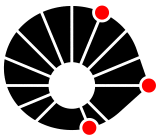

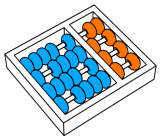

Tópicos em Programação Competitiva: Caminhos, Árvores, Combinatória e Probabilidade

B. Archegas *F. Usberti*

Relatório Técnico - IC-PFG-25-14
Projeto Final de Graduação
2025 - Junho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

The contents of this report are the sole responsibility of the authors.
O conteúdo deste relatório é de única responsabilidade dos autores.

Tópicos em Programação Competitiva: Caminhos, Árvores, Combinatória e Probabilidade

Bernardo Archegas Fábio Usberti

Resumo

Este trabalho é o relatório do Projeto Final de Graduação em Engenharia de Computação. O projeto consistiu na escrita de quatro capítulos para o livro MaratonIC, que tem como objetivo ensinar técnicas de Programação Competitiva voltadas a competições como a Olimpíada Brasileira de Informática (OBI) e a Maratona de Programação.

Os capítulos desenvolvidos abordam temas centrais na área. Um deles trata de Caminhos Mínimos, incluindo algoritmos como Dijkstra e BFS. Outro aborda Análise Combinatória e Probabilidade, com foco em técnicas fundamentais para resolver problemas envolvendo contagem e eventos probabilísticos. O capítulo sobre Árvores discute métodos como Decomposição de Centróide, Decomposição de Leves e Pesados (Heavy-Light), Small to Large e DSU na árvore. Por fim, o capítulo de Programação Dinâmica com Bitmasks explora técnicas para resolver problemas com estados representados por subconjuntos.

Ao todo, foram produzidas mais de 60 páginas de conteúdo técnico e didático, incluídas no corpo deste relatório.

Acreditamos que este material terá um impacto positivo para as futuras gerações de competidores da Unicamp e de outras instituições no Brasil. Ele oferece uma base sólida de estudo e prática para quem deseja se preparar para competições, servindo como um recurso complementar ao treinamento regular e facilitando o acesso a conteúdos avançados de forma estruturada.

1 Truques para caminhos mínimos

Nesse capítulo, abordaremos problemas que envolvem caminho mínimo entre vértices em um grafo, focando em truques interessantes e variações dos algoritmos clássicos BFS e Dijkstra e diferentes modelagens.

1.1 BFS Multisource

A maioria dos problemas de caminhos mínimos nos pede a distância entre dois vértices ou a distância de todos os vértices para um vértice escolhido, normalmente chamado de source. Para resolver esses problemas, implementamos uma Breadth-First Search (BFS) quando as arestas têm peso unitário e Dijkstra quando as arestas têm pesos não negativos. Nessa seção vamos aprender a estender esses algoritmos para outros casos.

Exercício

(CSES - Monsters) Você está preso em um labirinto cheio de monstros. Quando você dá um passo em alguma direção, todos os monstros podem simultaneamente dar um passo também. Você quer escapar do labirinto sem passar pelo mesmo quadrado que algum monstro.
Sua tarefa é descobrir se isso é possível, e se sim, encontrar um caminho que funcione mesmo que os monstros saibam o caminho que você vai percorrer.

O importante é notar que os monstros se movem de forma ótima, podendo conhecer nossa estratégia. Dessa forma, nós só podemos nos mover para uma posição se alcançarmos essa posição antes que todos os monstros, ou seja, se nossa distância for menor que a distância de qualquer monstro para aquela posição.

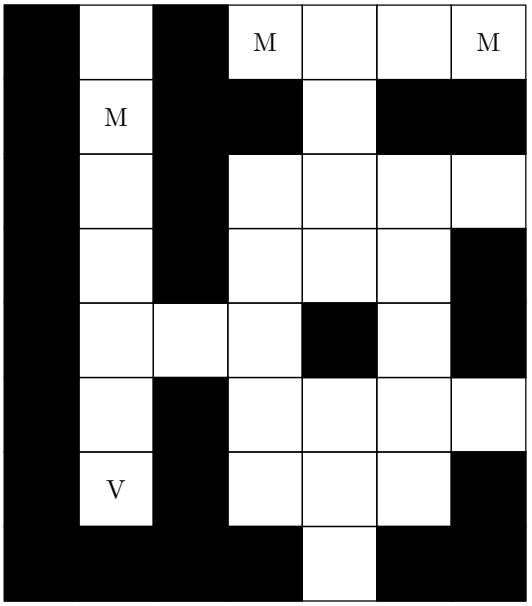

Figure 1.1
Tabuleiro de exemplo para o problema Monsters. Os monstros estão representados por M, você está representado por V. Os quadrados brancos estão livres e os pretos estão bloqueados. As saídas são os quadrados brancos da borda.

Com isso, podemos pensar em uma solução: Como o problema é em um grid, consideramos os quadrados como nós e adicionamos arestas entre quadrados adjacentes (com exceção dos quadrados bloqueados). Já que as arestas não têm peso, podemos rodar uma BFS por monstro e achar para cada posição do tabuleiro a menor distância para um monstro.

Agora, fazemos uma BFS saindo da nossa posição e tentamos alcançar alguma borda do tabuleiro, com a condição de que nossa distância para um vértice seja sempre menor que a distância de todos os monstros para aquele vértice.

Seguindo essa ideia, na Figura 1.1, o único caminho possível seria subir 2 casas, andar 2 para a direita e então se dirigir para a saída de baixo. Em qualquer outra saída um monstro conseguiria alcançá-lo.

No entanto, essa solução tem complexidade de tempo $O(V^2)$, em que V é o número de vértices do grafo, já que temos $O(V)$ monstros e cada BFS roda em $O(V)$.

Para melhorar isso, podemos usar uma BFS multisource. A mudança é simples: No início do algoritmo, ao invés de inserir apenas um vértice na fila com distância 0, inserimos todos os vértices que são sources.

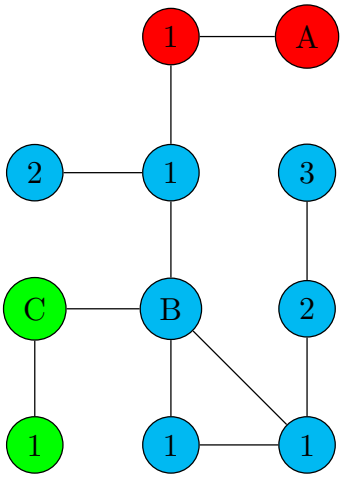

Figure 1.2
BFS multisource, em que as sources estão marcadas com letras e o resto dos vértices com a distância à source mais próxima e cor correspondente.

Considere esse algoritmo no grafo da Figura 1.2, em que as letras são as sources e os vértices foram pintados para indicar a source mais próxima.

Ao serem colocados na fila, os vértices são marcados como visitados por um vértice pai da mesma cor. Dessa forma, todos os vértices serão colocados na fila da BFS uma única vez, de modo que a BFS multisource é linear no número de vértices e arestas.

Aplicando essa ideia ao problema dos monstros, conseguimos resolvê-lo em tempo linear.

```
vector<int> dist(n, n);
vector<int> parent_source(n);
queue<int> fila;
for (int source : sources) {
    parent_source[source] = source;
    dist[source] = 0;
    fila.push(source);
}
while (!fila.empty()) {
    int at = fila.front();
    fila.pop();
    for (int x : v[at]) { // v[at] são os vizinhos de at
```

```
        int vizinho = x;
        if (dist[vizinho] > dist[at] + 1) {
            dist[vizinho] = dist[at] + 1;
            parent_source[vizinho] = parent_source[at];
            fila.push(vizinho);
        }
    }
}
```

Código 1.1
BFS multisource

No Código 1.1, temos uma implementação da BFS multisource que calcula a distância para a source mais próxima e quem é essa source mais próxima para todos os vértices do grafo. No Monsters não precisamos saber qual é o monstro que está mais perto de cada célula, apenas qual é essa distância. Todavia, saber calcular o vetor *parent_source* é importante para uma ampla gama de problemas.

Também vale ressaltar que essa ideia de múltiplas sources pode ser usada para modificar o algoritmo do Dijkstra no caso de grafos com pesos nas arestas. Nesse caso, ao invés de inserir uma única source à priority queue no início do Dijkstra, inserimos múltiplos vértices.

1.2 Tornando o grafo acíclico

Sabemos que o algoritmo do Dijkstra/BFS pode ser utilizado para encontrar a distância mínima. Mas como lidar quando o problema pede mais informações sobre os diferentes caminhos mínimos?

Exercício

(CSES - Investigation - modificado) Você tem um grafo direcionado em que as arestas têm peso. Você quer ir do vértice 1 ao n por um caminho cuja soma dos pesos das arestas seja mínimo. Encontre um caminho de peso mínimo com o menor número de arestas.

Uma das dificuldades desse problema é que a quantidade de caminhos mínimos em um grafo pode ser exponencial, tornando inviável enumerá-los.

Vamos tomar o grafo da Figura 1.3 de exemplo. Note que esse grafo tem ciclos e existem vários caminhos de 1 até $n = 6$, e nem todos são mínimos. No entanto, considere o grafo da Figura 1.4. Esse grafo é um

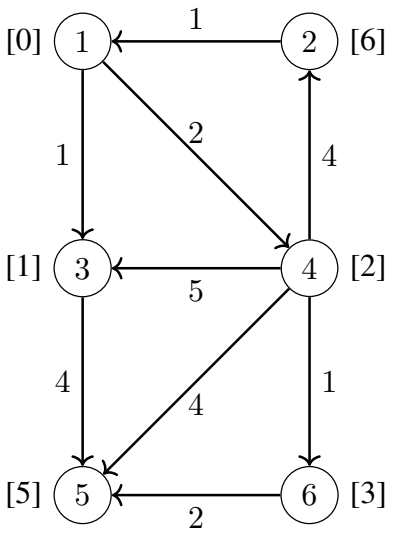

Figure 1.3
Grafo direcionado com pesos nas arestas. Cada vértice x está marcado com a distância mínima ao vértice 1, chamada de $dist[x]$.

Directed Acyclic Graph (DAG), ou seja, não tem ciclos, e todo caminho de 1 até n é mínimo. Porque isso ocorre?

Definindo $dist[i]$ como a distância mínima de 1 até i, percebemos que o grafo da Figura 1.4 foi construído removendo arestas do grafo original, de forma que só foram mantidas as arestas (u, v) de peso c que satisfazem $dist[u] + c = dist[v]$. Isso significa que qualquer aresta nesse grafo nos leva para um vértice mais longe do nó 1 do que estávamos, o que garante a não existência de ciclos. Além disso, independente do caminho que tomarmos para chegar no vértice x, esse caminho será mínimo e terá custo $dist[x]$.

Chamamos esse grafo da Figura 1.4 de DAG do Dijkstra (ou DAG da BFS, para os grafos sem peso), e com ele conseguimos várias informações sobre os caminhos mínimos.

Como todo caminho de 1 a n no DAG do Dijkstra é mínimo, conseguimos facilmente resolver o Investigation: basta rodar uma BFS saindo do vértice 1 e checar a quantas arestas de distância estamos do vértice n. A BFS vai encontrar o caminho mínimo que usa menos arestas.

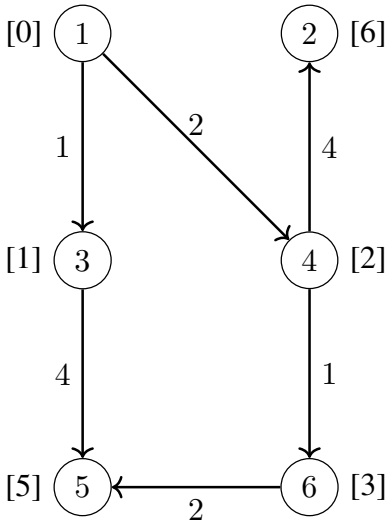

Figure 1.4
Grafo direcionado acíclico com pesos nas arestas. Toda aresta (u, v) de peso c nesse grafo é tal que $dist[u] + c = dist[v]$.

Exercício

(CSES - Visiting Cities) Você tem um grafo direcionado em que as arestas têm peso. Você quer ir do vértice 1 ao n por um caminho cuja soma dos pesos das arestas seja mínimo. Quais cidades você certamente vai visitar?

Novamente, construímos o DAG do Dijkstra, mas agora queremos saber quais vértices estão em todos os caminhos do vértice 1 ao n. Existe mais de uma forma de resolver esse problema, mas uma interessante é contar $qtd1[i]$ = quantos caminhos existem de 1 até i e $qtdn[i]$ = quantos caminhos existem de i até n. Um vértice i está em todo caminho de 1 até n se o produto $qtd1[i] \cdot qtdn[i]$ for igual ao total de caminhos de 1 a n.

Para calcular $qtd1$ podemos usar programação dinâmica em cima do DAG, processando os vértices em alguma ordenação topológica.

Já para calcular $qtdn$, fazemos algo análogo, mas primeiro invertendo as arestas do grafo, visto que o número de caminhos de i a n é igual ao de n a i no grafo invertido.

Uma dica importante para essa abordagem é que, como o número de caminhos pode ser exponencial, o ideal é guardá-lo módulo algum primo grande, como $10^9 + 7$.

Note que isso pode dar um falso positivo, como no caso em que o total de caminhos for $10^9 + 9$ e os que passam por x for 2, por exemplo. Um falso positivo pode ser improvável, mas não tão improvável quando se testam 10^5 vértices em 100 casos de teste.

Então o mais seguro é calcular os valores com dois módulos diferentes, e só considerar igualdade quando os valores forem iguais sobre os dois módulos distintos.

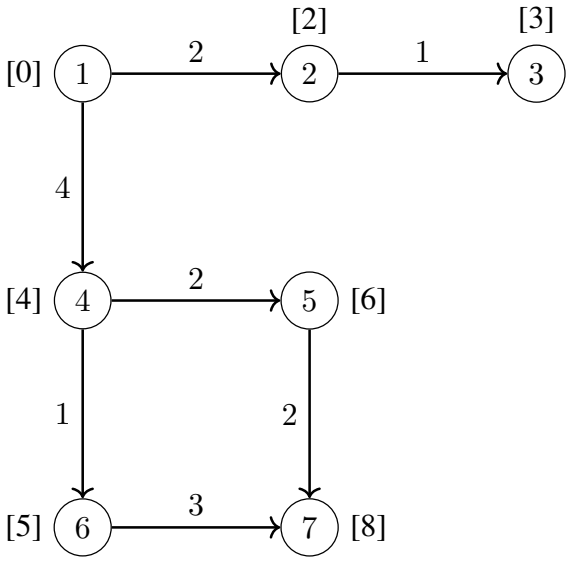

Figure 1.5
Grafo direcionado acíclico do problema Visiting Cities.

Uma outra solução envolve checar propriedades do DAG. Considere a figura 1.5. Os vértices que estão em todos os caminhos são 1, 4 e 7.

Primeiro devemos marcar quais vértices de fato alcançam n, o que eliminaria os vértices 2 e 3. Com o grafo que sobrar, percebemos que os vértices que estão em todos os caminhos são aqueles que, ao apagá-los, separam o 1 e o n em componentes distintas. Isso significa que não existe nenhum caminho mínimo que conecta a componente do 1 à componente do n que não passa pelo vértice apagado.

Encontrar esse vértices é equivalente a encontrar os pontos de articulação do nosso grafo modificado, o que pode ser feito com algoritmos clássicos não cobertos neste livro.

1.3 Truques para o Dijkstra

Vimos como retirar arestas para tornar o grafo acíclico. No entanto, alguns problemas necessitam que adicionemos novos vértices e arestas para serem resolvidos. Veremos a seguir alguns exemplos desses.

Exercício

(CSES - Flight Discount) Você tem um grafo direcionado em que as arestas têm peso. Você também tem um ticket especial que pode ser usado uma única vez e reduz o peso de uma aresta pela metade. Qual o peso do menor caminho do vértice 1 ao n?

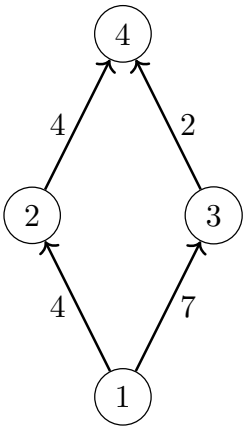

Figure 1.6
Grafo do problema Flight Discount.

Uma primeira ideia para resolver esse problema pode ser encontrar um caminho mínimo qualquer e então subtrair metade de sua aresta mais pesada. Mas olhando para o grafo da figura 1.6 notamos que o caminho mínimo sem usar o ticket seria $1-2-4$ com peso 8 e usando ticket ficaria com peso 6. No entanto, temos o caminho é $1-3-4$ com peso 9 sem usar o ticket, mas peso 5.5 usando o ticket, o que quebra nossa ideia inicial.

