno (≤ 20), precisa rastrear um subconjunto de itens/nós usados.	`dp[mask][i]` = resposta para o subconjunto `mask` terminando no item i.	Problema do Caixeiro Viajante (TSP), problemas de atribuição.[154]
DP em Árvores	A entrada é uma árvore; a resposta para um nó depende das respostas de seus filhos.	`dp[u][state]` = resposta para a subárvore de u em um determinado estado.	Conjunto Independente Máximo, Diâmetro da Árvore.[155]

8.2 Problemas de Sequência (LIS, LCS)

Subsequência Crescente Mais Longa (LIS) Encontrar o comprimento da subsequência mais longa de uma dada sequência tal que todos os elementos da subsequência estejam em ordem estritamente crescente.[153]

- **Solução DP $O(N^2)$:** `dp[i]` = comprimento da LIS terminando no índice i. `dp[i]` = 1 + max(`dp[j]`) para todo $j < i$ onde `a[j]` < `a[i]`.[153]
- **Solução $O(N \log N)$:** Uma abordagem mais inteligente que mantém o menor elemento final para todos os comprimentos possíveis de LIS, usando busca binária para encontrar a posição correta para o elemento atual.[160]

Subsequência Comum Mais Longa (LCS) Dadas duas sequências, encontrar o comprimento da subsequência mais longa presente em ambas.[152]

- **Solução DP $O(N \cdot M)$:** `dp[i][j]` = comprimento da LCS de `s1[0..i-1]` e `s2[0..j-1]`. A recorrência depende se `s1[i-1] == s2[j-1]`.[152]

8.3 Compressão de Estado com DP com Bitmask

Conceito Usado quando N é pequeno (tipicamente $N \leq 20$) e precisamos manter o controle de um subconjunto de itens/nós que foram usados ou visitados.[154]

Lógica Uma bitmask (um inteiro) é usada para representar o conjunto. O i-ésimo bit é 1 se o i-ésimo elemento está no conjunto, e 0 caso contrário. O estado da DP geralmente se parece com `dp[mask][i]`, representando um valor (por exemplo, caminho mais curto) tendo visitado o subconjunto de nós na `mask` e terminando no nó i.[154]

Aplicação Problema do Caixeiro Viajante (TSP) com N pequeno, contagem de pareamentos, problemas de atribuição.[167] Iterar por todas as submáscaras de uma máscara é uma operação comum neste padrão.[169]

8.4 Programação Dinâmica em Árvores

Conceito Problemas de DP onde a entrada é uma árvore. Os subproblemas são definidos nas subárvores dos nós.[155]

Lógica Uma travessia DFS é tipicamente usada. O estado da DP `dp[u]` calcula um valor para a subárvore enraizada em u, com base nos valores de DP já computados de seus filhos. O estado pode ter múltiplas dimensões, por exemplo, `dp[u][0]` para quando o nó u não está incluído no conjunto da solução, e `dp[u][1]` para quando está.[155]

Aplicação Conjunto independente máximo em uma árvore, encontrar o diâmetro da árvore e muitos outros problemas de caminho/coloração em árvores.[156] O **Re-enraizamento** é uma técnica avançada usada para resolver problemas que pedem um valor para cada nó como raiz, atualizando eficientemente os valores de DP sem reexecutar toda a DFS de cada nó.[155]

9 Uma Introdução à Geometria Computacional

Problemas de geometria são frequentemente temidos em PC por sua dependência de precisão de ponto flutuante e numerosos casos de borda.[26] O sucesso na implementação de algoritmos de geometria depende menos de teoremas geométricos complexos e mais no tratamento robusto de erros de precisão e casos degenerados.

9.1 Conceitos Fundamentais: Pontos, Linhas e Orientações

Representação Pontos são tipicamente representados como uma `struct` ou `pair` de `double` ou `long long`. Usar `complex<double>` da biblioteca C++ pode simplificar muitas operações como rotação e álgebra vetorial.[27]

Produto Vetorial Uma ferramenta fundamental para determinar a orientação de três pontos ordenados (p, q, r). O sinal de $(q.x - p.x)(r.y - p.y) - (q.y - p.y)(r.x - p.x)$ informa se a curva de pq para qr é no sentido horário, anti-horário ou se são colineares. Isso é essencial para algoritmos de fecho convexo e interseção de linhas.[172] É preferível usar aritmética inteira sempre que possível para evitar problemas de precisão com ponto flutuante.

9.2 Construindo o Fecho Convexo

Conceito O menor polígono convexo que envolve um conjunto de pontos.[172]

Algoritmos

- **Graham Scan:** Ordena os pontos por ângulo polar em torno de um pivô (o ponto mais baixo em y) e depois usa uma pilha para construir o fecho, adicionando iterativamente pontos e removendo pontos anteriores que criariam uma curva não anti-horária.[172] Complexidade: $O(N \log N)$ devido à ordenação.
- **Monotone Chain (Algoritmo de Andrew):** Ordena os pontos pela coordenada x e depois constrói os fechos superior e inferior do conjunto de pontos separadamente em tempo $O(N)$, para um total de $O(N \log N)$.[175] Muitas vezes é mais simples de implementar corretamente do que o Graham Scan.
- **Jarvis March (Gift Wrapping):** Um algoritmo $O(N \cdot H)$ (onde H é o número de pontos no fecho) que "embrulha"os pontos encontrando repetidamente o próximo ponto mais anti-horário.[173]

9.3 O Algoritmo de Linha de Varredura (Sweep-Line)

Conceito Um paradigma algorítmico para resolver problemas geométricos. Uma "linha de varredura"vertical conceitual se move através do plano, processando objetos geométricos (por exemplo, pontos de extremidade de segmentos de linha) à medida que os encontra.[177]

Lógica O algoritmo mantém uma estrutura de dados (o "status") dos objetos atualmente intersectados pela linha de varredura, ordenados por sua coordenada y. Eventos (as coordenadas x onde o status muda) são armazenados em uma fila de prioridade ou uma lista ordenada. Ao verificar apenas as interações entre objetos adjacentes na estrutura de status, ele reduz o número de pares a serem verificados.

Aplicação Encontrar interseções de segmentos de linha, encontrar o par de pontos mais próximo, calcular a área da união de retângulos.[177] A complexidade é tipicamente $O(N \log N)$.

