Graph Theory

1 Diameter

性质：对于任意一个点 u，距离 u 最远的点一定是直径的一个端点。

但是路径不满足这个性质。hack: 考虑路径是直径本身。涉及到路径的时候，需要更细致的讨论。

1.1

前提条件：$\mathrm{w} \geq 0$，但是这个方法找到直径比较方便。

```
vector<i64> dep(n + 1);
auto dfs = [&](auto&& dfs, int u, int fa) -> void {
  for (auto v : adj[u]) {
    if (v == fa) continue;
    dep[v] = dep[u] + w;
    dfs(dfs, v, u);
  }
};

dep[1] = 0, dfs(dfs, 1, 0);
int x = max_element(dep.begin() + 1, dep.end()) - dep.begin();
dep[x] = 0, dfs(dfs, x, 0);

i64 ret = *max_element(dep.begin() + 1, dep.end());
```

1.2

```
i64 ret = 0;
vector<array<i64, 2>> dp(n + 1);

auto dfs = [&](auto&& dfs, int u, int fa) -> void {
  dp[u] = { 0, 0 };
  for (auto v : adj[u]) {
    if (v == fa) continue;
    dfs(dfs, v, u);

    i64 get = dp[v][0] + w;
    if (get > dp[u][0]) {
      dp[u][1] = dp[u][0];
      dp[u][0] = get;
    } else if (get > dp[u][1]) {
      dp[u][1] = get;
    }
  }
  ret = max(ret, dp[u][0] + dp[u][1]);
}; dfs(dfs, 1, 0);
```

1.3 例题

给你一棵树，选择一条长度不超过 k 的路径，最小化树上任意一点到这条路径的距离的最大值。

通过上面的性质，合理地放缩，不容易证明：选择的路径一定是直径的一部分。于是双指针、单调队列一下就做完了。你也可以使用 ST 表和二分大力草过去。

2 Center

感觉这个东西屁用没有啊

最小化 $\max \mathrm{dep}_{\mathrm{center}}(u)$ 的点。

性质：

- 中心最多有两个。如果有两个，他们相邻（和重心一样）。
- 中心一定位于树的直径上
- 对于任意一个点 u，其最长路径一定经过中心。
- 当通过在两棵树间连一条边以合并为一棵树时，连接两棵树的中心可以使新树的直径最小。
- 树的中心到其他任意节点的距离不超过树直径的一半。

换根 dp 维护最大值和次大值即可实现。

```
vector<array<i64, 2>> dp(n + 1);
auto dfs1 = [&](auto&& dfs, int u, int fa) -> void {
  dp[u] = { 0, 0 };
  for (auto [v, w] : adj[u]) {
    if (v == fa) continue;
    dfs(dfs, v, u);

    i64 get = dp[v][0] + w;
    if (get > dp[u][0]) {
      dp[u][1] = dp[u][0];
      dp[u][0] = get;
    } else if (get > dp[u][1]) {
      dp[u][1] = get;
    }
  }
}; dfs1(dfs1, 1, 0);

auto dfs2 = [&](auto&& dfs, int u, int fa) -> void {
  for (auto [v, w] : adj[u]) {
    if (v == fa) continue;
    i64 get;
    if (dp[v][0] + w == dp[u][0]) {
      get = dp[u][1] + w;
    } else get = dp[u][0] + w;

    if (get > dp[v][0]) {
      dp[v][1] = dp[v][0];
      dp[v][0] = get;
    } else if (get > dp[v][1]) {
      dp[v][1] = get;
    }

    dfs(dfs, v, u);
  }
}; dfs2(dfs2, 1, 0);
```

3 Centroid

感觉这个比较重要。这是把分治思想应用到树上的基础，性质也比较多。

性质：

- 删除重心后，任意一棵子树的大小小于等于 $\lfloor \frac{n}{2} \rfloor$
- 考虑以重心为根，那么节点的**深度和最小**
- 最多存在两个重心。如果存在，两个重心相邻，且**删掉这条边后连通块大小相等**
- 删除一个点，如果删的是重心，那么剩余的子树的最大值最小
- 两棵树合并的时候，新树的重心一定在两棵树重心的路径上
- 重心一定在根节点所在重链上。根节点的重子树的重心一定是整棵树重心的后代。