Uma solução correta consiste em adicionar uma estado adicional por vértice, esse estado será uma flag que indica se já usamos ou não o ticket. Começamos então no vértice $(1, 0)$ e queremos chegar no vértice $(n, 1)$, ou seja, começamos sem usar o ticket no vértice 1 e queremos chegar no vértice n tendo usado o ticket.

Se antes tínhamos uma aresta de a para b com peso c, agora temos uma aresta de $(a, 0)$ para $(b, 0)$ com peso c, uma de $(a, 1)$ para $(b, 1)$ com peso c e uma de $(a, 0)$ para $(b, 1)$ com peso $c/2$.

Com essa duplicação do grafo, podemos rodar o Dijkstra normalmente e resolver o problema em $O(nlogn)$.

Exercício

(SPOJ - Fisher) Você tem um grafo não direcionado em que as arestas tem um custo e um tempo para ser atravessada. Qual o menor custo para viajar de 1 até n em um tempo $\leq t$?.

Note que não queremos minimizar o tempo, queremos minimizar o custo enquanto mantemos o tempo abaixo de um parâmetro.

Podemos levar a ideia do problema anterior adiante e adicionar ao grafo um estado novo representando o tempo. Assim, cada nó no grafo será representado por (posição, tempo).

Uma aresta de a para b com tempo x e custo y torna-se múltiplas arestas de (a, i) para $(b, i + x)$ com custo y para todo $i = 0$ até $i = t - x$. E queremos o menor caminho de $(1, 0)$ para (n, x) em que $x \leq t$.

Nesse caso, ao rodar o Dijkstra, resolvemos o problema em $O(nt \cdot log(nt))$, visto que agora temos $O(nt)$ vértices e arestas.

Exercício

(CSES - Flight Routes) Você tem um grafo direcionado em que as arestas têm peso. Qual o peso do k-ésimo menor caminho do vértice 1 ao n?

Sabemos que o algoritmo de Dijkstra nos devolve o peso do menor caminho do vértice 1 ao n. Também sabemos que podemos usar o DAG do Dijkstra para calcular quantos caminhos mínimos existem, de modo que se k for menor que esse total, o resultado ainda vai ser o peso do menor caminho. Mas e se k for maior, como podemos resolver?

Ao rodar o Dijkstra, ao invés de manter a distância do menor caminho para cada vértice, podemos manter as distâncias dos k menores caminhos, criando uma priority queue para todo vértice.

```
using pll = pair<long long, long long>;
void dijkstra() {
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < k; j++) dist[i].push(INF);
    }
    dist[1].pop();
    dist[1].push(0);
    priority_queue<pll, vector<pll>, greater<pll>> fila;
    fila.push({0, 1});
    while (!fila.empty()) {
        pll a = fila.top();
        int atual = a.second;
        ll dist_atual = a.first;
        fila.pop();
        if (dist_atual > dist[atual].top()) continue;
        for (pii x : v[atual]) {
            int vizinho = x.first, peso = x.second;
            if (dist[x].top() > dist_atual + peso) {
                dist[x].pop();
                dist[x].push(dist_atual + peso);
                fila.push({dist_atual + peso, x});
            }
        }
    }
}
```

Código 1.2
Dijkstra modificado

Assim, a cada iteração do algoritmo teremos os k menores caminhos para todo vértice alcançado, e no final, teremos os k menores caminhos até n. Vale ressaltar que, quando o número de caminhos for menor que k, manteremos a constante INF na priority queue para denotar caminhos de comprimento infinito. No final, cada vértice é processado no máximo k vezes, de modo que temos uma complexidade de tempo de $O(nk \cdot log(nk))$.

Exercises

2.1) [Codeforces Gym - 100625D]

Você tem um grafo não direcionado com n vértices e m arestas com peso. Um ladrão começa no vértice s e está caminhando para um de t possíveis destinos $d_1, d_2 ... d_t$. O ladrão não desperdiça tempo, então sabemos que ele sempre anda por um caminho mínimo entre s e seu destino. Você está tentando descobrir o destino do ladrão, e você descobriu que ele passou pela aresta entre g e h. Você precisa

responder quais são todos os destinos para o qual o ladrão pode estar indo.

$n \leq 2000$, $m \leq 50000$, $1 \leq s \leq n$, $1 \leq t \leq 100$.

2.2) [CSES - Investigation - modificado]

Você tem um grafo direcionado com n vértices com m arestas com peso. Você quer ir do vértice 1 ao n por um caminho cuja soma dos pesos das arestas seja mínimo. Quantos caminhos possíveis existem? Qual o maior número de arestas em um caminho mínimo?

$1 \leq n \leq 100000$, $1 \leq m \leq 200000$.

2.3) [Codeforces - 1272E]

Você tem um vetor a com n inteiros. Em uma rodada, você pode ir da posição i para $i + a_i$ se $i + a_i \leq n$, ou ir de $i - a_i$ se $i - a_i \geq 1$.

Você precisa responder para todo i qual o menor número de rodadas para alcançar uma posição j tal que a paridade de a_i e a_j são opostas.

$n \leq 200000$.

2.4) [POI 2012 - ODL Distance]

Em uma operação você pode pegar um número x e multiplicá-lo por um primo qualquer ou dividí-lo por um primo (x deve ser divisível por esse primo). Consideramos a distância entre dois números a e $bd(a, b)$ como o menor número de operações para que $a = b$.

Você tem um vetor a com n inteiros. Para cada $1 \leq i \leq n$ imprima o $j \neq i$ tal que $d(i, j)$ é mínima.

$n \leq 100000$, $1 \leq a_i \leq 1000000$.

2.5) [Codeforces - 1307F]

Você tem uma árvore de n arestas sem peso. Você quer fazer v caminhos, com o i-ésimo começando em a_i e terminando em b_i. O problema é que você consegue andar no máximo k arestas consecutivas sem parar pra descansar, e só r vértices específicos possuem casas de descanso.

Ao andar de a_i para b_i, você pode repetir vértices e arestas, só não pode andar por mais de k arestas consecutivas sem descansar. Responsa para cada caminho se é possível completá-lo.

$2 \leq n \leq 200000$, $1 \leq k, r \leq n$, $1 \leq v \leq 200000$.

2.6) [Codeforces - 59E]

Você tem um grafo com n vértices e m arestas não direcionadas com peso. Você também tem k triplas a_i, b_i, c_i proibidas, ou seja, você não pode passar por a_i, b_i e c_i um seguido do outro.

As triplas são ordenadas, então passar por a_i, c_i e b_i é permitido. Você precisa encontrar o caminho mínimo de 1 até n.

$2 \leq n \leq 3000$, $m \leq 20000$, $0 \leq k \leq 100000$.

2 Combinatória e Probabilidade

Uma das partes mais relevantes da matemática em Programação Competitiva é Análise Combinatória e Probabilidade. Tema esse que é integrado inclusive com Teoria dos Grafos e Programação Dinâmica, de modo que é extremamente relevante ter uma base forte teórica, e também saber como implementar os fundamentos.

2.1 Fundamentos de combinatória

Nessa seção vamos abordar os princípios de combinatória através de problemas interessantes.

Exercício

(Codeforces - 1536E) Você tem os inteiros n, m e um grid n por m com duas propriedades:
1. Para toda célula adjacente, a diferença absoluta entre os números dessas células é no máximo 1.
2. Se o número de uma célula for maior que 0, ele deve ser estritamente maior que o número de alguma célula adjacente.

Considere que você não consegue enxergar esse grid por completo. Você apenas enxerga cada célula como um 0 ou um #. Se você enxerga uma célula como 0, então o número dessa célula é 0, agora se você enxerga como #, então o número pode ser qualquer inteiro não negativo (inclusive 0).
Determine quantos grids existem que satisfazem essas condições. Imprima sua resposta módulo $10^9 + 7$.

Considere o grid da figura 2.1, temos a opção de substituir o # por um 0 ou por um 1, de modo que a resposta seria 2. Qualquer número maior que 1 não satisfaria a primeira propriedade.

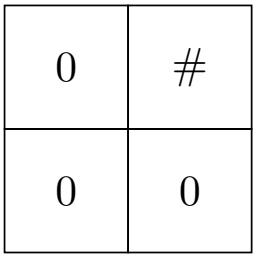

Figure 2.1
Exemplo de grid 2 x 2.

Vamos primeiro tentar resolver uma versão mais simples do problema, em que os # não podem virar 0, apenas números positivos. Nesse caso, sempre existiria um único grid que satisfaz as duas propriedades.

Para provar isso, mostramos que, para qualquer célula cujo valor é maior que 0, o valor dessa célula deve ser a distância ao 0 mais próximo. Como sabemos que toda célula maior que 0 precisa de um vizinho de valor menor, para toda célula existe um caminho que vai dela até o 0 mais próximo e que sempre diminui de valor, o que significa que o valor de cada célula deve ser sua distância ao 0 mais próximo, visto que o caminho precisa acabar.

Provamos que, se fixarmos um conjunto de 0s, existe apenas uma resposta. Voltando ao problema original, cada # pode ou não ser fixado como 0, ou seja, se temos x #s, temos 2^x possíveis conjuntos de 0s, e cada conjunto produz apenas um grid válido, de forma que a resposta para o problema é 2^x.

Devemos apenas nos atentar para o caso em que todas as células do grid são #, nesse caso a resposta é $2^x - 1$, pois precisamos de pelo menos um 0 no tabuleiro para a configuração ser válida.

0	#	#
#	#	#
#	0	#

Figure 2.2
Exemplo de grid 3 x 3.

0	1	2
1	1	2
1	0	1

Figure 2.3
Grid 3 x 3 depois da BFS Multisource.

Note que, fixado um conjunto de 0s, o valor das outras células pode ser encontrada usando uma BFS multisource, como vimos no capítulo de caminhos mínimos. A figura 2.2 e 2.3 mostram um exemplo disso.

Exercício

(CSES - Distributing Apples) Você tem m maçãs para distribuir para n crianças. As maçãs são indistinguíveis entre si.
Quantas formas existem de fazer essa distribuição? Imprima sua resposta módulo $10^9 + 7$.
Por exemplo, para $n = 3$ e $m = 2$, existem 6 formas: $(2, 0, 0)$, $(1, 1, 0)$, $(1, 0, 1)$, $(0, 2, 0)$, $(0, 1, 1)$ e $(0, 0, 2)$.

Podemos visualizar cada distribuição pelo método de estrelas e barras. Cada maçã se torna uma estrela e temos $n - 1$ barras que separam as maçãs. Todas as maçãs que vem antes da primeira barra vão para a primeira criança, todas as maçãs entre a primeira e segunda barra vão para a segunda criança, assim sucessivamente até as maçãs que vem depois de todas as barras.

Por exemplo, a distribuição $(2, 0, 0)$ seria equivalente à visualização $**||$, enquanto $(0, 1, 1) = |*|*$.

Assim, temos uma bijeção entre o número de formas de distribuir m maçãs para n crianças e o número de permutações de $m + n - 1$ objetos (m estrelas e $n - 1$ barras) com m itens iguais e $n - 1$ itens iguais. Isso é equivalente a $(m + n - 1)!/(m!(n - 1)!) = \binom{m+n-1}{m}$.

Mas como calcular um n escolhe k em código?

Em problemas de CP, geralmente queremos um n escolhe k módulo algum número primo grande. Ou seja, $\binom{n}{k} = n! \cdot k!^{-1} \cdot (n - k)^{-1}$. Vimos no

capítulo de Aritmética Modular como calcular o inverso modular de um número em $O(logMOD)$, em que MOD seria o primo grande.

Para evitar esse custo logarítmico toda vez que calculamos um n escolhe k, podemos já deixar precalculado tanto os fatoriais quanto o inverso dos fatoriais de 0 até um $MAXN$ em $O(MAXN)$.

```
int mul(ll a, ll b) {
    return (a * b) % MOD;
}

void precalc() {
    fat[0] = 1;
    for (ll i = 1; i < MAXN; i++)
        fat[i] = mul(i, fat[i - 1]);
    inv[MAXN - 1] = modpow(fat[MAXN - 1], MOD - 2);
    for (ll i = MAXN - 2; i >= 0; i--)
        inv[i] = mul(i + 1, inv[i + 1]);
}
```

Código 2.1
Funções mul e precalc para problemas de combinatória.

No código 2.1, primeiro definimos a função mul, que torna o código mais limpo e ajuda a evitar overflow quando as fórmulas ficam mais complexas.

Depois disso, calculamos todos os fatoriais em tempo linear. Para calcular o inverso fatorial, primeiro calculamos o último elemento do vetor usando exponenciação rápida, e depois calculamos o resto de trás pra frente também em tempo linear. Com isso, o código do n escolhe k fica bem simples, apenas fazemos um if no começo para evitar acessar posições inválidas, e depois consultamos os valores precalculados, como podemos ver no código 2.2.

```
ll nck(int n, int k) {
    if (k > n || k < 0 || n < 0) return 0;
    return mul(fat[n], mul(inv[k], inv[n-k]));
}
```

Código 2.2
Função que retorna n escolhe k e trata overflow e casos de borda.

Exercício

(CSES - Christmas Party) Você e sua família estão organizando um amigo secreto para n pessoas.
Cada uma dessas n pessoas vai pegar um amigo secreto que não é ela mesma.
Quantas distribuições possíveis existem?

Podemos modelar cada distribuição de amigos secretos como uma permutação. Cada pessoa tem um número de 1 a n. Se a pessoa de número i pegou a pessoa de número j, então o número na i-ésima posição da permutação será j.

A única restrição que temos sobre essas permutações é que uma pessoa não pode pegar ela mesma, ou seja, qualquer permutação em que $p(i) = i$ para algum i é inválida. Essas permutações são chamadas de caóticas e o problema basicamente pergunta quantas permutações caóticas de tamanho n existem.

Para calcular o número de permutações caóticas, podemos recorrer ao princípio da inclusão-exclusão.

No caso de dois conjuntos X e Y, o princípio nos diz que

$$|X \cup Y| = |X| + |Y| - |X \cap Y| \tag{2.1}$$

Isso faz sentido intuitivamente, já que $|X \cup Y| = |X - Y| + |Y - X| + |X \cap Y|$, e $|X| = |X - Y| + |X \cap Y|$ e $|Y| = |Y - X| + |X \cap Y|$. Ou seja, ao somar $|X|$ e $|Y|$, estamos somando a interseção de X e Y duas vezes, então precisamos tirar uma para não ter duplicatas.

Já para três conjuntos X, Y e Z temos a fórmula

$$|X \cup Y \cup Z| = |X| + |Y| + |Z| - |X \cap Y| - |X \cap Z| - |Y \cap Z| + |X \cap Y \cap Z| \tag{2.2}$$

Note que a interseção de todos os possíveis subconjuntos estão presentes no cálculo, no entanto, somamos os que tem uma quantidade ímpar de conjuntos e subtraímos os que tem uma quantidade par.

Essa ideia de somar ou subtrair baseada na paridade é o princípio generalizado de inclusão-exclusão, e funciona para encontrar o tamanho da interseção de qualquer número n de conjuntos.

A ideia por trás é análoga a que vimos para dois conjuntos. Somando apenas os conjuntos individuais acabamos somando duas vezes os pares de conjuntos, então precisamos subtrair uma vez cada par. Mas, ao sub-

trair uma vez cada par, acabamos zerando cada trio, então adicionamos de volta cada trio, o que nos faz subtrair cada quarteto e assim sucessivamente.

Para modelar o número de permutações caóticas com o princípio da inclusão-exclusão podemos pensar no conjunto P_i como o conjunto das permutações que tem $p(i) = i$, ou seja, P_1 é o conjunto de todas as permutações que começam com o número 1. O número de permutações caóticas é então

$$n! - \bigcup_{i=1}^{n} P_i \tag{2.3}$$

Usando a fórmula da inclusão-exclusão, sabemos que para calcular o tamanho da união de conjuntos devemos calcular todas as interseções possíveis desses conjuntos e então somar ou subtrair o tamanho dessas interseções baseado na paridade da quantidade de conjuntos que participam de cada uma.