Parte III
Guia Prático de Algoritmos para a INTERIF

10 Algoritmos Fundamentais de Ordenação e Busca

10.1 QuickSort

Um algoritmo de ordenação eficiente, baseado na estratégia de "dividir para conquistar". Ele seleciona um elemento como pivô e particiona os outros elementos em dois sub-arrays, de acordo com se são menores ou maiores que o pivô.

Explicação do Funcionamento O algoritmo escolhe um pivô e reorganiza o array para que todos os elementos menores que o pivô venham antes dele, e todos os elementos maiores venham depois. Este processo de particionamento é aplicado recursivamente aos sub-arrays. Sua complexidade média é $O(n \log n)$, mas o pior caso é $O(n^2)$, embora seja raro na prática.

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

using namespace std;

template <typename T>
int partition(vector<T>& arr, int low, int high) {
    T pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; ++j) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

template <typename T>
void quickSort(vector<T>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    vector<int> data = {10, 7, 8, 9, 1, 5};
    quickSort(data, 0, data.size() - 1);

    cout << "Array ordenado com QuickSort: ";
    for (const auto& elem : data) {
        cout << elem << " ";
    }
```

```
    cout << endl;

    return 0;
}
```

Listing 12: Implementação do QuickSort.

10.2 MergeSort

Outro algoritmo de ordenação do tipo "dividir para conquistar"que garante uma complexidade de tempo de $O(n \log n)$ em todos os casos. É também um algoritmo de ordenação estável.

Explicação do Funcionamento O MergeSort divide o array pela metade repetidamente até que cada sub-array contenha apenas um elemento. Em seguida, ele mescla (merge) os sub-arrays de volta, ordenando-os no processo. Sua desvantagem é a necessidade de espaço extra ($O(n)$) para o processo de mesclagem.

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

using namespace std;

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

    vector<T> leftArr(n1), rightArr(n2);

    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;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k++] = leftArr[i++];
        } else {
            arr[k++] = rightArr[j++];
        }
    }

    while (i < n1) arr[k++] = leftArr[i++];
    while (j < n2) arr[k++] = rightArr[j++];
}

template <typename T>
void mergeSort(vector<T>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
    vector<int> data = {12, 11, 13, 5, 6, 7};
    mergeSort(data, 0, data.size() - 1);

    cout << "Array ordenado com MergeSort: ";
    for (const auto& elem : data) {
        cout << elem << " ";
    }
    cout << endl;

    return 0;
}
```

Listing 13: Implementação do MergeSort.

10.3 Counting Sort

Um algoritmo de ordenação não-comparativo que é extremamente rápido para dados com um intervalo de valores pequeno e conhecido.

Explicação do Funcionamento O Counting Sort funciona contando o número de ocorrências de cada elemento no array de entrada. Essa contagem é armazenada em um array auxiliar. Em seguida, o array de contagem é usado para determinar as posições de cada elemento no array de saída, resultando em uma ordenação estável com complexidade de tempo linear, $O(n + k)$, onde k é o intervalo dos valores de entrada.

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

using namespace std;

void countingSort(vector<int>& arr) {
    if (arr.empty()) return;

    int maxVal = *max_element(arr.begin(), arr.end());
    vector<int> count(maxVal + 1, 0);

    for (int num : arr) {
        count[num]++;
    }

    int index = 0;
    for (int i = 0; i <= maxVal; ++i) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}

int main() {
    vector<int> data = {4, 2, 2, 8, 3, 3, 1};
    countingSort(data);

    cout << "Array ordenado com Counting Sort: ";
    for (const auto& elem : data) {
        cout << elem << " ";
    }
    cout << endl;

    return 0;
}
```

Listing 14: Implementação do Counting Sort para inteiros positivos.

11 Algoritmos em Grafos

11.1 Contagem de Componentes Conectados (usando DFS)

Este algoritmo é útil para problemas que pedem para agrupar elementos com base em relações de conectividade, como formar equipes ou contar ilhas.

Explicação do Funcionamento O código representa o grafo usando uma lista de adjacência. Ele percorre cada vértice (nó) do grafo. Se um vértice ainda não foi visitado, significa que ele pertence a um novo componente conectado. O contador de componentes é incrementado, e uma Busca em Profundidade (DFS) é iniciada a partir desse vértice. A DFS explora todos os vértices alcançáveis, marcando-os como visitados para que não sejam contados novamente.

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

```cpp
#include <numeric>

using namespace std;

// Funcao de Busca em Profundidade (DFS)
void dfs(int u, const vector<vector<int>>& adj, vector<bool>& visited) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs(v, adj, visited);
        }
    }
}

int contarComponentes(int n, const vector<vector<int>>& adj) {
    vector<bool> visited(n + 1, false);
    int componentes = 0;

    for (int i = 1; i <= n; ++i) {
        if (!visited[i]) {
            dfs(i, adj, visited);
            componentes++;
        }
    }
    return componentes;
}

int main() {
    int n_nodes = 7;
    vector<vector<int>> adj(n_nodes + 1);
    vector<pair<int, int>> edges = {{1, 2}, {2, 3}, {4, 5}};

    for(const auto& edge : edges) {
        adj[edge.first].push_back(edge.second);
        adj[edge.second].push_back(edge.first);
    }

    int num_componentes = contarComponentes(n_nodes, adj);
    // Componentes: {1,2,3}, {4,5}, {6}, {7}
    cout << "Numero de componentes conectados: " << num_componentes << endl; // Saida: 4