实现代码参考**点分治**部分的 `get_root()` 部分

3.1 例题

其实我也不知道放什么题了。

对于所有 u 求 u 的子树的重心。u 的子树的重心一定是 u 的重儿子的重心的祖先，那么考虑从重儿子的重心暴力往上跳。

注意到跳的边一定是重边，而重边最多也就 $n-1$ 条，所以时间复杂度是 $O(n)$ 的.

```
vector<int> ans(n + 1), son(n + 1), siz(n + 1);
auto dfs = [&](auto&& dfs, int u) -> void {
  siz[u] = 1;
  for (auto v : adj[u]) {
    dfs(dfs, v);
    siz[u] += siz[v];
    if (siz[v] > siz[son[u]]) {
      son[u] = v;
    }
  }
  if (son[u] == 0) {
    assert(adj[u].empty());
    ans[u] = u;
    return;
  }

  ans[u] = ans[son[u]];
  for (int& x = ans[u]; x != u; x = fa[x]) {
    int y = fa[x];
    int now = max(siz[son[x]], siz[u] - siz[x]);
    int nxt = max(siz[son[y]], siz[u] - siz[y]);
    if (nxt >= now) break;
  }
}; dfs(dfs, 1);
```

4 点分治

好像确实就是：如果我能快速求解过某个点的情况，那么就可以用点分治。类似于分治的：如果我能快速求解过区间中点的答案，那么就可以用分治。

```
vector<int> vis(n + 1), siz(n + 1);
auto get_root = [&](int u, int size) -> int {
  int root = 0, Min = size;
  auto dfs = [&](auto dfs, int u, int fa) -> void {
    siz[u] = 1;
    int Max = 0;
    for (auto [v, w] : adj[u]) {
      if (vis[v] || v == fa) continue;
      dfs(dfs, v, u);
      siz[u] += siz[v];
      Max = max(Max, siz[v]);
    }
    Max = max(Max, size - siz[u]);
    if (Max < Min) {
      Min = Max, root = u;
    }
  }; dfs(dfs, u, 0);
  return root;
};

auto dfz = [&](auto&& dfz, int u, int size) -> void {
  u = get_root(u, size);

// 这段是求答案的
// 维护每条链的长度以及它是从哪个儿子出发的，就可以 O(nlogn) 求解
// 时间复杂度 O(nlog^2n)
// Map<int, vector<int>> d;
// d[0].push_back(u);
// auto dfs = [&](auto&& dfs, int u, int fa, int dep, int from) -> void {
//  d[dep].push_back(from);
//  for (auto [v, w] : adj[u]) {
//    if (vis[v] || v == fa) continue;
//    dfs(dfs, v, u, dep + w, from);
//  }
// };
// for (auto [v, w] : adj[u]) {
//  if (vis[v]) continue;
//  dfs(dfs, v, u, w, v);
// }
// for (auto& [x, pos] : d) {
//  sort(pos.begin(), pos.end());
//  pos.erase(unique(pos.begin(), pos.end()), pos.end());
// }
// for (int i = 1; i <= m; ++i) {
//  if (ans[i]) continue;
//  for (auto& [x, pos] : d) {
//    auto it = d.find(query[i] - x);
//    if (it == d.end()) continue;
//    ans[i] |= pos.size() > 1 | (it->second).size() > 1 | pos[0] != (it->second)[0];
//  }
// }
```

```
  vis[u] = 1;
  for (auto [v, w] : adj[u]) {
    if (vis[v]) continue;
    if (siz[v] < siz[u]) {
      dfz(dfz, v, siz[v]);
    } else {
      dfz(dfz, v, size - siz[u]);
    }
  }
}; dfz(dfz, 1, n);
```