Chamaremos de C_j o somatório do tamanho de todas as interseções de j conjuntos. Pela fórmula da inclusão-exclusão, temos que

$$\bigcup_{i=1}^{n} P_i = \sum_{j=1}^{n} (-1)^{j+1} C_j \tag{2.4}$$

Basta conseguir calcular C_j. Primeiro precisamos escolher j conjuntos P dentre os n vamos usar. Para saber quantas permutações estão nessa interseção, sabemos que j elementos dessas permutações estarão fixos, e podemos permutar apenas os elementos restantes, de modo que

$$C_j = \binom{n}{j} (n-j)! = n!/j! \tag{2.5}$$

Com isso, o número de permutações caóticas de n elementos é

$$n! - \sum_{j=1}^{n} (-1)^{j+1} n!/j! \tag{2.6}$$

2.2 Aplicações de combinatória

Exercício

(Codeforces - 9D) Você quer saber quantas árvores binárias diferentes com n nós e altura pelo menos h existem.
Nesse caso altura é definida como o maior número de vértices no caminho de uma folha até a raiz, incluindo a folha e a raiz.
Uma árvore binária é uma árvore em que cada nó tem no máximo dois filhos, um para a direita e outro para a esquerda.
É garantido que a resposta não passa de $9 \cdot 10^{18}$.

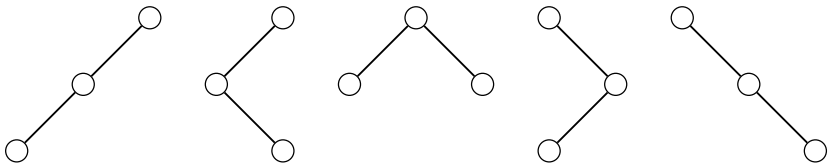

Figure 2.4
As 5 árvores binárias com 3 nós e altura mínima 2.

Para $n = 3$ e $h = 2$, temos as 5 árvores da figura 2.4.

Podemos pensar em cada árvore como sendo a raiz, com uma árvore (potencialmente vazia) para a esquerda e outra árvore (potencialmente vazia) para a direita.

Definindo $dp(i, j)$ como o número de árvores com i nós e altura exatamente j, temos que a resposta para o problema seria

$$\sum_{i=h}^{n} dp(n, i) \tag{2.7}$$

Basta calcularmos os valores de dp. O caso base é

$$dp(0, 0) = dp(1, 1) = 1 \tag{2.8}$$

Esses são os casos da árvore vazia e da árvore com apenas um nó.

Para o caso geral, temos o código 2.3. Nele, i é a quantidade de nós da árvore atual, a é a quantidade de nós na árvore da esquerda, b da árvore da direita, ha e hb são suas respectivas alturas.

Note que não usamos nenhum módulo nessa dp pois o enunciado garante que a resposta não passa de $9 \cdot 10^{18}$.

```
dp[0][0] = 1;
dp[1][1] = 1;
for (int i = 2; i <= n; i++) {
    for (int a = 0; a < n; a++) {
        for (int ha = 0; ha <= a; ha++) {
            int b = i - a - 1;
            for (int hb = 0; hb <= b; hb++) {
                dp[i][max(ha, hb) + 1] += dp[a][ha] * dp[b][hb];
            }
        }
    }
}
ll ans = 0;
for (int i = h; i <= n; i++) ans += dp[n][i];
```

Código 2.3
Algoritmo da dp para cálculo da quantidade de árvores binárias com n vértices e altura mínima h.

Essa ideia de um elemento ser composto por duas partes menores de mesma estrutura aparece também no número de Catalan, usado para contar a quantidade de árvores binárias no geral, ou a quantidade de sequências de parênteses válidas, por exemplo.

Exercício

(CSES - Counting Sequences) Você quer contar a quantidade de sequências v de n elementos em que cada elemento é um inteiro entre 1 e k e cada inteiro entre 1 e k aparece pelo menos uma vez.

Por exemplo, para $n = 3$ e $k = 2$, existem 6 sequências: $(1, 1, 2)$, $(1, 2, 1)$, $(2, 1, 1)$, $(1, 2, 2)$, $(2, 1, 2)$ e $(2, 2, 1)$.

Esse problema é equivalente a distribuir n objetos distintos em k caixas distintas não vazias, considere $v[i] = j$ como dizer que o i-ésimo elemento foi pra j-ésima caixa. As caixas não podem estar vazias pois cada inteiro precisa aparecer pelo menos uma vez. A quantidade de formas de distribuir n objetos distintos em k caixas distintas, permitindo caixas vazias é simplesmente k^n.

Seja A_i o número de distribuições tal que a i-ésima caixa está vazia e as outras caixas não importam. Isso é equivalente a distribuir os n objetos nas $k - 1$ caixas restantes, com as caixas restantes podendo estar vazias, ou seja, $(k - 1)^n$.

Agora, notamos que $|A_1| = |A_2| = ... = |A_k|$, já que não importa qual caixa está vazia (sempre distribuimos entre as $k - 1$ restantes).

Sabemos que existem $\binom{k}{1}A_i$ possíveis, de modo que

$$\sum_{i=1}^{k} |A_i| = \binom{k}{1}(k-1)^n \tag{2.9}$$

Usando um argumento parecido

$$|A_i \cap A_j| = (k-2)^n \tag{2.10}$$

Pois envolve a distribuição entre as $k-2$ caixas restantes. Como existem $\binom{k}{2}$ pares possíveis, temos que

$$\sum_{1 \le i < j \le k}^{k} |A_i \cap A_j| = \binom{k}{2}(k-2)^n \tag{2.11}$$

E para o caso geral com l caixas vazias

$$\sum_{1 \le i < j < \ldots l \le k}^{k} |A_i \cap A_j \ldots \cap A_l| = \binom{k}{l}(k-l)^n \tag{2.12}$$

Usando inclusão-exclusão, chegamos em

$$\sum_{l=0}^{k} (-1)^l \binom{k}{l}(k-l)^n \tag{2.13}$$

Se os recipientes fossem idênticos entre si, então o problema seria equivalente à encontrar o número de Stirling do segundo tipo $S(n,k)$, que conta o número de partições de n objetos distintos em k subconjuntos não vazios idênticos. Mas como os recipientes são distintos podemos pegar $S(n,k)$ e multiplicar por $k!$.

Assim, essa solução nos fornece uma fórmula linear para calcular $S(n,k)$

$$S(n,k) = \sum_{l=0}^{k} (-1)^l \frac{\binom{k}{l}(k-l)^n}{k!} \tag{2.14}$$

Também é interessante saber que $S(n,k)$ pode ser calculado por meio da forma recursiva $S(n+1,k) = k \cdot S(n,k) + S(n,k-1)$, para $0 < k < n$. Para provar essa recorrência, observe que na partição de $n+1$ objetos em k recipientes não vazios ou o $(n+1)$-ésimo objeto vai estar sozinho ou não.

Se ele estiver sozinho, existem $S(n,k-1)$ formas de distribuir o resto dos objetos. Agora, se ele não estiver sozinho, existem $S(n,k)$ formas de distribuir o resto e k opções para o último objeto.

Exercício

Quantas permutações de n elementos com k ciclos existem?

O que seria um ciclo na permutação? Podemos visualizar uma permutação como um grafo com n nós em que cada nó tem grau de entrada 1 e grau de saída 1. Se $p(i) = j$ temos uma aresta que sai de i e vai para j, o que explica o grau de saída 1. Já o grau de entrada é 1 pois cada número na permutação aparece exatamente uma vez.

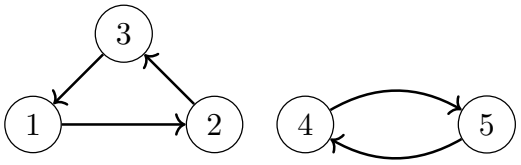

Figure 2.5
Grafo da permutação $(2, 3, 1, 5, 4)$.

Com essas características, notamos que o grafo da permutação é apenas uma coleção de ciclos, como podemos ver na figura 2.5.

O número de permutações com n nós e k ciclos é exatamente o que o número de Stirling do primeiro tipo $s(n, k)$ calcula e pode ser encontrado com uma recorrência parecida com a do segundo tipo.

$$s(n + 1, k) = n \cdot s(n, k) + s(n, k - 1) \tag{2.15}$$

Note que a única diferença é que multiplicamos o primeiro termo por n ao invés de multiplicar por k.

Para provar essa recorrência, pensamos em formar uma permutação de $n + 1$ elementos a partir de uma permutação de n elementos. Podemos formar um ciclo unitário novo com o último elemento, de modo que existem $s(n, k - 1)$ formas de distribuir o resto.

A outra opção é inserir o elemento novo em algum ciclo existente. Podemos inserir ele depois de cada um dos n elementos existentes no ciclo, que é o $n \cdot s(n, k)$ da fórmula 2.16.

Exercício

(CSES - Functional Graph Distribution) Um grafo funcional é um grafo direcionado em que cada nó tem grau de saída igual a um.
Você quer saber quantos grafos funcionais de n nós e k componentes conexas $g_n(k)$ existem para todo k entre 1 e n.

O grafo é chamado de funcional pois cada nó mapeia para exatamente outro nó, da mesma forma como uma função. Um grafo funcional pode ser modelado como uma sequência a de n elementos em que cada elemento da sequência é um inteiro entre 1 e n. Dessa forma, se $a(i) = j$, significa que a única aresta que sai do nó i vai para o nó j. Também significa que existem n^n grafos funcionais com n vértices, pois cada vértice i tem exatamente uma aresta que pode apontar para qualquer um dos n vértices (inclusive apontar para si mesmo).

Esse problema aparenta ser uma generalização do problema anterior, com a diferença de que agora o grau de entrada pode ser diferente de 1.

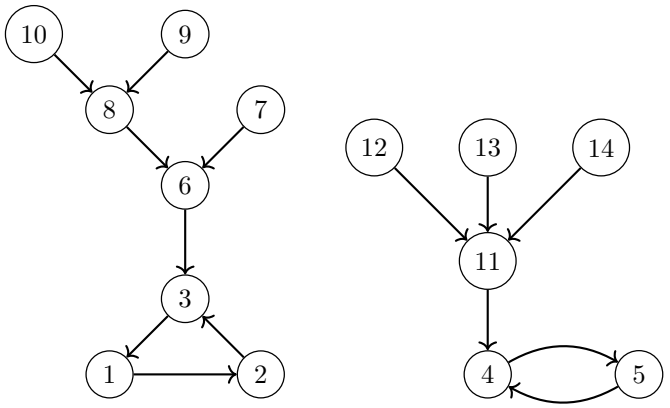

Figure 2.6
Exemplo de grafo funcional.

Se olharmos para o grafo funcional da figura 2.6, veremos que ele é formado por alguns ciclos e algumas árvores cujas raízes são os nós dos ciclos. Por conta das propriedades de grau dos vértices, essa é a estrutura de todo grafo funcional: cada componente tem um ciclo central com árvores penduradas nos nós do ciclo.

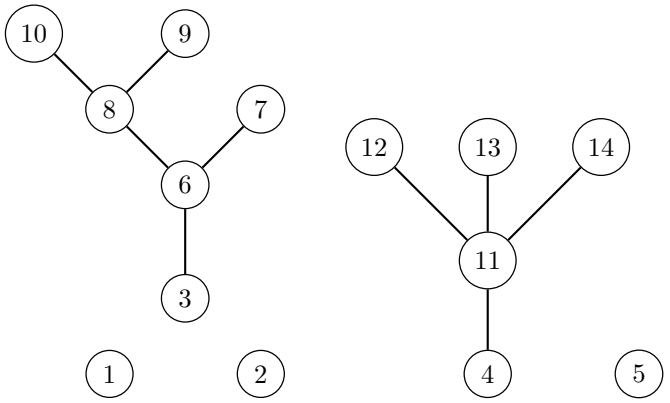

Figure 2.7
Floresta obtida a partir do grafo funcional da figura 2.6.

Podemos pensar em todo grafo funcional como a combinação de uma permutação (os nós que estão nos diferentes ciclos), com uma floresta. No caso do grafo da figura 2.6, ele é formado pela junção da permutação de 5 nós e 2 componentes do grafo da figura 2.5 com a floresta de 14 nós e 5 componentes da figura 2.7, que é o grafo funcional original porém com as arestas não direcionadas e apagando as arestas dos ciclos.

Com essa ideia, temos uma forma de resolver o problema em $O(n^2)$. Primeiro calculamos todos os valores de $s(n,k)$ em $O(n^2)$ usando a recorrência do problema anterior. Depois, temos que

$$g_n(k) = \sum_{i=k}^{n} \binom{n}{i} s(i,k) F(n,i) \tag{2.16}$$

Em que $F(n,i)$ é a quantidade de florestas com n vértices e i árvores. Basicamente nessa soma estamos escolhendo os i vértices que estarão nos ciclos $\binom{n}{i}$, então formamos k ciclos distintos com esses vértices, que é calculado por $s(i,k)$. Com os vértices que sobraram formamos florestas com i raízes, os vértices dos ciclos, que é calculado por $F(n,i)$.

Para calcular a quantidade de florestas, podemos usar a fórmula de Cayley, que conta o número de florestas com conjunto de vértices $\{1, 2, ..., n\}$, com s componentes, em que os vértices de 1 a s estão em componentes distintas:

$$F(n,s) = sn^{n-s-1} \tag{2.17}$$

Em particular, para $s = 1$, temos que o número de árvores com n vértices rotulados é n^{n-2}. Vamos apresentar agora uma prova combinatória dessa fórmula demonstrada por Takács (1990), baseada na seguinte recorrência para $n > 1$ e $1 \leq s \leq n$:

$$F(n, s) = \sum_{j=0}^{n-s} \binom{n-s}{j} F(n-1, s+j-1) \tag{2.18}$$

Com o caso base $F(1, 1) = 1$ e $F(n, 0) = 0$ para $n \geq 1$. Com esse somatório estamos iterando pelo grau do vértice 1. O vértice 1 pode estar conectado com $j = 0, 1, ..., n-s$ vértices distintos. Existem $\binom{n-s}{j}$ formas de escolher esses vértices. Se retirarmos o vértice 1 e todas suas arestas do grafo, ficamos com um grafo de $n-1$ vértices, $s+j-1$ componentes em que os vértices $2, 3, ..., s$ e os j filhos de 1 estão todos em componentes distintas, que é contado por $F(n-1, s+j-1)$. Isso prova a equação 2.18.

Para provar a equação 2.17, vamos usar indução. O caso base $n = 1$ é válido pois $F(1, 1) = 1 = sn^{n-s-1} = 1 \cdot 1^0$. Vamos supor que $F(n-1, i) = i(n-1)^{n-i-2}$ para $1 \leq i \leq n-1$ e $n > 1$, então, substituindo na equação 2.18, chegamos em:

$$F(n, s) = \sum_{j=0}^{n-s} \binom{n-s}{j} (s+j-1)(n-1)^{n-s-j-1} \tag{2.19}$$

Separando em dois somatórios:

$$F(n, s) = (s-1) \sum_{j=0}^{n-s} \binom{n-s}{j} (n-1)^{n-s-j-1} + \sum_{j=0}^{n-s} \binom{n-s}{j} j(n-1)^{n-s-j-1} \tag{2.20}$$

Substituindo $j\binom{n-s}{j}$ por $(n-s)\binom{n-s-1}{j-1}$ para $j \geq 1$, chegamos em:

$$F(n, s) = (s-1) \sum_{j=0}^{n-s} \binom{n-s}{j} (n-1)^{n-s-j-1} + (n-s) \sum_{j=1}^{n-s} \binom{n-s-1}{j-1} (n-1)^{n-s-j-1} \tag{2.21}$$

Reescrevendo o segundo somatório com $k = j - 1$:

$$F(n, s) = (s-1) \sum_{j=0}^{n-s} \binom{n-s}{j} (n-1)^{n-s-j-1} + (n-s) \sum_{k=0}^{n-s-1} \binom{n-s-1}{k} (n-1)^{n-s-k-2} \tag{2.22}$$

Colocando um $(n-1)$ extra para dentro de cada somatório:

$$F(n,s) = \frac{s-1}{n-1}\sum_{j=0}^{n-s}\binom{n-s}{j}(n-1)^{n-s-j} + \frac{n-s}{n-1}\sum_{k=0}^{n-s-1}\binom{n-s-1}{k}(n-1)^{n-s-k-1} \tag{2.23}$$

Aplicando o binômio de Newton:

$$F(n,s) = \frac{s-1}{n-1}n^{n-s} + \frac{n-s}{n-1}n^{n-s-1} = \frac{ns-s}{n-1}n^{n-s-1} = sn^{n-s-1} \tag{2.24}$$

Com isso, completamos a prova da equação 2.17.

2.3 Probabilidade

Vários problemas de probabilidade no fundo se tornam problemas de combinatória, pois acabamos resolvendo eles contando a quantidade de eventos desejados e dividindo pela quantidade de eventos totais. Nessa seção vamos focar em problemas que introduzem conceitos específicos para Probabilidade.