    return 0;
}
```

Listing 15: Contando componentes conectados com DFS.

11.2 Caminho Mínimo (Algoritmo de Dijkstra)

Usado para encontrar a menor distância (ou custo) de um ponto de partida a todos os outros em um grafo com pesos não negativos.

Explicação do Funcionamento Dijkstra utiliza uma fila de prioridade para explorar sempre o vértice mais próximo (com menor distância acumulada) que ainda não foi finalizado. O algoritmo mantém um array `dist` com as menores distâncias conhecidas da origem. A cada passo, extrai o vértice `u` com menor distância da fila e, para cada vizinho `v`, verifica se o caminho passando por `u` é mais curto que a distância conhecida para `v`.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

const int INF = numeric_limits<int>::max();

void dijkstra(int start, int n, const vector<vector<pair<int, int>>>& adj) {
```

```
    vector<int> dist(n + 1, INF);
    dist[start] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, start}); // {distancia, vertice}

    while (!pq.empty()) {
        int u = pq.top().second;
        int d = pq.top().first;
        pq.pop();

        if (d > dist[u]) continue;

        for (auto const& [v, weight] : adj[u]) {
            if (dist[u] != INF && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    cout << "Distancias a partir de " << start << ":" << endl;
    for (int i = 1; i <= n; ++i) {
        cout << "  Para " << i << ": " << (dist[i] == INF ? "Infinito" : to_string(dist[i])) << endl;
    }
}

int main() {
    int n_nodes = 5;
    int start_node = 1;
    vector<vector<pair<int, int>>> adj(n_nodes + 1);

    adj[1].push_back({2, 2}); adj[2].push_back({1, 2});
    adj[1].push_back({3, 4}); adj[3].push_back({1, 4});
    adj[2].push_back({3, 1}); adj[3].push_back({2, 1});
    adj[2].push_back({4, 7}); adj[4].push_back({2, 7});
    adj[3].push_back({5, 3}); adj[5].push_back({3, 3});
    adj[4].push_back({5, 1}); adj[5].push_back({4, 1});

    dijkstra(start_node, n_nodes, adj);

    return 0;
}
```

Listing 16: Encontrando o caminho mais curto com Dijkstra.

11.3 Ordenação Topológica (Kahn's Algorithm)

Usada para ordenar linearmente os vértices de um Grafo Acíclico Dirigido (DAG) de forma que, para toda aresta $u \to v$, u venha antes de v na ordenação. É fundamental para problemas de dependências.

Explicação do Funcionamento O algoritmo de Kahn utiliza uma fila e um array para contar o "grau de entrada"(in-degree) de cada vértice. Inicialmente, todos os vértices com grau de entrada 0 são adicionados à fila. O algoritmo então processa a fila: remove um vértice, adiciona-o à ordenação final e, para cada um de seus vizinhos, decrementa o grau de entrada. Se o grau de entrada de um vizinho se tornar 0, ele é adicionado à fila.

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

using namespace std;

vector<int> topological_sort(int n, vector<vector<int>>& adj) {
    vector<int> in_degree(n + 1, 0);
    for (int u = 1; u <= n; ++u) {
```

```cpp
        for (int v : adj[u]) {
            in_degree[v]++;
        }
    }

    queue<int> q;
    for (int i = 1; i <= n; ++i) {
        if (in_degree[i] == 0) {
            q.push(i);
        }
    }

    vector<int> result;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        result.push_back(u);

        for (int v : adj[u]) {
            in_degree[v]--;
            if (in_degree[v] == 0) {
                q.push(v);
            }
        }
    }

    if (result.size() != n) {
        return {}; // O grafo tem um ciclo
    }
    return result;
}

int main() {
    int n_nodes = 6;
    vector<vector<int>> adj(n_nodes + 1);
    adj[5].push_back(2);
    adj[5].push_back(1);
    adj[4].push_back(1);
    adj[4].push_back(2);
    adj[2].push_back(3);
    adj[3].push_back(6);

    vector<int> sorted_order = topological_sort(n_nodes, adj);

    cout << "Ordem topologica: ";
    for (int node : sorted_order) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

Listing 17: Ordenação Topológica com o Algoritmo de Kahn.

11.4 Algoritmo de Floyd-Warshall

Calcula o caminho mais curto entre todos os pares de vértices em um grafo ponderado. É ideal para grafos pequenos e densos.

Explicação do Funcionamento É um algoritmo de programação dinâmica com complexidade $O(V^3)$. Ele inicializa uma matriz de distâncias com os pesos das arestas diretas. Em seguida, itera por todos os vértices k e, para cada par de vértices (i, j), verifica se passar por k cria um caminho mais curto de i para j.

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

```

```cpp
using namespace std;

const int INF = 1e9; // Um valor grande para representar infinito

void floyd_warshall(int n, vector<vector<int>>& dist) {
    for (int k = 1; k <= n; ++k) {
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (dist[i][k] != INF && dist[k][j] != INF) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }
}

int main() {
    int n_nodes = 4;
    vector<vector<int>> dist(n_nodes + 1, vector<int>(n_nodes + 1, INF));

    for (int i = 1; i <= n_nodes; ++i) dist[i][i] = 0;
    dist[1][2] = 5;
    dist[1][4] = 10;
    dist[2][3] = 3;
    dist[3][4] = 1;

    floyd_warshall(n_nodes, dist);

    cout << "Matriz de distancias minimas:" << endl;
    for (int i = 1; i <= n_nodes; ++i) {
        for (int j = 1; j <= n_nodes; ++j) {
            if (dist[i][j] == INF) cout << "INF ";
            else cout << dist[i][j] << "   ";
        }
        cout << endl;
    }

    return 0;
}
```

Listing 18: Caminho Mínimo para Todos os Pares com Floyd-Warshall.

11.5 Menor Ancestral Comum (LCA - Lowest Common Ancestor)

Dado uma árvore e dois nós `u` e `v`, o LCA é o nó mais profundo que é ancestral de ambos. Esta é uma consulta fundamental em uma vasta gama de problemas em árvores.

Explicação do Funcionamento A abordagem mais comum e eficiente é a de **Elevação Binária (Binary Lifting)**. Após uma única DFS para precalcular a profundidade de cada nó e o pai imediato, construímos uma tabela `up[i][j]`, que armazena o 2^j-ésimo ancestral do nó `i`. Com esta tabela, qualquer ancestral pode ser alcançado em tempo logarítmico. Para encontrar o LCA(u, v), primeiro nivelamos os nós para a mesma profundidade e, em seguida, "subimos"ambos simultaneamente em potências de dois até que seus pais sejam o mesmo nó.

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

using namespace std;

const int MAXN = 100001;
const int LOGN = 17; // ceil(log2(MAXN))

vector<int> adj[MAXN];
int depth[MAXN];
int up[MAXN][LOGN];
int n;
```

```cpp

void dfs_lca(int u, int p, int d) {
    depth[u] = d;
    up[u][0] = p;
    for (int i = 1; i < LOGN; ++i) {
        up[u][i] = up[up[u][i - 1]][i - 1];
    }
    for (int v : adj[u]) {
        if (v != p) {
            dfs_lca(v, u, d + 1);
        }
    }
}

int lca(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);

    for (int i = LOGN - 1; i >= 0; --i) {
        if (depth[u] - (1 << i) >= depth[v]) {
            u = up[u][i];
        }
    }

    if (u == v) return u;

    for (int i = LOGN - 1; i >= 0; --i) {
        if (up[u][i] != up[v][i]) {
            u = up[u][i];
            v = up[v][i];
        }
    }
    return up[u][0];
}

int main() {
    n = 7;
    adj[1].push_back(2); adj[2].push_back(1);
    adj[1].push_back(3); adj[3].push_back(1);
    adj[2].push_back(4); adj[4].push_back(2);
    adj[2].push_back(5); adj[5].push_back(2);
    adj[3].push_back(6); adj[6].push_back(3);
    adj[3].push_back(7); adj[7].push_back(3);

    dfs_lca(1, 1, 0);

    cout << "LCA(5, 4) = " << lca(5, 4) << endl;
    cout << "LCA(6, 7) = " << lca(6, 7) << endl;
    cout << "LCA(4, 6) = " << lca(4, 6) << endl;

    return 0;
}
```