5 LCA

5.1 倍增

```
void solve() {
  int n, q, rt;
  std::vector<std::vector<int>> adj(n + 1);

  std::vector<int> dep(n + 1);
  std::vector fa(std::__lg(n) + 1, std::vector<int> (n + 1));

  auto dfs = [&](auto&& dfs, int u) -> void {
    dep[u] = dep[fa[0][u]] + 1;
    for (auto& v : adj[u]) {
      if (v == fa[0][u]) {
        continue;
      }
      fa[0][v] = u;
      dfs(dfs, v);
    }
  }; dfs(dfs, rt);

  for (int i = 1; i <= std::__lg(n); ++i) {
    for (int u = 1; u <= n; ++u) {
      fa[i][u] = fa[i - 1][fa[i - 1][u]];
    }
  }

  auto lca = [&](int u, int v) -> int {
    if (dep[u] < dep[v]) std::swap(u, v);
    while (dep[u] != dep[v]) {
      u = fa[std::__lg(dep[u] - dep[v])][u];
    }
    if (u == v) return u;
    for (int i = std::__lg(n); i >= 0; --i) {
      if (fa[i][u] != fa[i][v]) {
        u = fa[i][u], v = fa[i][v];
      }
    }
    return fa[0][u];
  };
}
```

5.2 树剖

```cpp
struct HLD {
  int n, cur;
  std::vector<std::vector<int>> adj;
  std::vector<int> dfn, idfn, siz, fa, top, dep;

  HLD() = default;
  HLD(int n) { init(n); }

  void init(int n) {
    this -> n = n;
    cur = 0;
    adj.assign(n + 1, {});
    dfn.assign(n + 1, 0);
    idfn.assign(n + 1, 0);
    siz.assign(n + 1, 0);
    fa.assign(n + 1, 0);
    top.assign(n + 1, 0);
    dep.assign(n + 1, 0);
  }

  void add(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  void work(int rt = 1) {
    dfs1(rt), dfs2(rt);
  }

  void dfs1(int u) {
    if (fa[u] != 0) {
      adj[u].erase(find(adj[u].begin(), adj[u].end(), fa[u]));
    }
    siz[u] = 1;
    for (auto& v : adj[u]) {
      dep[v] = dep[fa[v] = u] + 1;
      dfs1(v);
      siz[u] += siz[v];
      if (siz[v] > siz[adj[u][0]]) {
        std::swap(v, adj[u][0]);
      }
    }
  }

  void dfs2(int u) {
    dfn[u] = ++cur;
    idfn[cur] = u;
    for (auto v : adj[u]) {
      top[v] = v == adj[u][0] ? top[u] : v;
      dfs2(v);
    }
  }
```

```
  int lca(int u, int v) {
    while (top[u] != top[v]) {
      if (dep[top[u]] > dep[top[v]]) {
        u = fa[top[u]];
      } else {
        v = fa[top[v]];
      }
    }
    return dep[u] < dep[v] ? u : v;
  }

  int jump(int u, int k) {
    assert(dep[u] >= k);
    int d = dep[u] - k;
    while (dep[top[u]] > d)
      u = fa[top[u]];
    return idfn[dfn[u] - dep[u] + d];
  }
};
```

5.3 O(1) LCA

```
struct LCA {
  int n, cur;
  std::vector<std::vector<int>> adj, st;
  std::vector<int> fa, dep, dfn, siz;

  LCA() = default;
  LCA (int n) { init(n); }

  void init(int n) {
    this -> n = n;
    cur = 0;
    adj.assign(n + 1, {});
    st.assign(std::__lg(n) + 1, std::vector<int> (n + 1));
    fa.assign(n + 1, 0);
    dep.assign(n + 1, 0);
    dfn.assign(n + 1, 0);
    siz.assign(n + 1, 0);
  }

  void add(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  void dfs(int u) {
    if(fa[u] != 0) adj[u].erase(find(adj[u].begin(), adj[u].end(), fa[u]));
    siz[u] = 1;
    st[0][dfn[u] = ++cur] = u;
    for(auto v : adj[u]) {
      dep[v] = dep[fa[v] = u] + 1;
      dfs(v);
      siz[u] += siz[v];
    }
  }
```

```
  int merge(int x, int y) {
    return dep[x] < dep[y] ? x : y;
  }

  void work(int rt = 1) {
    dep[rt] = 1; dfs(rt);
    for(int i = 1; i <= std::__lg(n); ++i) {
      for(int j = 1; j + (1 << i) - 1 <= n; ++j) {
        st[i][j] = merge(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
      }
    }
  }

  int lca(int u, int v) {
    if (u == v) return u;
    u = dfn[u], v = dfn[v];
    if (u > v) std::swap(u, v);
    int k = std::__lg(v - u);
    return fa[merge(st[k][u + 1], st[k][v - (1 << k) + 1])];
  }
};
```