Exercício

(CSES - Dice Probability) Você rola um dado n vezes, e cada vez um número entre 1 e 6 sai com igual probabilidade.
Qual a probabilidade da soma dos valores dos dados estar entre a e b? Sua resposta deve estar arredondada para as primeiras 6 casas decimais.

A primeira coisa a se notar nesse problema é que ele não pede a probabilidade módulo algum primo grande, o que significa que iremos usar aritmética de ponto flutuante, e que provavelmente não vamos calcular a quantidade de eventos desejados e dividir pela quantidade de possibilidades totais. Isso porque esses dois valores são enormes, a quantidade de possibilidades totais é 6^n, o que rapidamente ultrapassa o limite de um long long integer em C++.

Uma ideia melhor é guardar a probabilidade da soma dos valores conforme você vai rolando os dados. Chamaremos de $p_i(j)$ a probabilidade da soma dos valores dos primeiros i dados ser exatamente j. O caso base é $p_0(0) = 1$, e as transições são

$$p_i(j) = \sum_{k=j-6}^{j-1}\frac{p_{i-1}(k)}{6} \tag{2.25}$$

A probabilidade da soma estar entre a e b se torna

$$\sum_{i=a}^{b} p_n(i) \tag{2.26}$$

Usando programação dinâmica, facilmente implementamos essa solução.

Exercício

(CSES - Moving Robots) Você tem um tabuleiro de xadrez 8x8, com um robô em cada posição. Todo robô vai se mover de forma independente r vezes. Em cada rodada, um robô pode se mover para cima, baixo, esquerda ou direita, e ele escolhe de forma equiprovável entre os movimentos que ele pode realizar sem sair do tabuleiro.
Os robôs são completamente independentes, ou seja, dois robôs podem ocupar a mesma casa e não interagem entre si.
Qual o valor esperado de casas vazias após essas r rodadas de movimentação?

Esse problema introduz o conceito de valor esperado (ou esperança) para uma variável aleatória X. Nesse caso, X é a quantidade de casas vazias no tabuleiro de xadrez após os robôs de movimentarem r vezes. O valor esperado de X é:

$$E[X] = \sum_{x} x \cdot P(x) \tag{2.27}$$

Em que x passa por todos os valores possíveis de X. Ou seja, o valor esperado de uma variável aleatória X é a média ponderada dos seus possíveis valores, em que o peso é a probabilidade de X assumir aquele valor. Por exemplo, ao rolar um dado, o valor esperado é:

$$1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5 \tag{2.28}$$

Uma propriedade crucial do valor esperado é a linearidade, que diz que

$$E[X_1 + X_2 + ... + X_n] = E[X_1] + E[X_2] + ... + E[X_n] \tag{2.29}$$

E isso vale mesmo quando as variáveis aleatórias dependem uma da outra.

No nosso problema, queremos o valor esperado de casas vazias. Se numerarmos as casas de 1 até 64, podemos pensar que queremos $E[C_1 +$

$C_2 + ... + C_{64}]$, em que C_i é a variável aleatória da casa i, que pode estar vazia ou não. Usando linearidade da esperança, queremos $E[C_1]+E[C_2]+...E[C_{64}]$. Nesse caso

$$E[C_i] = 1 \cdot p_i + 0 \cdot (1 - p_i) = p_i \tag{2.30}$$

Em que p_i é a probabilidade da i-ésima casa estar vazia. Ou seja, para resolvermos o problema, basta saber a probabilidade de cada casa individual estar sozinha.

Podemos usar uma abordagem parecida à do problema dos dados. Para cada robô i, mantemos a probabilidade $p_{i,j}(k)$ dele estar na casa k após j rodadas. O caso base é $p_{i,0}(i) = 1$ para todo i, e as transições são

$$p_{i,j}(k) = \frac{1}{|V_k|} \sum_{v \in V_k} p_{i,j-1}(v) \tag{2.31}$$

Em que V_k são todos os vizinhos de k, e é um conjunto que muda de tamanho, dependendo se a casa está no meio, na borda ou nos cantos do tabuleiro. Ou seja, a probabilidade do i-ésimo robô estar na posição k após j rodadas é igual ao somatório das probabilidades do i-ésimo robô estar em um dos vizinhos de k após a $j-1$ rodadas, multiplicadas pela probabilidade do i-ésimo robô escolher a casa k como próxima casa $\frac{1}{viz(v)}$.

Com isso, a probabilidade da k-ésima casa estar vazia após as r rodadas é

$$\prod_{i=1}^{64} 1 - p_{i,r}(k) \tag{2.32}$$

Ou seja, é o produtório da probabilidade de cada um dos robôs não estar na casa k.

Exercício

(Autoral) Você tem uma árvore com n nós. Um dos caminhos dessa árvore vai ser escolhido aleatoriamente com probabilidade uniforme.
Qual o valor esperado do tamanho do caminho escolhido?
Tamanho nesse caso é a quantidade de arestas no caminho.

Como só existe um único caminho do nó a para o nó b para todo a e b em uma árvore, sabemos que existem exatamente $n \cdot (n-1)$ caminhos

distintos em uma árvore de n nós. Usando a fórmula (2.27), o valor esperado do tamanho do caminho escolhido seria

$$\sum_{c} \frac{len(c)}{n \cdot (n-1)} \tag{2.33}$$

Em que c passa por todos os caminhos da árvore e $len(c)$ é a quantidade de arestas do caminho c.

Para resolver esse problema em $O(n^2)$, podemos rodar uma BFS saindo de cada nó, e no final da BFS saindo do nó i, as distâncias de i para os outros nós é igual aos tamanhos de todos os caminhos que começam em i. No entanto, podemos desenvolver uma solução mais rápida. Como a probabilidade de cada caminho ser escolhido é igual, podemos notar que a única coisa que precisamos calcular é a soma do tamanho de todos os caminhos da árvore, e no final dividir por $n \cdot (n-1)$.

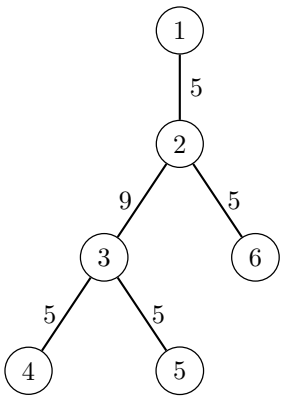

Figure 2.8
Árvore em que as arestas estão marcadas com o número de caminhos que cada uma participa.

Isso é equivalente a contar de quantos caminhos cada aresta participa, e então somar esses valores. Uma aresta participa de todos os caminhos que começam na componente de uma de suas extremidades e termina na componente da sua outra extremidade. Se olharmos para a figura 2.8, veremos que a soma do tamanho de todos os caminhos é $5+5+5+5+9 = 29$. Por exemplo, a aresta $(2, 3)$ participa de 9 caminhos, todos aqueles que começam em um vértice do conjunto $\{3, 4, 5\}$ e terminam em um vértice do conjunto $\{1, 2, 6\}$.

Rodando uma DFS para calcular o tamanho de cada subárvore, resolvemos o problema em $O(n)$.

Exercício

(Autoral) Você tem uma moeda viciada em que a chance de cair cara é $\frac{1}{5}$ e coroa é $\frac{4}{5}$.
Qual o valor esperado $E[J]$ de moedas que você vai ter que jogar até conseguir uma cara?

Podemos novamente aplicar a fórmula (2.27) do valor esperado. Temos $\frac{1}{5}$ de chance de conseguir cara com 1 jogada. Para duas jogadas, temos $\frac{4}{5} \cdot \frac{1}{5}$ de conseguir cara na segunda rodada (primeiro rodamos uma coroa e em seguida uma cara). Já a chance de conseguir a primeira cara na i-ésima rodada é

$$\left(\frac{4}{5}\right)^{i-1} \cdot \frac{1}{5} \tag{2.34}$$

Que seria primeiro rodar $i-1$ coroas e em seguida uma cara. Colocando isso na fórmula:

$$E[J] = \sum_{i=1}^{\infty} i \left(\frac{4}{5}\right)^{i-1} \frac{1}{5} \tag{2.35}$$

Esse somatório até pode ser resolvido usando progressão aritmético-geométricas, mas existe uma forma mais poderosa de calcular o valor esperado. Podemos usar a seguinte fórmula:

$$E[J] = 1 + \frac{4}{5} E[J] \tag{2.36}$$

Ou seja, jogamos a moeda uma vez (por isso o 1), e existe uma chance de $\frac{4}{5}$ de sair uma coroa e termos que continuar jogando a moeda (por isso o $E[J]$). Essa fórmula é cíclica, pois o valor esperado de moedas até sair uma cara é o mesmo, independente de quantas coroas já tiramos. Resolvendo a equação (2.36), achamos que $E[J] = 5$.

Exercício

(Autoral) Você tem uma moeda viciada em que a chance de cair cara é $\frac{1}{5}$ e coroa é $\frac{4}{5}$.
Qual o valor esperado $E[J_2]$ de moedas que você vai ter que jogar até conseguir duas caras consecutivas?

Podemos expandir a abordagem anterior para chegar na seguinte fórmula:

$$E[J_2] = \frac{1}{25} \cdot 2 + \frac{4}{25} \cdot (2 + E[J_2]) + \frac{4}{5} \cdot (1 + E[J_2]) \tag{2.37}$$

Basicamente, temos uma chance de $\frac{1}{25}$ de rolar duas caras seguidas e acabar com o processo (por isso o $\frac{1}{25} \cdot 2$). Também temos uma chance de $\frac{4}{25}$ de rolar uma cara seguida de uma coroa, e voltar à estaca zero ($\frac{4}{25} \cdot (2 + E[J_2])$). Por fim, temos uma chance de $\frac{4}{5}$ de rolar uma coroa direto e retornar à estaca zero ($\frac{4}{5} \cdot (1 + E[J_2])$). Resolvendo a equação, encontramos que $E[J_2] = 30$.

Exercício

(Codeforces - Blog 71097) Você tem n pontos num plano $2D$ e precisa achar uma reta que passe pela maior quantidade de pontos possíveis.
No entanto, os pontos são dados de tal forma que a resposta é no mínimo $\frac{n}{4}$.

Uma abordagem simples para resolver esse problema é, para todo par de pontos, considerar a reta que conecta esse par de pontos, e então checar quantos pontos estão exatamente em cima dessa reta. Essa abordagem roda em $O(n^3)$, pois temos n^2 pares de pontos diferentes e checamos cada uma das n^2 linhas em $O(n)$.

Todavia, o enunciado nos dá uma informação essencial de que a resposta é no mínimo $\frac{n}{4}$. Com isso, ao escolher um ponto aleatório, existe uma chance de pelo menos $\frac{1}{4}$ dele estar na linha da resposta. Se tomarmos dois pontos aleatórios, existe uma chance de pelo menos $\frac{1}{16}$ de que a linha que conecta eles é a linha da resposta.

Isso nos fornece um algoritmo randomizado: escolhemos um par de pontos aleatórios e checamos em $O(n)$ quantos pontos estão na linha que conecta esse par de pontos. Existe uma chance de $\frac{15}{16}$ de não termos

encontrado a resposta. No entanto, se repetirmos esse processo x vezes, teremos um algoritmo que roda em $O(xn)$ cuja probabilidade de não ter encontrado a resposta é $\left(\frac{15}{16}\right)^x$. O ideal é encontrar um valor de x de modo que o algoritmo ainda caiba no limite de tempo mas tenha uma probabilidade desprezável de estar errado. Rodando esse processo 200 vezes por exemplo, a probabilidade de não termos encontrado a resposta é de 0.00024%.

Exercises

2.1) [Autoral]

Existem quantas sequências de n números em que cada número é um inteiro entre 1 e n que são não-decrescentes?

Uma sequência é não-decrescente se todo número é maior ou igual ao número anterior. Por exemplo, $[1, 1, 2, 3, 3, 6]$ é não decrescente, enquanto $[1, 2, 3, 2, 5]$ não é.

$n \leq 10^6$.

2.2) [Codeforces Gym - 105669C1]

Quantas permutações de n vértices têm pelo menos um ciclo de tamanho maior ou igual a T? Imprima a resposta módulo $10^9 + 7$.

$1 \leq T \leq n \leq 10^3$.

2.3) [Codeforces Gym - 105669C2]

Quantas permutações de n vértices têm pelo menos um ciclo de tamanho maior ou igual a T? Imprima a resposta módulo $10^9 + 7$.

Única diferença pro problema anterior são os limites de T e n.

$1 \leq T \leq n \leq 10^6$.

2.3) [Codeforces Gym - 105669E]

Você tem uma mesa circular com n assentos numerados de 1 a n e k pessoas. De quantas formas essas k pessoas conseguem sentar na mesa sem que ninguém esteja do lado de ninguém?

Note que importa qual pessoa está sentada em qual cadeira, e não apenas qual cadeiras estão ocupadas.

$1 \leq k \leq n \leq 10^5$.

2.5) [Codeforces - 1042E]

Vasya tem uma matriz nxm, em que o elemento da i-ésima linha e j-ésima coluna é a_{ij}.

Vasya também tem uma moeda. Essa moeda inicialmente está na linha r e coluna c. Vasya vai fazer o seguinte processo até não dar mais: Dentre todas as posições da matriz cujo elemento tem valor

menor que o elemento da posição da moeda, Vasya escolhe de forma aleatória e equiprovável uma dessas posições.

Ela então move a moeda para essa posição e adiciona o quadrado da distância euclidiana entre as duas posições (a que moeda estava e a que a meoda está) para seu score. O processo acaba quando não existir posições com valor menor do que a que a moeda está.

A distância euclidiana entre um ponto (x_1, y_1) e (x_2, y_2) é $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

O valor esperado pode ser representado como $\frac{P}{Q}$, em que P e Q são inteiros comprimos. Imprima o resultado de $P \cdot Q^{-1}$ módulo 998244353.

$1 \leq n, m \leq 1000$, $0 \leq a_{ij} \leq 10^9$, $1 \leq r \leq n$, $1 \leq c \leq m$.

2.6) [Codeforces - 364D] Você tem uma sequência A de n inteiros $a_1, a_2, ...a_n$. Encontre o maior número g tal que g divide pelo menos metade dos inteiros de A. Note que A pode ter inteiros repetidos.

$1 \leq n \leq 10^6$, $1 \leq a_i \leq 10^{12}$.

3 Árvores

Uma árvore é um grafo conexo de n vértices e $n-1$ arestas sem ciclo. Vários problemas são em cima desse tipo específico de grafo. Nesse capítulo vamos resolver problemas que exploram propriedades específicas dessa classe.

3.1 Small to Large

Exercício

(CSES - Distinct Colors) Você tem uma árvore de n nós em que o nó 1 é a raiz. Cada nó tem uma cor.
Para cada nó, você precisa definir quantas cores distintas existem em sua subárvore.

Uma subárvore de um vértice i é o subgrafo contendo apenas os vértices e arestas abaixo de i. No caso da figura 3.1, a subárvore de 3 contém apenas os vértices 3, 4 e 5, e tem duas cores distintas, vermelho e verde. O conceito de subárvore nesse caso depende de quem é a raiz da árvore.

Uma solução possível para esse problema seria rodar uma DFS e manter em cada nó i um set com todas as cores da subárvore de i.

```
void dfs(int node, int parent) {
  for (int x : v[node]) {
    if (x == parent) continue;
    dfs(x, node);
    for (int y : colors[x]) colors[node].insert(y);
  }
}
```

Código 3.1
DFS calculando as cores de cada subárvore em $O(n^2 log(n))$.

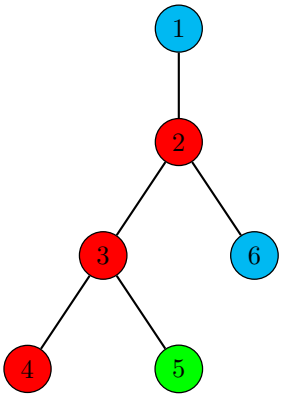

Figure 3.1
Árvore com os vértices coloridos.

No código 3.1, v é a lista de adjacência e *colors* são os sets de cores. No entanto, essa solução roda em $O(n^2 logn)$. O pior caso seria um grafo caminho como o da figura 3.2, em que todos os vértices têm cores diferentes e a cor do último nó passa por n sets diferentes, do penúltimo por $n-1$ sets, e assim sucessivamente.

No entanto, adicionando apenas uma if ao código 3.1, conseguimos uma solução muito mais eficiente.

```
void dfs(int node, int parent) {
  for (int x : v[node]) {
    if (x == parent) continue;
    dfs(x, node);
    if (colors[x].size() > colors[node].size())
        swap(colors[node], colors[x]);
    for (int y : colors[x]) colors[node].insert(y);
  }
  ans[node] = (int)colors[node].size();
}
```

Código 3.2
DFS calculando as cores de cada subárvore em $O(nlog^2(n))$.