Listing 19: LCA com Elevação Binária.

11.6 Fluxo Máximo (Maximum Flow)

Modela problemas que envolvem o transporte de "material"através de uma rede com capacidades. O objetivo é encontrar a quantidade máxima de fluxo que pode ser enviada de um nó fonte para um nó sumidouro.

Explicação do Funcionamento O **Algoritmo de Edmonds-Karp** é uma implementação do método de Ford-Fulkerson. Ele funciona encontrando repetidamente um "caminho de aumento"(um caminho da fonte ao sumidouro com capacidade residual positiva) no grafo residual. A busca pelo caminho é feita com uma BFS. O fluxo da rede é incrementado pelo gargalo (a menor capacidade residual) desse caminho. O processo se repete até que não existam mais caminhos de aumento.

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

```cpp
#include <queue>
#include <algorithm>

using namespace std;

const int INF = 1e9;

int max_flow(int s, int t, int n, vector<vector<int>>& capacity) {
    int flow = 0;
    vector<int> parent(n + 1);
    int new_flow;

    while (true) {
        fill(parent.begin(), parent.end(), -1);
        queue<pair<int, int>> q;
        q.push({s, INF});
        parent[s] = s;

        while(!q.empty()){
            int cur = q.front().first;
            int cur_flow = q.front().second;
            q.pop();

            for(int next = 1; next <= n; ++next){
                if(parent[next] == -1 && capacity[cur][next] > 0){
                    parent[next] = cur;
                    int new_pushed_flow = min(cur_flow, capacity[cur][next]);
                    if(next == t){
                        new_flow = new_pushed_flow;
                        goto end_bfs;
                    }
                    q.push({next, new_pushed_flow});
                }
            }
        }
        new_flow = 0;
        end_bfs:

        if (new_flow == 0) break;

        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }
    return flow;
}

int main() {
    int n = 6, s = 1, t = 6;
    vector<vector<int>> capacity(n + 1, vector<int>(n + 1, 0));

    capacity[1][2] = 16; capacity[1][3] = 13;
    capacity[2][3] = 10; capacity[2][4] = 12;
    capacity[3][2] = 4;  capacity[3][5] = 14;
    capacity[4][3] = 9;  capacity[4][6] = 20;
    capacity[5][4] = 7;  capacity[5][6] = 4;

    cout << "Fluxo maximo: " << max_flow(s, t, n, capacity) << endl;

    return 0;
}
```

Listing 20: Fluxo Máximo com Edmonds-Karp.

12 Programação Dinâmica

12.1 Abordagens: Memoização vs. Tabulação

O problema de Fibonacci é um exemplo clássico para ilustrar as duas principais abordagens da programação dinâmica.

1. Memoização (Top-Down) Utiliza uma abordagem recursiva. Se a solução para um subproblema já foi calculada e armazenada, ela é retornada imediatamente. Caso contrário, o subproblema é resolvido, e seu resultado é armazenado antes de ser retornado.

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

using namespace std;

vector<long long> memo;

long long fib_memo(int n) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    return memo[n] = fib_memo(n - 1) + fib_memo(n - 2);
}

int main() {
    int n = 50;
    memo.assign(n + 1, -1);
    cout << "Fibonacci(" << n << ") = " << fib_memo(n) << endl;
    return 0;
}
```

Listing 21: Fibonacci com Memoização (Top-Down).

2. Tabulação (Bottom-Up) Utiliza uma abordagem iterativa, construindo uma tabela "de baixo para cima", começando pelos casos base até atingir a solução para o problema original.

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

using namespace std;

long long fib_table(int n) {
    if (n <= 1) return n;
    vector<long long> table(n + 1);
    table[0] = 0;
    table[1] = 1;

    for (int i = 2; i <= n; ++i) {
        table[i] = table[i - 1] + table[i - 2];
    }
    return table[n];
}

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

Listing 22: Fibonacci com Tabulação (Bottom-Up).

12.2 Problema da Mochila 0/1

Dado um conjunto de itens com peso e valor, determina o subconjunto de itens que podem ser carregados em uma mochila de capacidade limitada de forma a maximizar o valor total.

Explicação do Funcionamento A solução clássica usa uma tabela onde $dp[i][w]$ é o valor máximo usando os primeiros i itens com capacidade w. Uma versão otimizada em espaço usa um array 1D. Para cada item, iteramos pela capacidade da mochila de trás para frente, decidindo se incluir o item atual leva a um valor total maior.

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

using namespace std;

int knapsack(int capacidade, const vector<int>& pesos, const vector<int>& valores) {
    int n = pesos.size();
    vector<int> dp(capacidade + 1, 0);

    for (int i = 0; i < n; ++i) {
        for (int w = capacidade; w >= pesos[i]; --w) {
            dp[w] = max(dp[w], valores[i] + dp[w - pesos[i]]);
        }
    }
    return dp[capacidade];
}

int main() {
    int capacidade_maxima = 10;
    vector<int> pesos = {5, 4, 6, 3};
    vector<int> valores = {10, 40, 30, 50};

    int resultado = knapsack(capacidade_maxima, pesos, valores);
    cout << "Valor maximo que pode ser levado: " << resultado << endl;

    return 0;
}
```

Listing 23: Solução para o Problema da Mochila 0/1.