6 rooted functions

```
bool isAncester(int f, int u) {
  return dfn[f] <= dfn[u] && dfn[u] <= dfn[f] + siz[f] - 1;
}

int rootedParent(int rt, int u) {
  if (rt == u) return rt;
  if (!isAncester(u, rt)) return fa[u];
  // 需要保证 adj[u] 里面只有子节点，不能包含父节点
  auto it = std::upper_bound(adj[u].begin(), adj[u].end(), rt, [&](int x, int y) {
    return dfn[x] < dfn[y];
  }) - 1;
  return *it;
}

int rootedSize(int rt, int u) {
  if (rt == u) return n;
  if (!isAncester(u, rt)) return siz[u];
  return n - siz[rootedParent(rt, u)];
}

int rootedLca(int rt, int u, int v) {
  return lca(rt, u) ^ lca(u, v) ^ lca(v, rt);
}
```

7 Tarjan

7.1 强连通分量

```
struct SCC {
  int n;
  std::vector<std::vector<int>> adj;
  std::vector<int> stk;
  std::vector<int> dfn, low, bel;
  int cur, cnt;

  SCC() = default;
  SCC(int n) { init(n); }

  void init(int n) {
    this -> n = n;
    adj.assign(n + 1, {});
    dfn.assign(n + 1, 0);
    low.assign(n + 1, 0);
    bel.assign(n + 1, 0);
    stk.clear();
    cur = cnt = 0;
  }

  void add(int u, int v) {
    adj[u].push_back(v);
  }

  void dfs(int u) {
    dfn[u] = low[u] = ++cur;
    stk.push_back(u);

    for (auto v : adj[u]) {
      if (dfn[v] == 0) {
        dfs(v);
        low[u] = std::min(low[u], low[v]);
      } else if (bel[v] == 0) {
        low[u] = std::min(low[u], dfn[v]);
      }
    }

    if (dfn[u] == low[u]) {
      int x = cnt++;
      do {
        bel[x = stk.back()] = cnt;
        stk.pop_back();
      } while (x != u);
    }
  }

  void work() {
    for (int i = 1; i <= n; i++) {
      if (dfn[i] == 0) dfs(i);
    }
  }
```

```
  auto getGraph() {
    work();
    std::vector<std::vector<int>> adj(cnt + 1);
    for (int u = 1; u <= n; ++u) {
      for (auto v : adj[u]) {
        if (bel[u] != bel[v]) {
          adj[bel[u]].push_back(bel[v]);
        }
      }
    }
    return std::pair { cnt, std::move(adj) };
  }
};
```

7.2 边双连通分量

```
struct EDCC {
  int n;
  int cur, cnt, edges;
  std::vector<int> stk, dfn, low, bel;
  std::vector<std::array<int, 2>> cut;
  std::vector<std::vector<std::array<int, 2>>> adj;

  EDCC() = default;
  EDCC(int n) { init(n); }

  void init(int n) {
    this -> n = n;
    adj.assign(n + 1, {});
    dfn.assign(n + 1, 0);
    low.assign(n + 1, 0);
    bel.assign(n + 1, 0);
    stk.clear();
    cut.clear();
    cur = cnt = edges = 0;
  }
  void add(int u, int v) {
    ++edges;
    adj[u].push_back({ v, edges });
    adj[v].push_back({ u, edges });
  }
  void dfs(int u, int fa) {
    dfn[u] = low[u] = ++cur;
    stk.push_back(u);
    for (auto [v, w] : adj[u]) {
      if (w == fa) continue;
      if (dfn[v] == 0) {
        dfs(v, w);
        low[u] = std::min(low[u], low[v]);
        if (dfn[u] < low[v]) {
          cut.push_back({ u, v });
        }
      } else {
        low[u] = std::min(low[u], dfn[v]);
      }
    }
```

```
    if (low[u] == dfn[u]) {
      int x = ++cnt;
      do {
        bel[x = stk.back()] = cnt;
        stk.pop_back();
      } while (x != u);
    }
  }

  void work() {
    for (int i = 1; i <= n; ++i) {
      if (dfn[i] == 0) dfs(i, 0);
    }
  }

  auto getTree() {
    work();
    std::vector<std::vector<int>> adj(cnt + 1);
    for (auto [u, v] : cut) {
      adj[bel[u]].push_back(bel[v]);
      adj[bel[v]].push_back(bel[u]);
    }
    return std::pair { cnt, std::move(adj) };
  }
};
```