Vale ressaltar que em c++, o swap de dois sets é realizado em $O(1)$, pois é realizado apenas a troca entre os ponteiros. Essa solução roda em $O(nlog^2 n)$. Para provar isso, note que estamos sempre passando os elementos de um set menor para um set maior. Ou seja, se o tamanho do set menor era X, o tamanho da união dos dois é pelo menos $2X$. Assim, se um elemento trocou de set k vezes, ele agora está em um set de tamanho pelo menos 2^k vezes maior que seu set original. Como o

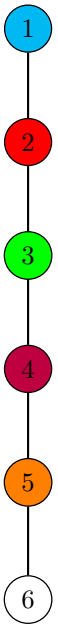

Figure 3.2
Grafo caminho com n cores distintas.

tamanho máximo de um set é n, cada elemento pode ser trocado de set no máximo $logn$ vezes, o que prova a complexidade final $O(nlog^2n)$.

Exercício

(BOI2023 - Car Race - Modificado) Você tem uma árvore de n nós em que alguns nós tem carros e o vetor booleano has, que indica se existe ou não um carro em cada vértice.
No tempo 0, todos os carros começam a caminhar em direção à raiz, andando uma aresta por unidade de tempo. Se dois carros chegarem no mesmo vértice ao mesmo tempo, ambos explodem. Ao chegar na raiz, os carros ficam livres e desaparecem da árvore. Você precisa dizer quais carros vão sobreviver.

Olhando para a figura 3.3, vemos que os carros 2 e 6 sobrevivem, enquanto os carros 3, 4 e 5 explodem ao chegar ao vértice 2. Ademais, percebemos que sobrevive no máximo apenas um vértice por nível.

Podemos fazer algo parecido ao problema anterior, rodar uma DFS e manter alguma estrutura com os carros que sobraram na nossa subár-

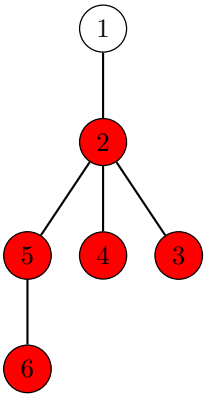

Figure 3.3
Árvore do problema Car Race, os vértices com carros estão pintados de vermelho.

vore. Como temos no máximo um carro vivo por profundidade, usaremos um vetor para cada vértice.

Quando estamos processando o vértice x, pensamos que todos os carros já subiram até x, e queremos guardar quais sobreviveram, separando-os pelo tempo levado para chegar em x, ou seja, profundidade a partir de x.

```
void dfs(int node, int parent) {
    for (int x : v[node]) {
        if (x == parent) continue;
        dfs(x, node);
        if (car[node].size() < car[x].size())
            swap(car[node], car[x]);
        for (int i = 0; i < (int)car[x].size(); i++) {
            if (car[x][i]) {
                if (car[node][i])
                    car[node][i] = -1;
                else
                    car[node][i] = car[x][i];
            }
        }
    }
    for (int &x : car[node]) if (x == -1) x = 0;
    if (has[node])
        car[node].push_front(node);
    else
        car[node].push_front(0);
}
```

Código 3.3
DFS para o problema Car Race usando deque.

No código 3.3, mantemos um deque *car* para cada vértice, em que $car_{i,j}$ indica qual carro j vértices abaixo de i sobreviveu. Primeiro fazemos o merge dos filhos, marcando uma posição como inválida quando temos pelo menos dois carros naquela profundidade. Por último, colocamos o carro do vértice atual no início do deque.

Ao construir o deque do carro 2 por exemplo, fazemos o merge do deque $\{5, 6\}$ com o deque $\{4\}$, o que resulta no deque $\{-1, 6\}$, visto que tínhamos dois carros vivos na mesma profundidade. Depois, fazemos o merge do deque $\{-1, 6\}$ com o deque $\{3\}$, o que resulta no deque $\{-1, 6\}$. Por fim, atualizamos as posições inválidas para vazias e colocamos o carro 2 no início do deque, resultando em $\{2, 0, 6\}$.

O interessante é calcular a complexidade dessa solução. Podemos notar que cada deque tem tamanho igual à altura da subárvore. Se no problema anterior cada vértice trocava de estrutura no máximo $log(n)$ vezes, podemos pensar que aqui, cada vértice troca de estrutura no máximo uma vez, de modo que essa solução é linear!

Isso acontece pois quando fazemos o merge de uma estrutura de tamanho x com outra de tamanho $y < x$, a estrutura resultante tem tamanho x, e não $x + y$. Então podemos pensar que todos os elementos de y trocaram de estrutura uma única vez e então foram destruídos, e restaram apenas os elementos de x.

O código fornecido aqui usa deque por simplicidade, mas vale ressaltar que o ideal é evitar essa estrutura e usar vector sempre que possível, pois tem uma constante muito menor. Na solução para esse problema, nós apenas colocamos elementos no início do vetor. Poderíamos então substituir o deque por um vector e manter os carros invertidos, como mostrado no código 3.4.

```
void dfs(int node, int parent) {
    for (int x : v[node]) {
        if (x == parent) continue;
        dfs(x, node);
        if (car[node].size() < car[x].size())
            swap(car[node], car[x]);
        int tam = car[node].size();
        for (int i = 0; i < (int)car[x].size(); i++) {
            int dep = (int)car[x].size() - i - 1;
            if (car[x][i]) {
                if (car[node][tam - dep - 1])
                    car[node][tam - dep - 1] = -1;
                else
                    car[node][tam - dep - 1] = car[x][i];
            }
        }
    }
```

```
    for (int &x : car[node]) if (x == -1) x = 0;
    if (has[node])
        car[node].push_back(node);
    else
        car[node].push_back(0);
}
```

Código 3.4
DFS para o problema Car Race usando vector.

3.2 Centroid Decomposition

Exercício

(CSES - Finding a Centroid) Você tem uma árvore de n nós. Encontre um centróide dessa árvore.

Dada uma árvore de n nós, um centróide é um vértice cuja remoção quebra a árvore em uma floresta de árvores, tal que nenhuma árvore dessa floresta tem tamanho maior que $\lfloor \frac{n}{2} \rfloor$.

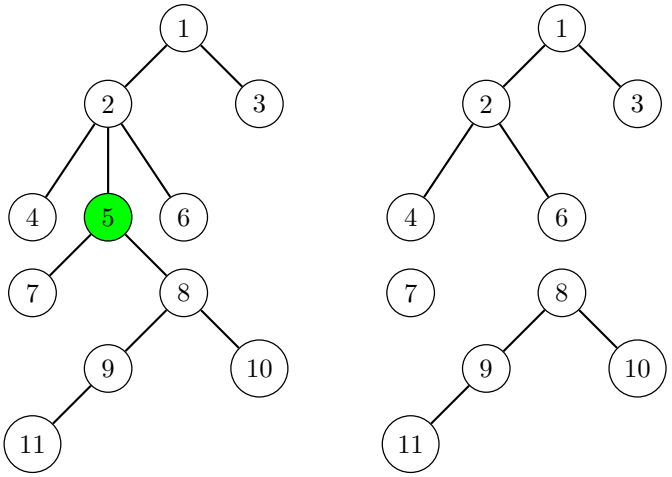

Figure 3.4
Ávore com 11 nós, em que o nó 5 é o centróide.

Olhando para a árvore da figura 3.4, notamos que o nó 5 é um centróide dessa árvore, pois todas as componentes resultantes têm tamanho menor ou igual que $\lfloor \frac{11}{2} \rfloor = 5$.

Agora que entendemos o que é o centróide de uma árvore, vamos provar que um centróide sempre existe.

Primeiro escolhemos um vértice v aleatório da árvore. Se v é um centróide, acabou. Caso contrário, existe exatamente uma componente conexa de tamanho maior que $\lfloor \frac{n}{2} \rfloor$. Consideramos agora o vértice u adjacente à v que pertence a essa componente, e repetimos o mesmo procedimento para u até encontrarmos um centróide. Dessa forma, nunca visitamos o mesmo vértice duas vezes, visto que a componente deixada para trás sempre tem tamanho estritamente menor que $\lfloor \frac{n}{2} \rfloor$. Como o número de vértices é finito, esse procedimento sempre termina e um centróide sempre existe.

Adicionalmente, essa prova nos fornece um algoritmo eficiente para encontrar um centróide: Começamos em uma raiz qualquer da árvore, e rodamos uma dfs para calcular o tamanho de cada subárvore. Depois, começamos novamente da raiz, e, enquanto existir algum filho com uma subárvore de tamanho maior que $\lfloor \frac{n}{2} \rfloor$, movemos para esse filho. Quando não existir, o nó atual será o centróide.

Exercício

(CSES - Network Renovation) Você tem uma árvore de n nós. Você quer adicionar a menor quantidade de arestas possíveis tal que, se qualquer uma das arestas do grafo final for removida, o grafo ainda é conexo. Além de saber a quantidade de arestas necessária, você precisa dizer quais são essas arestas.

Esse problema nos pergunta quais arestas devemos adicionar para que não exista nenhuma ponte no grafo. Uma ponte é uma aresta que, quando apagada, desconecta o grafo.

Uma aresta é uma ponte se e somente se ela não pertence a nenhum ciclo. Ou seja, queremos adicionar a menor quantidade de arestas possível para que toda aresta participe de pelo menos um ciclo.

Em uma árvore, inicialmente todas as arestas são pontes. Quando conectamos dois vértices u e v com uma aresta, criamos um ciclo que envolve todas as arestas no caminho entre u e v, de modo que essas arestas deixam de ser ponte. Diremos que uma aresta entre u e v conserta todas as arestas no caminho entre u e v.

Podemos notar que qualquer aresta adicionada conserta no máximo duas folhas da árvore, em que uma folha é um vértice com apenas uma

aresta. Logo, chamando a quantidade de folhas de f, temos que $\lceil \frac{f}{2} \rceil$ é um limitante inferior para a solução.

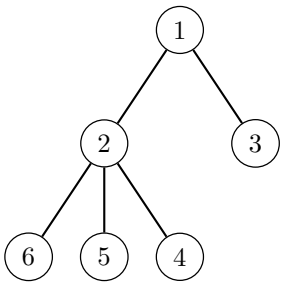

Figure 3.5
Exemplo de árvore para o problema Network Renovation.

Olhando para a árvore da figura 3.5, vemos que ela tem 4 folhas: os vértices 3, 4, 5 e 6. Usando o limitante inferior, provamos que essa árvore precisa de no mínimo duas arestas. Ou seja, se conseguirmos uma solução que adiciona apenas 2 arestas, sabemos que essa solução é ótima.

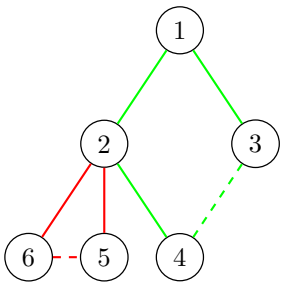

Figure 3.6
Solução para a árvore da figura 3.5.

Na figura 3.6, vemos uma solução de apenas 2 arestas, com as arestas pintadas para indicar os ciclos que cada uma faz parte. Mas esse é apenas um caso específico, será que sempre conseguimos uma solução com $\lceil \frac{f}{2} \rceil$ arestas?

Olhando para a árvore da figura 3.7, vemos que ela tem 6 folhas, e que adicionar 3 arestas entre pares de folhas nem sempre resolve o problema, visto que as arestas $(2, 3)$ e $(2, 4)$ continuam sendo pontes.

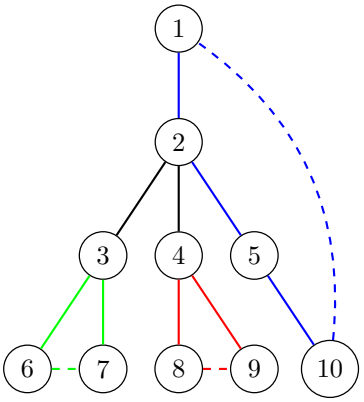

Figure 3.7
Árvore com adição de 3 arestas.

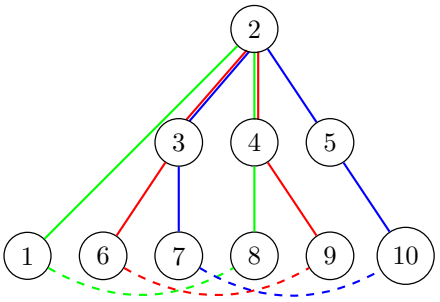

Figure 3.8
Árvore resolvida com adição de 3 arestas.

No entanto, se conseguirmos conectar os pares de folha de tal forma que o caminho entre qualquer par de folhas sempre passa por alguma raiz, então toda aresta a vai ser protegida, pelas folhas que estão abaixo de a, como pode ser visto na figura 3.8, em que todos os caminhos passam pelo vértice 2 (raiz).

Para resolver o problema, podemos enraizar a árvore em qualquer vértice que não seja uma folha, então fazemos uma dfs saindo desse vértice e criamos um vetor v com todas as folhas da esquerda para a direita. No caso da árvore da figura 3.5, v seria $6, 5, 4, 3$ e $f = 4$. Agora, conectamos toda folha i com a folha na posição $i + \lfloor \frac{f}{2} \rfloor$. Ou seja, conectamos 6 com 4 e 5 com 3. Isso sempre funciona, independente da árvore.

A prova é baseada em centróide. Digamos que agora vamos encontrar uma variação do centróide, chamado centróide de folhas, que é um vértice cuja remoção quebra a árvore em uma floresta de árvores, tal que nenhuma tenha mais que $\lfloor \frac{f}{2} \rfloor$ folhas. Pelo mesmo argumento do centróide, esse novo centróide sempre existe, e, se enraizarmos a árvore nele, notamos que todos os caminhos da nossa construção passam por esse centróide de folhas, de modo que nossa construção sempre é válida.

Vale ressaltar que não precisamos encontrar o centróide de folhas e enraizar a árvore nele. A existência do centróide de folhas apenas mostra que, se enraizarmos a árvore em um vértice não folha e seguirmos a solução descrita acima, então todo caminho obrigatoriamente passa pelo centróide de folhas, o que garante que a solução é válida.

Exercício

(Codeforces - 321C) Você tem uma árvore de n nós. Você precisa colocar um oficial do exército em cada vértice. Cada nível de oficial tem uma letra correspondente, ou seja, são 26 níveis distintos, sendo A o nível mais alto e Z o mais baixo.
Você precisa atribuir esses oficiais de tal forma que se duas cidades distintas tem dois oficiais do mesmo nível, então no caminho entre eles precisa ter um oficial de nível mais alto, para garantir que toda comunicação entre oficiais do mesmo nível é monitorada por um oficial de nível mais alto.
Mostre uma atribuição válida, ou diga que é impossível.

A primeira coisa a se notar é que não podemos ter dois oficiais no nível A, visto que esse é o nível mais alto e não teremos ninguém para monitorar a comunicação entre eles. Se escolhermos um vértice v para ser A, podemos então apagar v da árvore e resolver a floresta restante com cada árvore de forma independente, visto que a comunicação entre quaisquer vértices de duas árvores distintas já está resolvida, pois passará pelo oficial A em v.

Continuando o argumento, após removermos v, cada árvore restante pode ter no máximo um B, e após removermos os vértices com B, as árvores que restaram só podem ter um C, e assim sucessivamente. Com isso, precisamos que o tamanho das árvores diminua rápido o suficiente, pois só temos 26 níveis distintos e a árvore inicial pode ter até $2 \cdot 10^5$ nós.

É aqui que entra a ideia do centróide. Se sempre escolhermos o centróide de cada árvore para ter o oficial de maior nível, sempre diminuímos o tamanho da árvore de n para pelo menos $\lfloor \frac{n}{2} \rfloor$, de modo que vamos precisar de no máximo $log_2(n)$ níveis diferentes, e $26 > log_2(2 \cdot 10^5) \approx 18$.

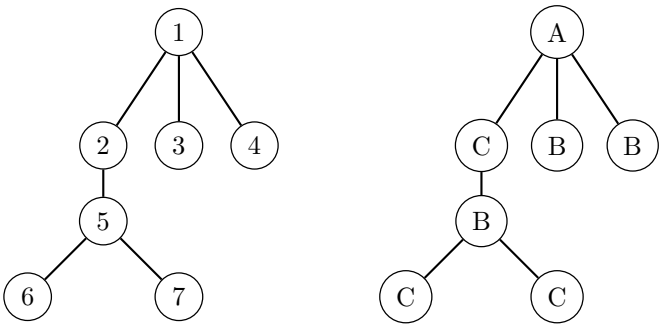

Figure 3.9
Árvore e sua atribuição de oficiais correspondente.