12.3 Maior Subsequência Crescente (LIS)

Encontra o comprimento da subsequência mais longa de uma dada sequência onde os elementos estão em ordem crescente.

Explicação do Funcionamento Este código usa uma abordagem de DP com complexidade O(n²). `lis[i]` armazena o comprimento da LIS que termina no elemento `arr[i]`. Para calcular `lis[i]`, o algoritmo verifica todos os elementos `j` anteriores a `i`. Se `arr[i]` for maior que `arr[j]`, significa que `arr[i]` pode estender a subsequência que termina em `j`.

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

using namespace std;

int longestIncreasingSubsequence(const vector<int>& arr) {
    int n = arr.size();
    if (n == 0) return 0;

    vector<int> lis(n, 1);

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1) {
                lis[i] = lis[j] + 1;
            }
        }
    }

    return *max_element(lis.begin(), lis.end());
}
```

```
int main() {
    vector<int> sequencia = {10, 22, 9, 33, 21, 50, 41, 60};
    int resultado = longestIncreasingSubsequence(sequencia);
    cout << "Tamanho da maior subsequencia crescente: " << resultado << endl;

    return 0;
}
```

Listing 24: Encontrando a Maior Subsequência Crescente.

12.4 Subsequência Comum Mais Longa (LCS)

Dadas duas sequências, o LCS encontra o comprimento da subsequência mais longa que está presente em ambas.

Explicação do Funcionamento A solução de DP cria uma tabela `dp[i][j]` que armazena o comprimento da LCS entre os primeiros `i` caracteres da primeira sequência e os primeiros `j` da segunda. Se os caracteres `seq1[i-1]` e `seq2[j-1]` são iguais, a LCS aumenta em 1 em relação à LCS de `dp[i-1][j-1]`. Caso contrário, pega-se o máximo entre `dp[i-1][j]` e `dp[i][j-1]`.

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

using namespace std;

string longestCommonSubsequence(const string& s1, const string& s2) {
    int m = s1.length();
    int n = s2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    string lcs_str = "";
    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (s1[i - 1] == s2[j - 1]) {
            lcs_str += s1[i - 1];
            i--; j--;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }
    reverse(lcs_str.begin(), lcs_str.end());
    return lcs_str;
}

int main() {
    string s1 = "AGGTAB";
    string s2 = "GXTXAYB";
    cout << "LCS: " << longestCommonSubsequence(s1, s2) << endl; // Saida: GTAB
    return 0;
}
```

Listing 25: Encontrando a Subsequência Comum Mais Longa (LCS).

12.5 Problema da Soma de Subconjuntos (Subset Sum)

Dado um conjunto de inteiros e um valor alvo, este problema determina se existe um subconjunto cujos elementos somam exatamente o alvo.

Explicação do Funcionamento A solução de DP usa uma tabela booleana `dp[i][j]`, onde `dp[i][j]` é verdadeiro se uma soma `j` pode ser formada usando um subconjunto dos primeiros `i` elementos. Para cada elemento, ou ele não é incluído (e o resultado depende de `dp[i-1][j]`), ou ele é incluído (e o resultado depende de `dp[i-1][j - valor_do_item]`).

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

using namespace std;

bool subsetSum(const vector<int>& set, int targetSum) {
    int n = set.size();
    vector<bool> dp(targetSum + 1, false);
    dp[0] = true;

    for (int i = 0; i < n; ++i) {
        for (int j = targetSum; j >= set[i]; --j) {
            dp[j] = dp[j] || dp[j - set[i]];
        }
    }
    return dp[targetSum];
}

int main() {
    vector<int> conjunto = {3, 34, 4, 12, 5, 2};
    int soma_alvo = 9;

    if (subsetSum(conjunto, soma_alvo)) {
        cout << "Existe um subconjunto com a soma " << soma_alvo << endl;
    } else {
        cout << "Nao existe subconjunto com a soma " << soma_alvo << endl;
    }

    return 0;
}
```

Listing 26: Resolvendo o Problema da Soma de Subconjuntos.

12.6 Problema do Troco (Coin Change - Minimização)

Um algoritmo clássico de DP para encontrar o menor número de itens (moedas) necessários para atingir um valor-alvo, com repetição ilimitada de cada item.

Explicação do Funcionamento Cria-se um vetor `dp` de tamanho `V+1`, onde `dp[v]` representa o menor número de itens para atingir a soma `v`. Inicialmente, `dp[0]=0` e os outros são infinito. Para cada valor `v`, tenta-se usar cada tipo de moeda, atualizando `dp[v]` com o mínimo de moedas possível.

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

using namespace std;

int coinChange(const vector<int>& coins, int amount) {
    const int INF = 1e9;
    vector<int> dp(amount + 1, INF);
    dp[0] = 0;

    for (int i = 1; i <= amount; ++i) {
        for (int coin : coins) {
            if (i >= coin && dp[i - coin] != INF) {
                dp[i] = min(dp[i], dp[i - coin] + 1);
```

```
            }
        }
    }

    return (dp[amount] == INF) ? -1 : dp[amount];
}

int main() {
    vector<int> moedas = {1, 5, 10, 25};
    int valor_alvo = 37;

    int resultado = coinChange(moedas, valor_alvo);
    if (resultado != -1) {
        cout << "Menor numero de moedas para " << valor_alvo << ": " << resultado <<
    endl;
    } else {
        cout << "Nao e possivel formar o valor " << valor_alvo << endl;
    }

    return 0;
}
```

Listing 27: Soma exata com o menor número de itens (Coin Change).

12.7 Digit DP

Uma técnica de DP usada para contar números em um intervalo $[A, B]$ que satisfazem uma determinada propriedade baseada em seus dígitos.

Explicação do Funcionamento A função recursiva `dp(index, tight, ...)` constrói o número dígito por dígito. O estado `tight` indica se estamos restritos aos dígitos do número original. Se escolhermos um dígito menor que o máximo, a restrição `tight` se torna falsa para os próximos dígitos, permitindo que eles sejam de 0 a 9.