7.3 点双连通分量

```
struct VDCC {
  int n, cur;
  std::vector<std::vector<int>> adj, vdcc;
  std::vector<int> dfn, low, stk, cut;

  VDCC() = default;
  VDCC (int n) { init(n); }

  void init(int n) {
    this -> n = n;
    cur = 0;
    adj.assign(n + 1, {});
    dfn.assign(n + 1, 0);
    low.assign(n + 1, 0);
    cut.assign(n + 1, 0);
    vdcc.clear();
    stk.clear();
  }

  void add(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
```

```
  void dfs(int u, int fa) {
    int son = 0;
    stk.push_back(u);
    low[u] = dfn[u] = ++cur;
    for (auto v : adj[u]) {
      if (dfn[v] == 0) {
        son += 1;
        dfs(v, u);
        low[u] = std::min(low[u], low[v]);
        if(low[v] >= dfn[u]) {
          if (fa != 0) cut[u] = 1;
          std::vector<int> cc { u };
          int x;
          do {
            cc.push_back(x = stk.back());
            stk.pop_back();
          } while(x != v);
          vdcc.push_back(cc);
        }
      } else if (v != fa) {
        low[u] = std::min(low[u], dfn[v]);
      }
    }

    if (fa == 0 && son == 0) vdcc.push_back({u});
    if (fa == 0 && son >= 2) cut[u] = 1;
  }

  void work() {
    for (int i = 1; i <= n; ++i) {
      if (dfn[i] == 0) dfs(i, 0);
    }
  }

  auto getTree() {
    work();
    int m = vdcc.size();
    std::vector<std::vector<int>> adj(n + m + 1);
    for (int i = 1; i <= m; ++i) {
      for (auto x : vdcc[i - 1]) {
        adj[x].push_back(i + n);
        adj[i + n].push_back(x);
      }
    }
    return std::pair { m, std::move(adj) };
  }
};
```

8 Virtual Tree

```cpp
struct VirtualTree {
  int n, tot;
  std::vector<std::vector<int>> adj, st;
  std::vector<int> fa, dep, dfn, siz;

  VirtualTree (int n) { init(n); }

  void init (int n) {
    this -> n = n;
    tot = 0;
    adj.assign(n + 1, {});
    st.assign(std::__lg(n) + 1, std::vector<int> (n + 1));
    fa.assign(n + 1, 0);
    dep.assign(n + 1, 0);
    dfn.assign(n + 1, 0);
    siz.assign(n + 1, 0);
  }

  void add (int u, int v) {
    adj[u].emplace_back(v);
    adj[v].emplace_back(u);
  }

  void dfs (int u) {
    if(fa[u] != 0) adj[u].erase(find(adj[u].begin(), adj[u].end(), fa[u]));
    siz[u] = 1;
    st[0][dfn[u] = ++tot] = u;
    for(auto v : adj[u]) {
      dep[v] = dep[fa[v] = u] + 1;
      dfs(v);
      siz[u] += siz[v];
    }
  }

  int merge (int x, int y) {
    return dep[x] < dep[y] ? x : y;
  }

  void work (int rt = 1) {
    dep[rt] = 1; dfs(rt);
    for(int i = 1; i <= std::__lg(n); ++i) {
      for(int j = 1; j + (1 << i) - 1 <= n; ++j) {
        st[i][j] = merge(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
      }
    }
  }

  int lca (int u, int v) {
    if(u == v) return u;
    u = dfn[u], v = dfn[v];
    if(u > v) std::swap(u, v);
    int len = std::__lg(v - u);
    return fa[merge(st[len][u + 1], st[len][v - (1 << len) + 1])];
  }
```

```
  template<typename Iter, typename Func>
  int build(Iter l, Iter r, Func&& link) {
    std::vector p(l, r);
    std::sort(p.begin(), p.end(), [&](int x, int y) {
      return dfn[x] < dfn[y];
    });

    std::vector<int> stk;
    stk.push_back(1);

    int len = p.size(), x = p[0] == 1;
    for (int i = p[0] == 1; i < len; ++i) {
      int u = p[i];
      int w = lca(u, stk.back());
      if (w != stk.back()) {
        while (stk.size() >= 2 && dep[w] <= dep[stk[stk.size() - 2]]) {
          link(stk[stk.size() - 2], stk.back());
          x |= stk[stk.size() - 2] == 1;
          stk.pop_back();
        }

        if (stk.back() != w) {
          link(w, stk.back());
          stk.pop_back();
          stk.push_back(w);
        }
      }
      stk.push_back(u);
    }

    len = stk.size();
    for (int i = !x; i < len - 1; ++i) {
      link(stk[i], stk[i + 1]);
    }
    return stk[!x];
  }
};
```