Temos a árvore da figura 3.9 de exemplo. O vértice 1 é o centróide da árvore completa, e por isso foi atribuído com A. Apagando o vértice 1, restam 3 árvores, e o centróide de cada uma foi atribuído com B. Apagando todos os vértices B, encontramos o centróide de cada componente que sobrou, e atribuímos C para esses vértices. Para o exemplo, isso já foi suficiente para atribuir oficiais a todos os vértices.

```
int subtree(int node, int p = 0) {
    sz[node] = 1;
    for (int x : v[node]) {
        if (!vis[x] && x != p) {
            sz[node] += subtree(x, node);
        }
    }
    return sz[node];
}
int centroid(int node, int desired, int p = 0) {
    for (int x : v[node]) {
        if (!vis[x] && x != p && sz[x] > desired) {
            return centroid(x, desired, node);
        }
    }
    return node;
}
void solve(int node, int p = 0) {
    int c = centroid(node, subtree(node) / 2);
    vis[c] = true;
    if (p == 0) {
```

```
        lvl[c] = 'A';
    }
    else {
        lvl[c] = lvl[p] + 1;
    }
    for (int x : v[c]) {
        if (!vis[x]) {
            solve(x, c);
        }
    }
}
```

Código 3.5
Centroid Decomposition para resolver o Problema 321C.

O código 3.5, resolve o problema por completo. Vale notar que implementar a remoção de um vértice em código é muito custoso, então simplesmente marcamos os vértices apagados como visitados e ignoramos os vértices já visitados nas iterações posteriores. Também precisamos constantemente recalcular o tamanho de cada subárvore, já que com a remoção de centróides, o tamanho das subárvores muda.

Esse problema torna-se portanto uma redução direta para o problema de decomposição de centróides. A maioria dos problemas mais complexos de centróide usam a decomposição de centróide como base.

A solução também evidencia a complexidade da solução como $O(nlog(n))$, visto que cada vértice é visitado no máximo $log(n)$ vezes, já que o tamanho de cada componente sempre cai pela metade ou mais.

Exercício

(Codeforces - 342E) Você tem uma árvore de n nós numerados de 1 a n. Inicialmente, o nó 1 está pintado de vermelho e o resto de azul. Você precisa processar dois tipos de query:
1. Pintar o nó azul x de vermelho
2. Calcular qual nó vermelho é o mais próximo do nó x.
Mostre a resposta para cada query do tipo 2.

Achar uma solução $O(nq)$ é relativamente simples, mas precisamos de uma solução mais eficiente. Todo caminho na árvore consiste em subir até um certo nó e depois descer até o nó destino. Podemos pensar em uma solução $O(nq)$ da seguinte forma: Vamos guardar $d(x)$, que é a menor distância para um vértice vermelho que está na subárvore de x. Se x for vermelho, $d(x) = 0$.

Chamaremos o ancestral que está a uma distância i de x de a_i, ou seja, a_0 é o próprio x e a_1 é o pai de x. Para resolver a query do tipo 1, iteramos por todos os a_i, e fazemos $d(a_i) = min(d(a_i), i)$.

Para responder a query do tipo 2, iteramos pelos ancestrais novamente, e pegamos $min(d(a_i) + i)$. Basicamente, o vetor d nos fornece o mínimo que precisamos descer para cada distância de subida distinta.

Essa solução funciona, mas ela roda em $O(nq)$, visto que resolver uma query depende da altura da árvore, que é $O(n)$. A sacada agora é usar essa mesma solução só que na árvore de centróide, cuja altura é $O(log_2(n))$.

A árvore de centróide é construída através da decomposição de centróide. A raiz da árvore é o centróide da árvore completa, os filhos da raiz são os centróides das componentes que restaram após remover a raiz, e assim sucessivamente. Olhando para a figura 3.10 é possível visualizar uma árvore e sua árvore de centróide correspondente.

A solução continua a mesma, para as queries do tipo 1 nós atualizamos todos os ancestrais de x e para responder as queries do tipo 2 nós iteramos pelos ancestrais de x. Contudo, é essencial lembrar que a árvore original é diferente da árvore de centróide, de modo que a distância entre dois nós na árvore original não necessariamente é igual a distância entre os mesmos dois nós na árvore de centróide. A maior parte dos bugs em problemas de árvore de centróide são decorrentes disso, visto que árvores dadas nos casos de teste de exemplo não são grandes o suficiente para que as diferenças entre a árvore original e a árvore de centróide apareçam.

Ademais, enquanto na árvore original era fácil saber a distância de x para um de seus ancestrais, já que os ancestrais eram todos sequenciais, na árvore de centróide essa distância varia, de modo que precisamos usar LCA para calcular isso, o que adiciona um log na complexidade final.

Após a decomposição de centróide em $O(nlog(n))$ para construir a árvore de centróide, precisamos armazenar o pai de cada nó nessa árvore. Depois disso, respondemos cada query em $O(log^2(n))$, pois a altura da árvore é $O(log(n))$ e para cada ancestral rodamos um LCA em $O(log(n))$.

Olhando para o código 3.6, vemos como fazer o update da query 1 e a resposta da query 2, em que $dist$ é uma função que retorna a distância entre dois nós na árvore original.

```
void upd(int node) {
    int x = node;
    while (true) {
        ans[x] = min(ans[x], dist(x, node));
```

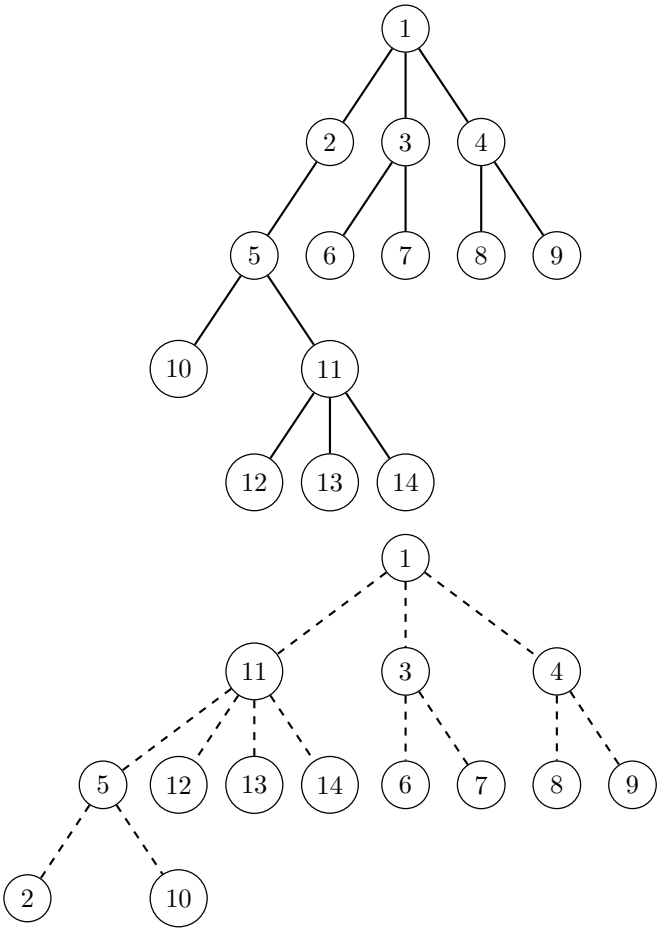

Figure 3.10
Árvore original e árvore de centróide correspondente.

```
        if (x == pai[x]) break;
        x = pai[x];
    }
}
int query(int node) {
    int mn = INF, x = node;
    while (true) {
        mn = min(mn, ans[x] + dist(x, node));
        if (x == pai[x]) break;
        x = pai[x];
    }
    return mn;
```

```
}
```

Código 3.6
Árvore de Centróide e solução das queries.

3.3 Heavy-Light Decomposition

Exercício

(CSES - Path Queries II) Você tem uma árvore de n nós com os nós numerados de 1 a n, e cada nó tem um valor inicial.
Você precisa processar dois tipos de query:
1. Mudar o valor do nó s para x.
2. Calcular o valor máximo no caminho entre a e b.

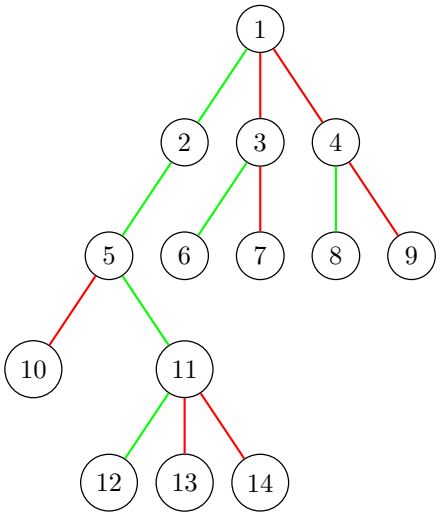

Figure 3.11
Árvore com Heavy-Light Decomposition, as arestas pesadas estão em verde e as leves em vermelho.

Esse problema parece ser perfeito para resolver com uma árvore de segmentos, mas estamos operando em cima de uma árvore, e não de um vetor, então o que podemos fazer? É aqui que surge a Heavy-Light Decomposition, também conhecida como HLD, ou decomposição em leves e pesados. A ideia é dividir as arestas em dois grupos. Para todo vértice, a aresta que aponta para o filho de maior subárvore será a aresta

pesada, enquanto as arestas para os outros filhos serão arestas leves, e empates podem ser quebrados arbitrariamente. A árvore da figura 3.11 mostra essa classificação de arestas, em que as pesadas são verdes e as leves são vermelhas.

O interessante dessa separação é que qualquer caminho de um vértice até a raiz passará por no máximo $O(log(n))$ arestas leves, visto que toda vez que você passar por uma aresta leve, o tamanho da sua subárvore pelo menos dobra, e como a árvore tem apenas n vértices, você só consegue dobrar o tamanho da sua subárvore no máximo $log(n)$ vezes.

Por que o tamanho dobra ao subir uma aresta leve? Como a aresta atual é leve, significa que existe algum outro filho com tamanho de subárvore maior ou igual do que a sua subárvore atual, e agora a subárvore desse filho pesado vai se juntar a sua subárvore atual.

Nós sabemos que em todo caminho que apenas sobe até um ancestral nós passamos por no máximo $O(log(n))$ arestas leves. Como todo caminho na árvore consiste em subir até um certo ancestral e depois descer até o vértice destino, então todo caminho na árvore contém no máximo $O(log(n))$ arestas leves.

Chamaremos as sequências de arestas pesadas de chains. Para trocar de chain você precisa passar por uma aresta leve, e como em todo caminho passamos por $O(log(n))$ arestas leves, significa que passamos por no máximo $O(log(n))$ chains. Agora, se construírmos uma árvore de segmentos por chain, conseguimos quebrar qualquer caminho na árvore em $O(log(n))$ chains e arestas leves, e com $O(log(n))$ consultas em árvores de segmento, resolvemos a query do tipo 2 em $O(log^2(n))$, enquanto a query do tipo 1 consiste em um único update em $O(log(n))$.

A ideia faz sentido, mas como implementar tudo isso?

```
void dfs_sz(int v = 1, int p = 0) {
    sz[v] = 1;
    for(auto &u: g[v]) {
        if (u == p) continue;
        pai[u] =  v;
        dfs_sz(u, v);
        sz[v] += sz[u];
        if(sz[u] > sz[g[v][0]] || g[v][0] == p) {
            swap(u, g[v][0]);
        }
    }
}
void dfs_hld(int v = 1, int p = 0) {
    in[v] = t++;
    for(auto u: g[v]) {
        if (u == p) continue;
        head[u] = (u == g[v][0] ? head[v] : u);
```

```
        dfs_hld(u, v);
    }
    out[v] = t;
}
int query_path(int a, int b) {
    int ans = -INF; // INF = 1e9 + 5
    while (head[a] != head[b]) {
        if (in[a] < in[b]) swap(a, b);
        ans = max(ans, query(in[head[a]], in[a]));
        a = pai[head[a]];
    }
    if (in[a] < in[b]) swap(a, b);
    return max(ans, query(in[b], in[a]));
}
```

Código 3.7
Heavy-Light Decomposition.

Analisando cada função do código 3.7, vemos que dfs_sz calcula o tamanho de cada subárvore e coloca o filho pesado no começo da lista de adjacência de cada vértice. dfs_hld calcula o tempo de entrada $in(v)$ e saída $out(v)$ da dfs para todo vértice, e também calcula $head(v)$, que é o começo da chain pesada a qual o vértice v pertence.

Podemos construir uma única árvore de segmentos em cima dos vértices, de tal forma que o segmento $[in(head(v)), in(v)]$ contém todos os vértices entre v e o começo da chain pesada a qual o vértice v pertence. Com isso, a função *query_path* recebe o início e o fim do caminho, e, enquanto eles estiverem em chains diferentes, ele sobe o vértice que foi acessado depois na dfs para a próxima chain, e vai mantendo o resultado da função *query*, que é uma query na árvore de segmentos.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$in[i]$	1	2	9	12	3	10	11	13	14	8	4	5	6	7
$out[i]$	15	9	12	15	9	11	12	14	15	9	8	6	7	8

Table 3.1
Vetores de *in* e *out* para a árvore da figura 3.11.

Vamos simular essa funções rodando na árvore da figura 3.11 para ficar mais claro. A tabela 3.1 mostra o valor de *in* e *out* retornado pelas duas primeiras funções, e a figura 3.11 mostra quais arestas são leves e pesadas. Digamos que queremos saber o valor máximo no caminho entre o vértice 10 e 8. A função *query_path* pegará o máximo entre os intervalos $[in(4), in(8)]$, $[in(10), in(10)]$ e $[in(1), in(5)]$. São apenas 3 queries que contém todos os vértices no caminho entre o 10 e o 8, uma

para cada chain. O intervalo $[in(10), in(10)]$ se refere à chain que contém apenas o vértice 10, o intervalo $[in(1), in(5)]$ se refere à chain que vai do vértice 12 ao 1, mas pega apenas os valores entre o vértice 5 e 1 e o intervalo $[in(4), in(8)]$ se refere à chain que vai do vértice 8 ao 4.

Para a query do tipo 1, apenas fazemos um update na seg no ponto $in(x)$.

Exercício

(Codeforces Gym - 101908L) Você tem uma árvore de n nós e precisa processar q queries da forma: (a, b) e (c, d). Para cada query, responda quantos vértices comuns existem entre o caminho de a pra b com o caminho de c pra d.

Esse problema é bem simples de resolver com HLD. Podemos realizar a decomposição e construir uma seg com operações de soma e queries de quantos nós tem o valor máximo. Com uma alteração simples na função *query_path*, conseguimos uma função que soma 1 em todos os vértices de um caminho. Agora, para responder uma query, somamos 1 em todo vértice no caminho entre a e b e 1 em todo vértice no caminho entre c e d. Então fazemos uma query de quantos vértices tem o valor máximo de 2, e por fim subtraímos 1 dos caminhos entre a e b e c e d para dar um reset na seg.

Dica

Antes de implementar HLD, tente simplificar a ideia e resolver o problema de outra forma.

Implementar uma decomposição em leves e pesados e uma árvore de segmento pode ser bem demorado, além de que a solução fica $O(qlog^2(n))$, por isso vale ressaltar que esse, assim como muitos problemas de HLD, podem ser resolvidos usando apenas LCA em $O(qlog(n))$. Novamente usamos a ideia de que todo caminho entre o vértice a e b em uma árvore consiste de primeiro subir de a até $lca(a, b)$ e depois descer do $lca(a, b)$ até b. Assim, se queremos encontrar a quantidade de vértices na interseção dos caminhos (a, b) e (c, d), podemos quebrar os caminhos em $(a, lca(a, b))$, $(b, lca(a, b))$, $(c, lca(c, d)$ e $(d, lca(c, d))$ e então contar as interseções par a par. Todos os caminhos agora são de um formato mais fácil, eles apenas sobem, então a quantidade de nós na interseção entre os caminhos $(x, anc(x))$ e $(y, anc(y))$ é $dep(lca(anc(x), anc(y)) - dep(lca(x, y))$.

Em que $dep(x)$ é a profundidade do vértice x. Alguns nós serão contados duas vezes, mas isso é fácil de corrigir na implementação.

Exercício

(CSES - Subtree Queries) Você tem uma árvore de n nós numerados de 1 a n e o nó 1 é a raiz. Cada nó possui um valor. Você precisa processar q queries. Existem dois tipos de query:
1. Troque o valor do nó i para x.
2. Calcule a soma dos valores de todos os nós na subárvore de i.