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

using namespace std;

string S;
int memo[20][2]; // index, tight

int solve(int pos, bool tight) {
    if (pos == S.length()) return 1;
    if (memo[pos][tight] != -1) return memo[pos][tight];

    int count = 0;
    int upper_bound = tight ? (S[pos] - '0') : 9;

    for (int digit = 0; digit <= upper_bound; ++digit) {
        if (digit == 3) continue;

        bool new_tight = tight && (digit == upper_bound);
        count += solve(pos + 1, new_tight);
    }

    return memo[pos][tight] = count;
}

int countValid(int n) {
    S = to_string(n);
    memset(memo, -1, sizeof(memo));
    return solve(0, true);
}

int main() {
    int A = 1, B = 345;
```

```
    // count(B) - count(A-1)
    // Nao contamos o '0' como um numero valido.
    int result = (countValid(B) - 1) - (countValid(A - 1) - 1);

    cout << "Numeros sem o digito '3' no intervalo [" << A << ", " << B << "]: " <<
    result << endl;

    return 0;
}
```

Listing 28: Digit DP para contar números sem o dígito '3' até N.

13 Teoria dos Números

13.1 Crivo de Eratóstenes

Um algoritmo eficiente para encontrar todos os números primos até um determinado limite.

Explicação do Funcionamento Cria-se uma lista booleana `is_prime`, onde todos os números são inicialmente marcados como primos. O algoritmo itera a partir de 2. Se o número `p` atual for primo, ele marca todos os seus múltiplos como não-primos. A iteração principal só precisa ir até $\sqrt{n}$.

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

using namespace std;

void sieveOfEratosthenes(int n) {
    vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;

    for (int p = 2; p * p <= n; p++) {
        if (is_prime[p]) {
            for (int i = p * p; i <= n; i += p)
                is_prime[i] = false;
        }
    }

    cout << "Primos ate " << n << ":" << endl;
    for (int p = 2; p <= n; p++) {
        if (is_prime[p]) {
            cout << p << " ";
        }
    }
    cout << endl;
}

int main() {
    sieveOfEratosthenes(100);
    return 0;
}
```

Listing 29: Gerando números primos com o Crivo de Eratóstenes.

13.2 Segmented Sieve

Uma otimização do Crivo de Eratóstenes para encontrar primos em um intervalo $[L, R]$, especialmente quando L e R são grandes, mas a diferença $R - L$ é relativamente pequena.

Explicação do Funcionamento Primeiro, ele usa um crivo simples para encontrar todos os primos até $\sqrt{R}$. Em seguida, para cada um desses primos encontrados, ele os utiliza para marcar seus múltiplos dentro do intervalo específico $[L, R]$. Isso evita a necessidade de alocar um array booleano de tamanho R, que seria inviável para valores grandes.

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

using namespace std;

void segmentedSieve(long long L, long long R) {
    long long limit = sqrt(R);
    vector<bool> is_prime_small(limit + 1, true);
    is_prime_small[0] = is_prime_small[1] = false;
    for (long long p = 2; p * p <= limit; ++p) {
        if (is_prime_small[p]) {
            for (long long i = p * p; i <= limit; i += p)
                is_prime_small[i] = false;
        }
    }

    vector<long long> primes;
    for (long long p = 2; p <= limit; ++p) {
        if (is_prime_small[p]) {
            primes.push_back(p);
        }
    }

    vector<bool> is_prime_segment(R - L + 1, true);
    for (long long p : primes) {
        long long start = max(p * p, (L + p - 1) / p * p);
        for (long long j = start; j <= R; j += p) {
            is_prime_segment[j - L] = false;
        }
    }

    if (L == 1) is_prime_segment[0] = false;

    cout << "Primos no intervalo [" << L << ", " << R << "]: ";
    for (long long i = 0; i < R - L + 1; ++i) {
        if (is_prime_segment[i]) {
            cout << L + i << " ";
        }
    }
    cout << endl;
}

int main() {
    segmentedSieve(100, 200);
    return 0;
}
```

Listing 30: Encontrando primos em um intervalo com Segmented Sieve.

13.3 Algoritmo de Euclides para MDC

Calcula o Máximo Divisor Comum (MDC) de dois inteiros.

Explicação do Funcionamento Baseia-se no princípio de que o MDC(a, b) é igual ao MDC$(b, a \pmod b)$. A recursão continua até que o segundo número se torne 0, momento em que o primeiro número é o MDC.

```cpp
#include <iostream>
#include <numeric>

using namespace std;

long long gcd(long long a, long long b) {
    return b == 0 ? a : gcd(b, a % b);
}

int main() {
```

```
    long long num1 = 54;
    long long num2 = 24;

    cout << "MDC(" << num1 << ", " << num2 << ") = " << gcd(num1, num2) << endl;

    return 0;
}
```

Listing 31: Calculando o Máximo Divisor Comum (MDC).

14 Estruturas de Dados e Técnicas Comuns

14.1 Problema "Two Sum"com Hash Map

Dado um array de números e um alvo, encontra dois números no array que somam o alvo.

Explicação do Funcionamento O código utiliza um `unordered_map` para otimizar a busca. Ele percorre o array uma única vez. Para cada elemento, calcula o complemento necessário para atingir o alvo. Em seguida, verifica se esse complemento já existe no mapa. Se existir, encontrou o par. Se não, insere o elemento atual e seu índice no mapa, reduzindo a complexidade para O(n).

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

using namespace std;

vector<int> twoSum(const vector<int>& nums, int target) {
    unordered_map<int, int> map; // Armazena <numero, indice>
    for (int i = 0; i < nums.size(); ++i) {
        int complement = target - nums[i];
        if (map.count(complement)) {
            vector<int> result = {map[complement], i};
            sort(result.begin(), result.end());
            return result;
        }
        map[nums[i]] = i;
    }
    return {}; // Retorna vazio se nao encontrar
}

int main() {
    int target = 9;
    vector<int> numeros = {2, 7, 11, 15};

    vector<int> indices = twoSum(numeros, target);

    if (!indices.empty()) {
        cout << "Indices que somam " << target << ": " << indices[0] << " e " <<
    indices[1] << endl;
    } else {
        cout << "Nenhum par encontrado." << endl;
    }

    return 0;
}
```

Listing 32: Resolvendo o problema Two Sum com Hash Map.