9 最大流

```
template<typename T>
struct Flow {
  struct edge {
    int v; T cap;
    edge(int v, T cap) : v(v), cap(cap) {}
  };

  int n;
  std::vector<edge> e;
  std::vector<std::vector<int>> g;
  std::vector<int> cur, h;

  Flow() = default;
  Flow(int n) { init(n); }

  void init(int n) {
```

```
  this -> n = n;
  g.assign(n + 1, {});
}

void add(int u, int v, T cap) {
  g[u].push_back(e.size());
  e.emplace_back(v, cap);
  g[v].push_back(e.size());
  e.emplace_back(u, 0);
}

bool bfs(int s, int t) {
  std::queue<int> que;
  h.assign(n + 1, 0);
  h[s] = 1;
  que.push(s);
  while (!que.empty()) {
    int u = que.front();
    que.pop();
    for (auto i : g[u]) {
      auto [v, cap] = e[i];
      if (cap > 0 && h[v] == 0) {
        h[v] = h[u] + 1;
        if (v == t) return true;
        que.push(v);
      }
    }
  }
  return false;
}

T dfs(int u, int t, T f) {
  if (u == t) return f;
  T r = f;
  for (int& i = cur[u]; i < (int) g[u].size(); ++i) {
    int j = g[u][i];
    auto& [v, cap] = e[j];
    if (cap > 0 && h[v] == h[u] + 1) {
      T aug = dfs(v, t, std::min(r, cap));
      r -= aug;
      e[j].cap -= aug;
      e[j ^ 1].cap += aug;
      if (r == 0) break;
    }
  }
  return f - r;
}

T flow(int s, int t) {
  T ans = 0;
  while (bfs(s, t)) {
    cur.assign(n + 1, 0);
    ans += dfs(s, t, std::numeric_limits<T>::max());
  }
  return ans;
```

```
  }

  std::vector<int> cut() {
    std::vector<int> x(n + 1);
    for (int i = 1; i <= n; ++i) {
      x[i] = h[i] != 0;
    }
    return x;
  }

  using Edge = std::tuple<int, int, T, T>;
  auto edges() -> std::vector<Edge> {
    std::vector<Edge> E;
    for (int i = 0; i < (int) e.size(); i += 2) {
      E.emplace_back(
        e[i + 1].v,
        e[i].v,
        e[i].cap + e[i + 1].cap,
        e[i + 1].cap
      );
    }
    return E;
  }
};
```