Se conseguíssemos transformar a árvore em um vetor de tal forma que toda subárvore fosse um intervalo contínuo nesse vetor, processar a query 1 seria apenas alterar o valor de uma das posições, enquanto a query 2 seria calcular a soma de um intervalo. Isso poderia ser resolvido por uma BIT ou uma árvore de segmentos.

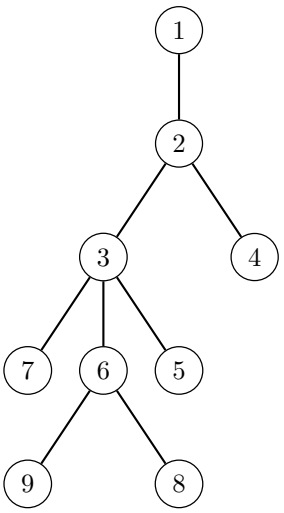

Figure 3.12
Exemplo de árvore.

Existe uma forma simples de transformar uma árvore em um vetor que satisfaz essa propriedade. O nome dessa técnica é Euler Tour, e ela consiste em rodar uma DFS e ordenar os vértices pela ordem em que são visitados e calcular o tamanho de cada subárvore. Podemos ver esse processo no código 3.8, em que o vetor ***euler*** é a ordem em

i	1	2	3	4	5	6	7	8	9
euler[*i*]	1	2	3	7	6	9	8	5	4
pos[*i*]	1	2	3	9	8	5	4	7	6
sub[*i*]	9	8	6	1	1	3	1	1	1

Table 3.2
Vetores de Euler Tour, posição no tour e tamanho de subárvore para a árvore da figura 3.12.

que os vértices são visitados pela DFS, ou seja, *euler*[*i*] é o *i*-ésimo nó visitado pela DFS, *pos*[*i*] indica a posição do nó *i* no vetor *euler* e *sz*[*i*] é o tamanho da subárvore de *i*. Olhando para a tabela 3.2, vemos o valor de cada um desses vetores para a árvore da figura 3.12.

```
void dfs(int node, int pai) {
    sz[node] = 1;
    euler.push_back(node);
    pos[node] = (int)euler.size()-1;
    for (int x : v[node]) {
        if (x != pai) {
            dfs(x, node);
            sz[node] += sz[x];
        }
    }
}
```

Código 3.8
Euler Tour.

Note que para todo nó *i*, o intervalo $[euler[pos[i]], euler[pos[i] + sz[i])$ contém todos os vértices na subárvore de *i*. E isso ocorre pela natureza recursiva da DFS, em que primeiro visitamos todos os nós de uma subárvore para então visitar os nós que estão fora dessa subárvore.

Aplicando alguma estrutura que suporte atualização de valores e soma de intervalos em cima do vetor *euler*, conseguimos resolver o problema em $O(nlogn)$.

Exercício

(Hackerrank - Subtrees and Paths) Você tem uma árvore de *n* nós com os nós numerados de 1 a *n* e o nó 1 é a raiz. Cada nó começa com valor 0.
Você precisa processar dois tipos de query:
1. Adicionar *t* ao valor de todos os nós na subárvore de *i*.
2. Calcular o valor máximo no caminho de *a* até *b*.

Esse problema nos pede alguma estrutura que una o euler tour com HLD. No entanto, se olharmos para o código 3.7 e para a tabela 3.1, veremos que o vetor *in* é análogo ao vetor *euler*, no sentido de que ambos marcam a ordem de visitação da dfs. Dessa forma, o intervalo $[in(i), in(i) + sz(i))$ contém todos os nós da subárvore de i, de modo que conseguimos realizar updates e responder queries para caminhos na árvore ou subárvores completas usando apenas HLD.

3.4 Miscelânea

Exercício

(Codeforces - 600E) Você tem uma árvore de n nós em que o nó 1 é a raiz. Cada nó tem uma cor.
Para uma subárvore, vamos chamar a cor c de dominante se não existe nenhuma outra cor que aparece mais que c nessa subárvore. Ou seja, é possível que mais de uma cor seja dominante, e cada cor é representada por um número de 1 a n. Para cada vértice v, você precisa dizer qual a soma das cores dominantes na subárvore de v.

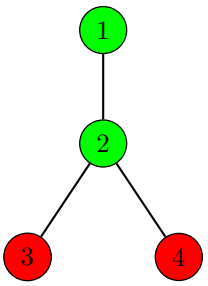

Figure 3.13
Árvore com os vértices coloridos e numerados. Consideramos a cor verde como sendo o número 1 e a cor vermelha como sendo o número 2.

Olhando para a figura 3.13, se considerarmos que o vermelho é o número 2 e o verde é o número 1, temos que a cor vermelha é dominante para os vértices 2, 3 e 4, de modo que a soma das cores dominantes seria 2. Já para o vértice 1, as cores dominantes são o verde e o vermelho, e a soma daria 3.

Vamos pensar em uma estrutura abstrata para resolver esse problema, como uma caixa, que inicialmente está vazia. Essa caixa é uma estrutura de dados que muda dependendo do problema. No nosso caso, é um vetor que guarda para cada cor, quantos vértices já apareceram com aquela cor. Nós podemos colocar e tirar vértices dessa caixa. Para esse problema, colocar um vértice de cor c na caixa seria somar um no contador da cor c, enquanto que remover um vértice de cor c seria subtrair um do contador da cor c. Por fim, dependendo de quais vértices estão na caixa, a caixa produz um resultado. No nosso caso, esse resultado é a soma das cores dominantes dos vértices da caixa.

O problema quer que, para todo vértice v, coloquemos todos os vértices da subárvore de v na caixa e peguemos o resultado da caixa. Uma solução quadrática seria iterar por todos os vértices v em qualquer ordem, então colocamos todos os vértices da subárvore de v na caixa, guardamos o resultado, e então tiramos todos os vértices da caixa, de forma que a caixa sempre fica vazio entre o processamento de um vértice e o próximo.

No entanto, podemos usar o conceito de leves e pesados da seção anterior para encontrar uma ordem mais eficiente de colocar e tirar os elementos da caixa.

```
int calcSz(int node, int pai) { // pré-cálculo
    sz[node] = 1;
    for (int x : v[node]) {
        if (x != pai) {
            sz[node] += calcSz(x, node);
        }
    }
    return sz[node];
}
void dfs(int node, int parent, bool keep) {
    // achar o filho pesado
    int mx = -1, bigChild = -1;
    for (int x : v[node]) {
        if (x != parent && sz[x] > mx) {
            mx = sz[x], bigChild = x;
        }
    }

    // resolver os filhos leves e tirar eles da caixa
    for (int x : v[node]) {
        if (x != parent && x != bigChild) {
            dfs(x, node, 0);
        }
    }

    // resolver o filho pesado e manter ele na caixa
    if (bigChild != -1) {
        dfs(bigChild, node, 1);
```

```
    }

    // adicionar à caixa todos os nós que estão na subárvore do
        nó atual
    // mas não estão na subárvore do filho pesado
    add(node, parent, bigChild);

    // calcular resposta para o nó atual
    ans[node] = get_ans();

    // tirar todos os nós da subárvore, caso node seja um filho
        leve
    if (!keep) reset(node, parent);
}
```

Código 3.9
Template para o algoritmo da caixa (Sack).

O código 3.9 é um template geral para resolver esse tipo de problema. A ideia é primeiro calcular o tamanho de cada subárvore, e depois rodar uma DFS. Para cada vértice, encontramos qual é seu filho pesado, e o resto dos filhos classificamos como leves, assim como no HLD.

Primeiro resolvemos recursivamente os filhos leves, sempre limpando a caixa após visitar cada filho. Depois disso, resolvemos o filho pesado, mas mantemos toda sua subárvore na caixa. Por fim, adicionamos *node* e as subárvores de todos os filhos leves à caixa, de modo que agora temos a subárvore completa de *node* na caixa, e então pegamos a resposta atual da caixa. No final, se *node* for um filho leve, removemos toda sua subárvore da caixa.

Essa ordem diferente de adicionar e remover os vértices da caixa nos proporciona uma complexidade de $O(nlog(n))$, isso porque o número de vezes que um vértice é adicionado/removido da caixa é igual ao número de arestas leves no caminho desse nó à raiz, e sabemos que esse número é no máximo $log_2(n)$, pois sempre que passamos por uma aresta leve, o tamanho da subárvore cai por pelo menos metade.

Esse é um template muito poderoso que se encaixa em vários problemas de árvore que perguntam sobre alguma propriedade para todo vértice. Essa técnica é conhecida como sack. Se você conseguir uma solução que usa a caixa, basta pensar em como implementar as funções de *add*, *reset* e *get_ans*.

No nosso caso, podemos manter o vetor $cnt(i)$, que guarda quantos vértices da cor i estão na caixa, o vetor $soma(i)$, que guarda a soma das cores que aparecem i vezes, e a variável *most*, que guarda o número de

vezes que a cor mais frequente aparece na caixa. Com isso, o *get_ans* retornaria *soma*(*most*).

Exercício

(Codeforces - 815C) Karen vai ao supermercado com b reais. O supermercado vende n produtos numerados de 1 a n, o produto i custa $c_i > 0$ reais, e cada produto só pode ser comprado uma única vez.
Karen também tem um cupom por produto. Se ela usar o cupom do produto i, seu preço se torna d_i, em que $0 < d_i < c_i$. No entanto, para todo $i \geq 2$, o cupom i só pode ser usado se o cupom $x_i < i$ também tiver sido usado.
Qual a maior quantidade de produtos que Karen consegue comprar?

Note que esse problema descreve uma árvore enraizada no vértice 1. Cada vértice $i \geq 2$ tem um pai x_i. A restrição $x_i < i$ garante que não existem ciclos. A restrição de cupons basicamente significa que para comprar o produto i com cupom, temos que ter comprado o produto de todos os seus ancestrais na árvore também. Para comprar sem desconto não existem restrições.

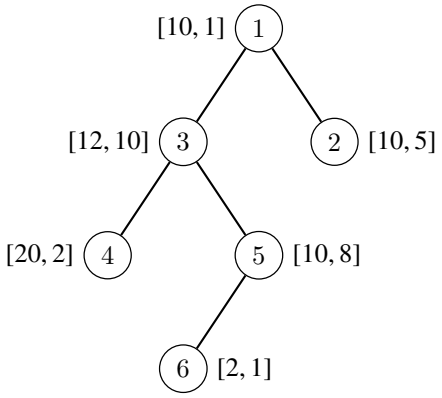

Figure 3.14
Exemplo de árvore da Karen. Ao lado de cada nó tem o preço cheio c_i e o preço com desconto $c_i - d_i$.

Vamos olhar para o exemplo da figura 3.14. Ao lado de cada nó temos o preço sem cupom c_i e o preço com cupom d_i. Digamos que $b = 8$. Nesse caso o ótimo seria comprar o item 1 e 2 com cupom e então o item 6 sem cupom, totalizando $1 + 5 + 2 = 8$ reais gastos e 3 produtos comprados. Agora se $b = 16$, o ótimo é comprar os itens 1, 3 e 4 com cupom e o item 6 sem cupom, totalizando $1 + 10 + 2 + 2 = 15$ reais gastos e 4 produtos comprados.

Ao invés de pensar na quantidade máxima de itens que conseguimos comprar com b reais, vamos calcular a menor quantidade de reais para conseguir comprar i itens, e no final iteramos para encontrar o maior i que precisa de b ou menos reais.

Com isso, definimos uma programação dinâmica: $f(i, j)$ é o menor custo para comprar j itens na subárvore de i, sendo que ainda podemos usar cupons, ou seja, compramos todos os ancestrais de i. E $g(i, j)$ é o menor custo para comprar j itens na subárvore de i, sendo que não podemos mais usar cupons, ou seja, deixamos de comprar algum ancestral de i. No final, iteramos pelos valores de $f(1, j)$ para encontrar a resposta.

Vamos calcular essa dp enquanto fazemos uma DFS na árvore, ou seja, para calcular a dp do nó i, primeiro calculamos a dp de todos os filhos de i. Inicialmente, $f(i, 0) = g(i, 0) = 0$, $f(i, 1) = d_i$ e $g(i, 1) = c_i$, enquanto todos os outros valores começam em infinito.

```
void dfs(int node) {
    g[node][0] = f[node][0] = 0;
    f[node][1] = c[node];
    g[node][1] = d[node];
    sub[node] = 1;
    for (int x : v[node]) {
        dfs(x);
        for (int i = 0; i <= sub[node] + sub[x]; i++) {
            nf[i] = ng[i] = 1e9+1;
        }
        for (int i = 0; i <= sub[node]; i++) {
            for (int j = 0; j <= sub[x]; j++) {
                ng[i+j] = min(ng[i+j], g[node][i] + g[x][j]);
            }
        }
        for (int i = 1; i <= sub[node]; i++) {
            for (int j = 0; j <= sub[x]; j++) {
                nf[i+j] = min(nf[i+j], f[node][i] + f[x][j]);
            }
        }
        sub[node] += sub[x];
        for (int i = 0; i <= sub[node]; i++) {
            g[node][i] = ng[i];
```

```
            f[node][i] = nf[i];
        }
    }
    for (int i = 0; i <= sub[node]; i++) {
        f[node][i] = min(f[node][i], g[node][i]);
    }
}
```

Código 3.10
DFS que realiza o merge dos filhos e calcula a dp que resolve o problema 600E.

Vamos analisar o código 3.10, em que $sub[i]$ é o tamanho da subárvore de i, nf e ng são vetores auxiliares para o cálculo da dp.

A ideia é fazer o merge das subárvores conforme iteramos pelos filhos. Antes de iterar pelos filhos, os vetores $f[node]$ e $g[node]$ consideram que a subárvore de $node$ é apenas o vértice $node$. Depois que já iteramos pelos k primeiros filhos, os vetores $f[node]$ e $g[node]$ vão considerar a subárvore de $node$ como sendo o vértice $node$ mais as subárvores dos k primeiros filhos. Quando já tivermos iterado por todos os filhos, $f[node]$ e $g[node]$ terão seus valores completos, considerando a subárvore inteira de $node$.

Ao iterar pelo i-ésimo filho de $node$, os vetores $f[node]$, $g[node]$ e $sub[node]$ consideram que a subárvore de $node$ é formada por $node$ mais as subárvores dos $i-1$ primeiro filhos de $node$. Agora queremos fazer o merge dessa subárvore incompleta de $node$ com a subárvore do i-ésimo filho. Enquanto fazemos esse merge, os vetores nf e ng representam as dps da subárvore formada por $node$ mais as subárvores dos i primeiros filhos. Por isso, ao terminar de iterar pelo i-ésimo filho, fazemos $f[node] = nf$ e $g[node] = ng$.

Para exemplificar, considere a árvore da figura 3.15. Na esquerda temos a árvore completa, mas imagine que estamos no meio da dfs, visitando o vértice 3. Esse vértice tem 3 filhos: 4, 6 e 5. Ao iniciarmos a DFS em 3 e calcular o caso base, é como se a subárvore de 3 tivesse apenas o 3. Agora, olhando para a árvore da direita, ela mostra a situação em que já passamos pelos dois primeiros filhos e agora estamos no terceiro filho, o vértice 5. Ao iterar pelo terceiro filho de 3, estamos fazendo o merge dos vértices da subárvore atual de 3, que contém os vértices 3, 4, 6, 7 e 8, com a subárvore de seu filho, que contém os vértices 5 e 9.

Passando pelo i-ésimo filho, para calcular ng, iteramos por quantos produtos vamos comprar sem cupom da atual subárvore de $node$ (calcu-

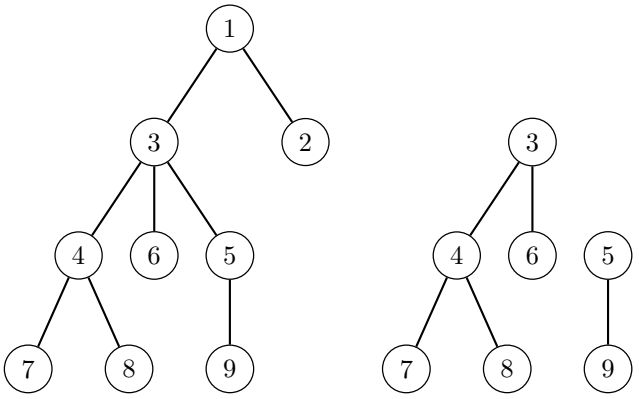

Figure 3.15
Exemplo de árvore da Karen durante o cálculo da dp.

lado por $g[node]$), e o resto devemos comprar da subárvore do i-ésimo filho (chamando esse filho de x, isso é calculado por $g[x]$). Ou seja,

$$ng[k] = min_j(g[node][j] + g[x][k - j]) \tag{3.1}$$

Para calcular nf, ainda iterando pelo i-ésimo filho, fazemos algo parecido, iteramos por quantos produtos vamos comprar podendo usar cupom da atual subárvore de $node$ (calculado por $f[node]$), e o resto devemos comprar da subárvore do i-ésimo filho (chamando esse filho de x, isso é calculado por $f[x]$). No entanto, para poder usar cupom na subárvore de x, precisamos ter comprado o produto em i, de modo que forçamos a comprar pelo menos um produto da atual subárvore de $node$. Ou seja,

$$nf[k] = min_{j \geq 1}(f[node][j] + f[x][k - j]) \tag{3.2}$$

Depois de iterarmos pelos filhos, rodamos $f[node] = min(f[node], g[node])$.