14.2 Disjoint Set Union (DSU / Union-Find)

Estrutura de dados para gerenciar uma partição de um conjunto em subconjuntos disjuntos. É extremamente rápida para determinar se dois elementos estão no mesmo grupo e para unir grupos.

Explicação do Funcionamento Usa uma floresta de árvores para representar os conjuntos. Cada árvore corresponde a um conjunto, e a raiz é o representante. As otimizações de "união por tamanho"(sempre anexa a árvore menor à maior) e "compressão de caminho"(faz com que todos os nós no caminho de uma busca apontem diretamente para a raiz) tornam as operações quase de tempo constante.

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

using namespace std;

struct DSU {
    vector<int> parent;
    vector<int> sz;
    DSU(int n) {
        parent.resize(n + 1);
        iota(parent.begin(), parent.end(), 0);
        sz.assign(n + 1, 1);
    }

    int find_set(int v) {
        if (v == parent[v]) return v;
        return parent[v] = find_set(parent[v]); // Compressao de caminho
    }

    void union_sets(int a, int b) {
        a = find_set(a);
        b = find_set(b);
        if (a != b) {
            if (sz[a] < sz[b]) swap(a, b); // Uniao por tamanho
            parent[b] = a;
            sz[a] += sz[b];
        }
    }
};

int main() {
    int n_elementos = 5;
    DSU dsu(n_elementos);

    dsu.union_sets(1, 2);
    dsu.union_sets(2, 3);
    dsu.union_sets(4, 5);

    cout << "1 e 3 estao no mesmo conjunto? " << (dsu.find_set(1) == dsu.find_set(3) ? "Sim" : "Nao") << endl;
    cout << "1 e 4 estao no mesmo conjunto? " << (dsu.find_set(1) == dsu.find_set(4) ? "Sim" : "Nao") << endl;

    return 0;
}
```

Listing 33: Implementação de DSU com Otimizações.

14.3 Árvore de Segmentos (Segment Tree)

Uma estrutura de dados versátil para responder a consultas sobre intervalos de um array (soma, mínimo, máximo) em tempo logarítmico.

Explicação do Funcionamento É uma árvore binária onde cada nó armazena uma informação agregada sobre um segmento do array. A raiz representa o array inteiro $[0, N-1]$. Um nó que representa o intervalo $[L, R]$ tem filhos que representam $[L, M]$ e $[M+1, R]$, onde M é o ponto médio. Consultas e atualizações de um ponto são feitas em $O(\log N)$.

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

```

```cpp
using namespace std;

vector<int> arr;
vector<int> tree;

void build(int node, int start, int end) {
    if (start == end) {
        tree[node] = arr[start];
        return;
    }
    int mid = start + (end - start) / 2;
    build(2 * node, start, mid);
    build(2 * node + 1, mid + 1, end);
    tree[node] = tree[2 * node] + tree[2 * node + 1];
}

void update(int node, int start, int end, int idx, int val) {
    if (start == end) {
        arr[idx] = val;
        tree[node] = val;
        return;
    }
    int mid = start + (end - start) / 2;
    if (start <= idx && idx <= mid) {
        update(2 * node, start, mid, idx, val);
    } else {
        update(2 * node + 1, mid + 1, end, idx, val);
    }
    tree[node] = tree[2 * node] + tree[2 * node + 1];
}

int query(int node, int start, int end, int l, int r) {
    if (r < start || end < l) return 0;
    if (l <= start && end <= r) return tree[node];
    int mid = start + (end - start) / 2;
    return query(2 * node, start, mid, l, r) + query(2 * node + 1, mid + 1, end, l, r);
}

int main() {
    arr = {1, 3, 5, 7, 9, 11};
    int n = arr.size();
    tree.resize(4 * n);

    build(1, 0, n - 1);

    cout << "Soma do intervalo [1, 3]: " << query(1, 0, n - 1, 1, 3) << endl; // 3+5+7=15
    update(1, 0, n - 1, 2, 6); // Muda o 5 para 6
    cout << "Nova soma do intervalo [1, 3]: " << query(1, 0, n - 1, 1, 3) << endl; // 3+6+7=16

    return 0;
}
```

Listing 34: Árvore de Segmentos para Soma de Intervalo.

15 Paradigmas e Técnicas de Otimização

15.1 Busca Binária na Resposta

Uma técnica poderosa para problemas de otimização que pedem o valor "mínimo possível"ou "máximo possível"que satisfaz uma condição.

Explicação do Funcionamento Se podemos criar uma função `check(x)` que nos diz se um valor `x` é uma resposta "válida"ou "possível", e essa propriedade é monotônica, podemos usar busca binária no intervalo de possíveis respostas para encontrar o valor ótimo.

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

using namespace std;

// Funcao que verifica se e possivel alocar k vacas com distancia minima 'dist'
bool check(int dist, int k, const vector<int>& posicoes) {
    int vacas_alocadas = 1;
    int ultima_posicao = posicoes[0];
    for (size_t i = 1; i < posicoes.size(); ++i) {
        if (posicoes[i] - ultima_posicao >= dist) {
            vacas_alocadas++;
            ultima_posicao = posicoes[i];
        }
    }
    return vacas_alocadas >= k;
}

int main() {
    // Problema classico: alocar k vacas em N baias para maximizar a distancia minima
    entre elas
    int n_baias = 5, k_vacas = 3;
    vector<int> pos_baias = {1, 2, 8, 4, 9};
    sort(pos_baias.begin(), pos_baias.end());

    int low = 0, high = 1e9, ans = 0;
    while(low <= high) {
        int mid = low + (high - low) / 2;
        if (check(mid, k_vacas, pos_baias)) {
            ans = mid;
            low = mid + 1; // Tenta uma distancia maior
        } else {
            high = mid - 1; // A distancia e muito grande
        }
    }