10 费用流

```
template< typename T, typename F>
struct CostFlow {
  struct edge {
    int v; T cap; F cost;
    edge(int v, T cap, F cost) : v(v), cap(cap), cost(cost) {}
  };

  int n;
  std::vector<edge> e;
  std::vector<std::vector<int>> g;
  std::vector<F> h, dp;
  std::vector<int> pre;

  CostFlow() = default;
  CostFlow(int n) { init(n); }

  void init(int n) {
    this -> n = n;
    g.assign(n + 1, {});
    h.assign(n + 1, 0);
    dp.assign(n + 1, 0);
    pre.assign(n + 1, 0);
  }

  void add(int u, int v, T cap, F cost) {
    g[u].push_back(e.size());
    e.emplace_back(v, cap, cost);
    g[v].push_back(e.size());
    e.emplace_back(u, 0,  -cost);
```

```
}

bool dijkstra(int s, int t) {
  using node = std::pair<F, int>;
  fill( dp.begin(),  dp.end(), std::numeric_limits<F>::max());
  fill(pre.begin(), pre.end(), -1);

  std::priority_queue<node, std::vector<node>, std::greater<node>> q;
  dp[s] = 0;
  q.emplace(dp[s], s);
  while (!q.empty()) {
    auto [w, u] = q.top();
    q.pop();
    if (dp[u] != w) continue;
    for (auto i : g[u]) {
      auto [v, cap, cost] = e[i];
      if (cap > 0 && dp[u] + h[u] - h[v] + cost < dp[v]) {
        dp[v] = dp[u] + h[u] - h[v] + cost;
        pre[v] = i;
        q.emplace(dp[v], v);
      }
    }
  }
  return dp[t] != std::numeric_limits<F>::max();
}

std::pair<T, F> flow(int s, int t) {
  T flow = 0;
  F cost = 0;
  while (dijkstra(s, t)) {
    for (int i = 1; i <= n; ++i) {
      h[i] += dp[i];
    }
    // 最小费用可行流：if (g[t] >= 0) break;
    T aug = std::numeric_limits<T>::max();
    for (int u = t; u != s; u = e[pre[u] ^ 1].v) {
      aug = std::min(aug, e[pre[u]].cap);
    }
    for (int u = t; u != s; u = e[pre[u] ^ 1].v) {
      e[pre[u]].cap -= aug;
      e[pre[u] ^ 1].cap += aug;
    }
    flow += aug;
    cost += aug * h[t];
  }

  return std::pair(flow, cost);
}

using Edge = std::tuple<int, int, T, T, F>;
auto edges() -> std::vector<Edge> {
  std::vector<Edge> E;
  for(int i = 0; i < (int)e.size(); i += 2) {
    E.emplace(
      e[i + 1].v,
```

```
        e[i].v,
        e[i].cap + e[i + 1].cap,
        e[i + 1].cap,
        e[i].cost
      );
    }
    return E;
  }
};
```

11 二分图

11.1 二分图判定

一个图为二分图当且仅当不存在奇环。可以通过染色判断。当然，如果不需要二分图，只需要判断是否存在奇环，也是可以用染色来做的。

11.2 二分图匹配

求解：考虑网络流，连边：$S \to L \to R \to T$。时间复杂度 $O(m\sqrt{n})$。使用最大流中的 `edges()` 函数找到 $L \to R$ 的边可以找到构造方案。

11.3 最小点覆盖

定义：每条边都存在至少一个端点被选择。

König 定理：最大匹配数等于最小点覆盖数。

11.4 最大独立集

性质：对于一般图： $C \subset V$ 是点覆盖当且仅当 $V - C$ 是独立集。

所以最大独立集 = $|V|$ - 最小点覆盖。

11.5 最小路径覆盖

定义：选择最少的路径，使得每个点都出现在恰好一条路径中。

由于一个点恰好出现在一条路径上，我们给路径的起点连上源点，终点连上汇点。这样每个点都是恰好 1 入度，1 出度。

考虑拆点。对于每个点，拆成 u_{in} 和 u_{out}，然后对于每条边 (u, v)，考虑在二分图上连边：$u_{\text{out}} \to v_{\text{in}}$ 这个时候最大匹配就是路径上边的数量，所以最小路径覆盖就是 $n - |f(g)|$。

如果一个点可以出现在多条路径中呢？传递闭包一下即可。

12 最小斯坦纳树

```
vector dp(n + 1, vector<i64> (1 << k, inf));
for (int i = 0; i < k; ++i) {
  dp[node[i]][1 << i] = 0;
}
for (int s = 1; s < (1 << k); ++s) {
  using node = std::pair<int, i64>;
  std::vector<int> vis(n + 1);
  std::priority_queue<node, std::vector<node>, std::greater<node>> q;
  for (int i = 1; i <= n; ++i) {
    for (int t = (s - 1) & s; t; t = (t - 1) & s) {
      dp[i][s] = min(dp[i][s], dp[i][s ^ t] + dp[i][s]);
```

```
    }
    q.emplace(dp[i][s], i);
  }
  while (!q.empty()) {
    auto [_, u] = q.top();
    q.pop();
    if (vis[u]) continue;
    vis[u] = 1;
    for (auto [v, w] : adj[u]) {
      if (dp[v][s] < dp[u][s] + w) {
        dp[v][s] = dp[u][s] + w;
        q.emplace(dp[v][s], v);
      }
    }
  }
}
```

13 Kruskal 重构树