Isso resolve o problema, mas o interessante é calcularmos a complexidade dessa solução. Note que, ao calcular a dp, estamos basicamente construindo a árvore aos poucos, fazendo o merge das subárvores até obtermos a árvore completa na raiz. Outro detalhe importante é que sempre iteramos apenas até o tamanho de cada subárvore (sub). Ou seja, fazemos o merge de uma subárvore de tamanho a com outra de tamanho b em $O(a \cdot b)$. Ao realizarmos esses merges, todo nó vai ser pareado com todos os outros nós exatamente uma vez, o que resulta em uma complexidade final de $O(n^2)$.

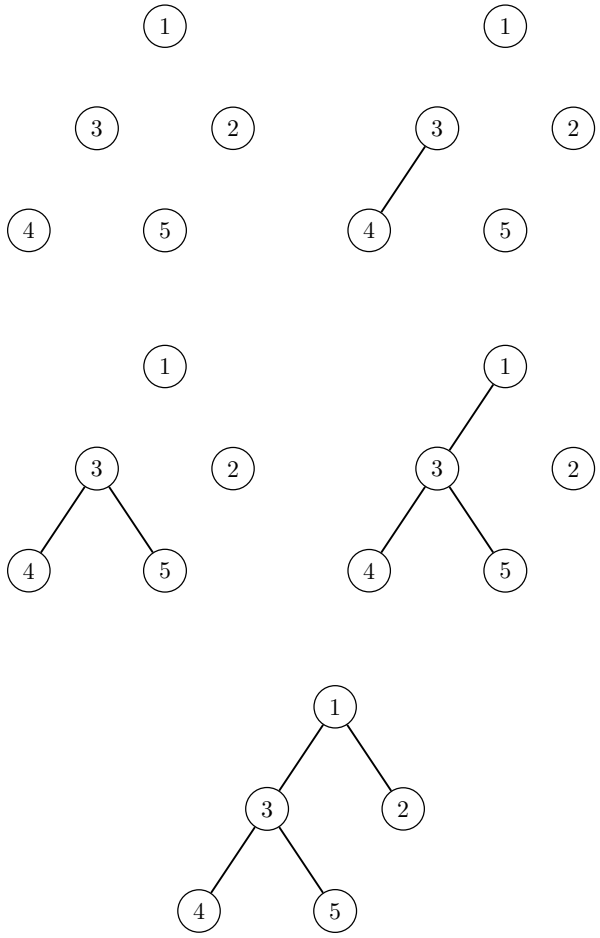

Figure 3.16
Árvore da Karen sendo montada durante a DFS.

Para entender melhor essa complexidade, considere os merges realizados na figura 3.16. Assumindo que a DFS sempre visita os filhos da subárvore da esquerda primeiro, inicialmente todo nó está desconectado e a subárvore de i tem apenas o vértice i para todo i. Ao chegarmos no vértice 3, vamos unir sua subárvore com a subárvore de 4. Nesse momento "pagamos" o par $(3, 4)$. Depois disso, passamos para a segunda árvore da imagem, e fazemos o merge da subárvore do 3 com a subárvore do 5, e aqui "pagamos" os pares $(4, 5)$ e $(3, 5)$. Na terceira árvore da imagem, já completamos a DFS em 3 e voltamos para 1, seu pai. Fazendo

o merge da subárvore de 1 com a de 3, "pagamos" os pares $(1,3)$, $(1,4)$ e $(1,5)$. Por fim, na quarta árvore da imagem, fazemos o merge da subárvore de 1 com a subárvore de 2 e "pagamos" os pares restantes: $(2,1)$, $(2,3)$, $(2,4)$ e $(2,5)$. Com isso, chegamos na quinta árvore da imagem, que é a árvore original, e "pagamos" cada par exatamente uma vez, o que demonstra a complexidade quadrática da solução.

Exercises

(2.1) [Codeforces - 715C]

Você tem uma árvore com n vértices em que cada aresta tem um único digito maior que 0, ou seja, um inteiro entre 1 e 9.

Você também tem um inteiro positivo M que é coprimo com 10, ou seja, não é divisível por 2 nem 5.

Você quer saber quantos pares ordenados (u, v) existem tal que, se você concatenar os dígitos das arestas no caminho de u até v, o inteiro resultante é divisível por M.

Ou seja, se as arestas que você passa no caminho de u a v forem 6, 4 e 1, você contaria o par (u, v) se 641 for divisível por M, e contaria o par (v, u) se 146 for divisível por M.

$2 \leq n \leq 200000$, $M \leq 10^9$, $GCD(M, 10) = 1$.

(2.2) [Codeforces Gym - 100633D]

Você tem uma árvore com n vértices em que as arestas têm peso e cada vértice tem uma cor. Inicialmente todo vértice está pintado com a cor 0.

Você quer responder q queries de dois tipos:

1. Pintar todos os vértices com distância menor ou igual que d para o vértice v com a cor c.
2. Dizer qual a cor do vértice v.

$2 \leq n, q \leq 10^5$, $0 \leq c, d \leq 10^9$, $1 \leq v \leq n$.

(2.3) [IOI 2011 - Race]

Você tem uma árvore com n vértices em que as arestas têm peso. Você quer achar o caminho de comprimento exatamente k com o menor número de arestas, ou dizer que tal caminho não existe.

Lembrando que um caminho não pode ter arestas repetidas.

$1 \leq n \leq 2 \cdot 10^5$, $1 \leq k \leq 10^6$.

(2.4) [Codeforces - 741D]

Você tem uma árvore com n vértices em que cada aresta tem uma letra entre a e v. A árvore tem o vértice 1 como raiz.

Para cada vértice, você precisa responder qual o tamanho do maior caminho na subárvore desse vértice tal que, se você pegar as letras

das arestas desse caminho, você consegue permutar elas para formar um palíndromo.
$2 \leq n \leq 5 \cdot 10^5$.

(2.5) [Codeforces - 1254D]
Você tem uma árvore com n vértices em que cada vértice tem um valor, inicialmente o valor de todo vértice é 0.
Você quer responder q queries de dois tipos:
1. Dado um vértice v e um valor d, você escolhe aleatoriamente e de forma uniforme um vértice r. Para todo vértice u tal que o caminho entre u e r na árvore passa por v, você aumenta o valor de u por d.
2. Dizer qual o valor esperado do vértice v.
O valor esperado pode ser representado como $\frac{P}{Q}$, em que P e Q são inteiros coprimos. Imprima o resultado de $P \cdot Q^{-1}$ módulo 998244353.
$2 \leq n, q \leq 1.5 \cdot 10^5$, $1 \leq v \leq n$, $1 \leq d \leq 10^7$.

(2.6) [CSES - Fixed Length Paths II] Você tem uma árvore de n nós e quer contar quantos caminhos distintos existem com no mínimo k_1 e no máximo k_2 arestas.
Resolver em $O(n)$.

4 Programaçao dinâmica

4.1 Bitmasks

Em muitos problemas de programação dinâmica é necessário que um dos estados da dp seja um conjunto de elementos. Isso geralmente é feito através de bitmasks que fornecem soluções exponenciais, mas que ainda são mais rápidas do que um simples brute force. Nessa seção vamos explorar esses problemas.

Exercício

(Atcoder Educational DP Contest - Matching) Você tem n homens e n mulheres numerados de 1 a n e uma matriz A. Se $A_{i,j} = 1$, o i-ésimo homem é compatível com a i-ésima mulher, mas se for 0 eles não são compatíveis.
Você está tentando criar n pares, cada par formado por um homem e uma mulher compatíveis. Cada pessoa só pode participar de um par, ou seja, no final todo mundo vai estar em exatamente um par. Quantas formas existem de fazer isso módulo $10^9 + 7$?
$n \leq 21$.

A primeira coisa a se notar nesse problema é o limite $n \leq 21$. Com um limite tão baixo, podemos pensar em um brute force. Podemos tentar parear o primeiro homem com qualquer uma das n mulheres, o segundo homem com alguma das $n - 1$ que sobraram, o terceiro com $n - 2$, e assim sucessivamente. Assim, temos $n!$ pareamentos possíveis, e então checamos quais desses são válidos. O problema é que $21! = 5.1 \cdot 10^{19}$, de modo que se torna inviável iterar por todos os pareamentos.

Podemos pensar então em uma $dp[i][S]$, que é a quantidade de pareamentos possíveis entre os primeiros i homens com todas as mulheres do

conjunto S. O caso base é $dp[0][\emptyset] = 1$, que é o estado em que ninguém está pareado. Já a resposta está em $dp[n][\{1, 2, ...n\}]$, que significa que já pareamos todo mundo.

Para as transições:

$$dp[i][S] = \sum_{j=1}^{n} dp[i-1][S \setminus j] \cdot f(S, i, j) \tag{4.1}$$

Em que $f(S, i, j)$ indica se i pode ser pareado com j, ou seja, é 1 quando $A_{i,j} = 1$ e j pertence a S, e 0 nos outros casos.

Como existem 2^n possibilidades para o conjunto S (cada um dos n itens pode estar ou não presente em cada um dos conjuntos), temos uma dp com $2^n n$ estados e transição em $O(n)$, resultando na complexidade final de $O(2^n n^2)$. Para termos uma ideia da diferença em relação à $O(n!)$, $2^{21}21^2 = 9 \cdot 10^7$.

Mas como vamos implementar essa dp? Como representar um conjunto de forma eficiente sem usar um set? A ideia é usar um inteiro para representar um conjunto, pensando em sua representação binária. Chamamos esse inteiro de uma máscara de bits, ou apenas de mask.

Nesse caso, queremos todos os conjuntos dos números de 1 a n. Podemos pensar que cada número é um bit. Assim, se uma mask tem o bit i aceso, quer dizer que o número i está no conjunto representado por essa mask. Por exemplo, a mask do número $13 = 1011_2$ representa o conjunto $\{1, 2, 4\}$, enquanto a mask do número $31 = 11111_2$ representa o conjunto $\{1, 2, 3, 4, 5\}$ e a mask do número 0 o conjunto vazio.

Essa representação é muito eficiente, pois nos fornece uma bijeção entre todos os conjuntos dos números de 1 a n com os inteiros de 0 a $2^n - 1$. Além disso, podemos realizar operações com bits para calcular operações de conjuntos. Em código, para adicionar o número j ao conjunto representado por i fazemos i *or* 2^j. Já para checar se um número j pertence ao conjunto representado pela mask i, checamos se $(i|2^j) == i$.

Uma outra vantagem é que geralmente para poder calcular a dp de um conjunto S já precisamos ter calculado a dp de todos os subconjuntos de S. Como todo inteiro que representa um subconjunto de S é menor que o inteiro que representa S, podemos simplesmente calcular a dp dos conjuntos iterando pelas masks de 0 a $2^n - 1$, e essa ordem será satisfeita.

```
int A[22][22], dp[22][(1 << 21)], MOD = 1e9+7;

bool f(int mask, int i, int j) {
  return A[i][j] && ((mask | (1 << j)) == mask);
}
```

```
int calc_dp(int n) {
  dp[0][0] = 1;
  for (int i = 0; i < n; i++) {
    for (int mask = 0; mask < (1 << n); mask++) {
      if (__builtin_popcount(mask) != i + 1) continue;
      for (int j = 0; j < n; j++) {
        dp[i+1][mask] += dp[i][mask ^ (1 << j)] \cdot f(mask, i,
            j);
        if (dp[i + 1][mask] >= MOD) dp[i + 1][mask] -= MOD;
      }
    }
  }
  return dp[n][(1 << n) - 1];
}
```

Código 4.1
Bitmask DP.

No código 4.1, a função popcount retorna o número de bits ativos de um número.

Exercício

(Atcoder Educational DP Contest - Grouping) Você tem n coelhos, numerados de 1 a n. Você também tem uma matriz A de compatibilidade. $A_{i,j}$ pode ser positivo, negativo ou zero e indica a compatibilidade do coelho i com o coelho j.
Você está dividindo os coelhos em grupos. Cada coelho deve participar de exatamente um grupo. No final, se o coelho i e j estão no mesmo grupo, $A_{i,j}$ é somado ao score. Qual o maior score que você consegue ao dividir os coelhos?
Note que se todas as compatibilidades forem negativas, podemos criar n grupos com um coelho em cada e atingir um score de zero.
$n \leq 16$.

Esse problema é bem parecido com o anterior, a diferença aqui é que não estamos criando pares, mas possivelmente grupos maiores. Se definirmos $dp[S]$ como o maior score que alcançamos se todos os coelhos do conjunto S já estão em grupos, temos que o caso base é $dp[\emptyset] = 0$ e a resposta final estará em $dp[\{1, 2, ...n\}]$. Também definimos $score[S]$ como o score do grupo formado pelos coelhos de S. Com isso, temos as transições da dp

$$dp[S] = \max_{X \subseteq S}(dp[S \setminus X] + score[X]) \tag{4.2}$$

Ou seja, para um conjunto S, iteramos por todos os subconjuntos de S para fazer a transição. Traduzindo para código, para cada calcular a dp de uma mask, iteramos por todas as suas submasks.

```
for (int submask=mask; submask; submask=(submask-1)&mask)
```

Código 4.2
Loop que passa por todas as submasks de uma mask, com exceção da mask em que todos os bits estão apagados.

Usando o código 4.2, iteramos por todas as submasks de uma mask sem repetição. Vamos examinar por que isso ocorre. Suponha que estamos em uma submask, e queremos passar para a próxima submask. Subtraindo 1, nós removemos o bit 1 menos significativo, e todos os bits à sua direita se tornam 1. Agora removemos os bits extras realizando um AND com a mask original.

Usando essa técnica de iterar pelas submasks, e a técnica anterior de representar conjuntos como inteiros, conseguimos implementar a equação 4.2 no código 4.3, que assume que já precalculamos o vetor *soma*.

```
int dp[(1 << 16)], soma[(1 << 16)];

int calc_dp(int n) {
  for (int mask = 0; mask < (1<<n); mask++) {
    for (int s = mask; s; s = (s-1)&mask) {
      dp[mask] = max(dp[mask], dp[mask ^ s] + soma[s]);
    }
  }
  return dp[(1 << n) - 1];
}
```

Código 4.3
Dp da equação 4.2.

A quantidade de transições pode parecer ser $O(2^n 2^n) = O(4^n)$, mas na verdade a quantidade é $O(3^n)$.

Para provar isso, vamos calcular quantos pares diferentes $(mask, s)$ existem, visto que fazemos uma transição para cada par desses. Para cada um dos n bits, temos exatamente 3 opções: Ou ele está presente em $mask$ e em s, ou ele está presente apenas em $mask$ ou ele não está presente em nenhum dos dois. A quarta opção, que seria estar apenas em s não é possível pois s é uma submask de $mask$. Com isso, temos 3^n pares diferentes.

Para uma segunda prova, note que se $mask$ tem k bits acesos, então ela terá 2^k submasks diferentes. Como existem $\binom{n}{k}$ máscaras com k bits acesos, temos que a quantidade de transições é dada por

$$\sum_{k=0}^{n} \binom{n}{k} 2^k \tag{4.3}$$

Usando o binômio de Newton

$$\sum_{k=0}^{n} \binom{n}{k} 2^k 1^{n-k} = (1+2)^n = 3^n \tag{4.4}$$

Exercises

2.2) [CSES - Hamiltonian Flights]
Você tem um grafo direcionado com n vértices e m arestas. Você quer ir do vértice 1 ao n passando por todos os vértices exatamente uma vez. Quantos caminhos possíveis existem módulo $10^9 + 7$?
$1 \leq n \leq 20$, $1 \leq m \leq n^2$.

2.4) [Atcoder - ABC187F]
Você tem um grafo não direcionado simples com n vértices e m arestas. Você quer dividir o grafo no menor número de cliques possíveis. Um clique é um subgrafo em que todo mundo está conectado.
$1 \leq n \leq 18$, $0 \leq m \leq \frac{n(n-1)}{2}$.