    cout << "Distancia minima maxima possivel: " << ans << endl; // Saida: 3

    return 0;
}
```

Listing 35: Exemplo de Busca Binária na Resposta.

15.2 Técnica dos Dois Ponteiros (Two Pointers)

Otimiza buscas por pares ou subarrays em um array ordenado, geralmente reduzindo a complexidade de $O(N^2)$ para $O(N)$.

Explicação do Funcionamento Dois ponteiros, `left` e `right`, são inicializados nas extremidades do array. Eles se movem um em direção ao outro com base na soma dos valores que apontam. Se a soma for menor que o alvo, `left` avança para aumentar a soma. Se for maior, `right` recua para diminuir a soma.

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

using namespace std;

pair<int, int> findPair(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left < right) {
        int sum = arr[left] + arr[right];
        if (sum == target) {
            return {arr[left], arr[right]};
        } else if (sum < target) {
```

```
            left++;
        } else {
            right--;
        }
    }
    return {-1, -1}; // Nao encontrou
}

int main() {
    vector<int> sorted_arr = {2, 7, 11, 15, 20, 28};
    int target_sum = 27;

    pair<int, int> result = findPair(sorted_arr, target_sum);

    if (result.first != -1) {
        cout << "Par encontrado: " << result.first << " e " << result.second << endl;
    } else {
        cout << "Nenhum par encontrado." << endl;
    }

    return 0;
}
```

Listing 36: Encontrando um par com soma alvo em O(N).

15.3 Backtracking

Uma forma refinada da busca completa, que explora recursivamente o espaço de soluções. A chave é que, se uma escolha parcial leva a um estado inválido, o algoritmo "retrocede"(backtracks), desfaz a escolha e tenta a próxima alternativa, podando galhos inteiros da árvore de busca.

Explicação do Funcionamento O exemplo clássico é o problema das N-Rainhas. A função recursiva tenta colocar uma rainha em cada coluna. Para cada linha em uma coluna, ela verifica se a posição é segura. Se for, ela marca a posição, faz uma chamada recursiva para a próxima coluna e, crucialmente, desmarca a posição após o retorno da chamada para explorar outras possibilidades.

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

using namespace std;

int n_queens_count = 0;
int board_size;

bool is_safe(int row, int col, const vector<string>& board) {
    for (int i = 0; i < col; i++)
        if (board[row][i] == 'Q') return false;
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 'Q') return false;
    for (int i = row, j = col; j >= 0 && i < board_size; i++, j--)
        if (board[i][j] == 'Q') return false;
    return true;
}

void solveNQueens(int col, vector<string>& board) {
    if (col >= board_size) {
        n_queens_count++;
        return;
    }

    for (int i = 0; i < board_size; i++) {
        if (is_safe(i, col, board)) {
            board[i][col] = 'Q';
            solveNQueens(col + 1, board);
            board[i][col] = '.'; // Backtrack
        }
    }
}
```

```cpp

int main() {
    board_size = 8;
    vector<string> board(board_size, string(board_size, '.'));

    solveNQueens(0, board);

    cout << "Numero de solucoes para o problema das " << board_size << "-Rainhas: " <<
     n_queens_count << endl;

    return 0;
}
```

Listing 37: Resolvendo o Problema das N-Rainhas com Backtracking.

16 Manipulação de Strings e Bits

16.1 Knuth-Morris-Pratt (KMP)

Um algoritmo de busca de padrões em strings altamente eficiente. Ele encontra todas as ocorrências de um padrão em um texto em tempo linear, $O(N + M)$.

Explicação do Funcionamento A genialidade do KMP está em sua fase de pré-processamento. Ele cria um array auxiliar `lps` (Longest Proper Prefix which is also Suffix) para o padrão. Este array armazena, para cada posição do padrão, o comprimento do maior prefixo próprio que também é um sufixo. Quando ocorre uma incompatibilidade durante a busca, o array `lps` informa exatamente quantos caracteres para frente o padrão pode ser deslocado, evitando comparações redundantes.

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

using namespace std;

vector<int> computeLPSArray(const string& pattern) {
    int m = pattern.length();
    vector<int> lps(m, 0);
    int length = 0;
    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;
}

void KMPSearch(const string& pattern, const string& text) {
    int m = pattern.length();
    int n = text.length();
    vector<int> lps = computeLPSArray(pattern);
    int i = 0; // indice para o texto
    int j = 0; // indice para o padrao

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

```
            j++;
        }
        if (j == m) {
            cout << "Padrao encontrado no indice " << (i - j) << endl;
            j = lps[j - 1];
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

int main() {
    string text = "ABABDABACDABABCABAB";
    string pattern = "ABABCABAB";
    KMPSearch(pattern, text);

    return 0;
}
```

Listing 38: Busca de Padrão em String com KMP.

16.2 Manipulação de Bits e Bitmasks

Operações bit-a-bit são extremamente rápidas e fundamentais para otimizar código e resolver problemas que envolvem conjuntos e estados.

Operadores Bit-a-Bit

- **AND (&):** Retorna 1 se ambos os bits correspondentes forem 1. Útil para verificar se um bit está "ligado".
- **OR (|):** Retorna 1 se pelo menos um dos bits correspondentes for 1. Útil para "ligar"um bit.
- **XOR (^):** Retorna 1 se os bits correspondentes forem diferentes. Útil para "inverter"um bit ou encontrar um elemento único.
- **NOT (~):** Inverte todos os bits de um número.
- **Left Shift («):** Desloca os bits para a esquerda. `x « n` é equivalente a $x \times 2^n$.
- **Right Shift (»):** Desloca os bits para a direita. `x » n` é equivalente a $x/2^n$.

Gerando Todos os Subconjuntos Uma das aplicações mais poderosas de bitmasks é a capacidade de iterar por todos os 2^N subconjuntos de um conjunto de N elementos de forma concisa. Como um inteiro de N bits pode representar qualquer subconjunto, podemos simplesmente iterar com um `for` de 0 até $2^N - 1$.

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

using namespace std;

void printAllSubsets(const vector<char>& S) {
    int n = S.size();
    for (int i = 0; i < (1 << n); ++i) {
        cout << "{ ";
        for (int j = 0; j < n; ++j) {
            if ((i & (1 << j)) != 0) {
                cout << S[j] << " ";
            }
        }
        cout << "}" << endl;
    }
```

```cpp
}

int main() {
    vector<char> conjunto = {'a', 'b', 'c'};
    printAllSubsets(conjunto);
    return 0;
}
```

Listing 39: Iterando por todos os subconjuntos de um conjunto.