XOR. Es bezeichne $c(n)$ für $n \in \mathbb{N}$ die minimalen Kosten für die Einrichtung eines Rechenzentrums mit genau n Servern. Sei außerdem

$$\begin{aligned} \mathrm{lsb}(n) &= \max\{k \geq 0 \mid 2^k \text{ teilt } n\} \\ \mathrm{msb}(n) &= \max\{k \geq 0 \mid 2^k \leq n\} \end{aligned}$$

für alle $n \in \mathbb{N}$ das niedrigste gesetzte Bit (engl. *Least Significant Bit*) bzw. das höchste gesetzte Bit (engl. *Most Significant Bit*) der Zahl n. Zudem schreiben wir $\lg(n)$ verkürzend für $\log_2(n)$, den Logarithmus zur Basis 2.

Teil (i). Es gibt $\binom{5}{2} = \frac{5 \cdot 4}{2} = 10$ mögliche Verbindungen zwischen 5 Servern und somit $2^{10} = 1024$ mögliche Netzwerke. Die vollständige Suche mit einem Programm führt zum Ziel. Diese Lösung bringt allerdings kaum Erkenntnisse und es gibt eine Alternative ohne Computereinsatz. Mindestens einer der Server 0, 1, 2 oder 3 muss mit Server 4 verbunden sein. Wegen $\mathrm{msb}(4) = 2$ und $\mathrm{msb}(k) \leq 1$ für $1 \leq k \leq 3$ ist $\mathrm{msb}(4 \oplus k) = 2$ für $0 \leq k \leq 3$, eine Verbindung zu Server 4 kostet also mindestens $2^2 = 4$ Euro, wobei die Verbindung $0 \leftrightarrow 4$ genau 4 Euro kostet. Insgesamt braucht man mindestens 4 Verbindungen, damit jeder Server jeden anderen Server erreichen kann. Wegen $2 \oplus 3 = 1$ und $0 \oplus 1 = 1$ gibt es zwei Verbindungen, die sich für 1 Euro einrichten lassen. Alle anderen Verbindungen sind teurer, kosten also 2 Euro oder mehr. Für genau 2 Euro gibt es die Verbindung $0 \leftrightarrow 2$. Mit $0 \leftrightarrow 1$, $2 \leftrightarrow 3$, $0 \leftrightarrow 2$ und $0 \leftrightarrow 4$ sind nun 4 Verbindungen gefunden, die sich zu den minimalen Kosten von $c(5) = 1 + 1 + 2 + 4 = 8$ Euro einrichten lassen (vgl. Abbildung 1).

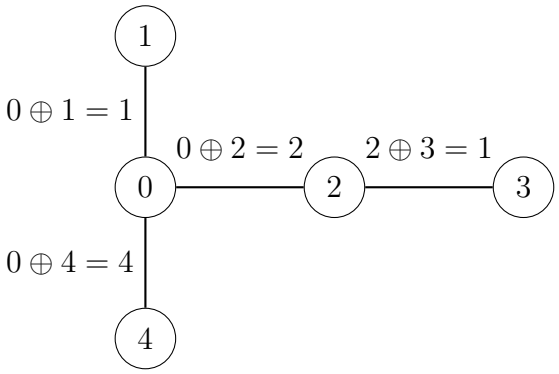

Abbildung 1: Günstigstes Netzwerk für $n = 5$ Server

Teil (ii). Wir richten die Verbindungen $0 \leftrightarrow 1$, $1 \leftrightarrow 2$, $2 \leftrightarrow 3$, …, $n-2 \leftrightarrow n-1$ ein, d. h. $k-1 \leftrightarrow k$ für $1 \leq k \leq n-1$. Die Kosten sind dann durch

$$f(n) = \sum_{k=1}^{n-1} (k-1) \oplus k$$

gegeben. Unser Ziel ist, $f(n) \leq 3n \lg(n) + 10$ für alle $n \in \mathbb{N}$ zu beweisen. Dazu ist das folgende Lemma für den Fall, dass $n-1$ eine Zweierpotenz ist, hilfreich.

Lemma 1. Sei $m \geq 0$ eine ganze Zahl. Dann gilt $f(2^m + 1) = (m + 1) \cdot 2^m$.

Beweis. Vollständige Induktion über m, der Induktionsanfang $m = 0$ ist klar. Nach Induktionsvoraussetzung gelte die Aussage für ein $m \geq 0$, für $m + 1$ folgt dann[1]:

$$\begin{aligned} f(2^{m+1} + 1) &= \sum_{k=1}^{2^{m+1}} (k-1) \oplus k = \sum_{k=1}^{2^m} (k-1) \oplus k + \sum_{k=2^m+1}^{2^{m+1}} (k-1) \oplus k \\ &= f(2^m + 1) + \sum_{k=2^m+1}^{2^{m+1}-1} (k-1) \oplus k + (2^{m+1} - 1) \oplus 2^{m+1} \\ &= 2 \cdot (m+1) \cdot 2^m - (2^{m+1} - 1) + (2^{m+2} - 1) = (m+2) \cdot 2^{m+1} \end{aligned}$$

Damit ist der Induktionsschritt vollständig und die Aussage bewiesen. □

Sei nun $m \geq 0$ so gewählt, dass $2^{m-1} < n - 1 \leq 2^m$ gilt. Dann ist $2(n-1) > 2^m$ und damit $1 + \lg(n) > 1 + \lg(n-1) > m$. Es folgt also

$$\begin{aligned} f(n) \leq f(2^m + 1) &= (m+1) \cdot 2^m < (2 + \lg(n)) \cdot 2(n-1) \\ &< 2n(2 + \lg(n)) = 2n \lg(n) + 4n \overset{(\star)}{\leq} 3n \lg(n) + 10, \end{aligned}$$

wobei die Ungleichung $(\star)$ sich für $n \leq 15$ händisch verifizieren lässt und für $n \geq 16$ dann $3n \lg(n) + 10 > 2n \lg(n) + n \lg(n) \geq 2n \lg(n) + n \lg(16) = 2n \lg(n) + 4n$ gilt. Der Beweis ist damit abgeschlossen und die Teilaufgabe gelöst.

Teil (iii). Die Eingaben sind Gruppen unterschiedlicher Schwierigkeit zugeordnet. Folgend werden die Lösungen daher in aufsteigender Effizienz präsentiert.

Erste Lösung. Wenn $n \leq 5$ (Testgruppe 1) gilt, dann können wir alle möglichen Netzwerke durchtesten, jeweils die Kosten berechnen und die minimalen Kosten ausgeben. Der Ansatz hat eine Laufzeit in $\mathcal{O}(n^2 \cdot 2^{\frac{n(n-1)}{2}})$, was bereits für $n \geq 8$ sehr lange Rechenzeiten verlangt. Alternativ lassen sich per Hand durch Probieren optimale Lösungen finden und direkt ausgeben.

Zweite Lösung. Es ist Zeit anzumerken, dass sich hinter dieser Aufgabe das Problem verbirgt, einen minimalen Spannbaum zu bestimmen. Minimale Spannbäume sind zentrale Objekte der Graphentheorie und es gibt verschiedene effiziente Algorithmen, um diese zu berechnen. In diesem Fall ist der Algorithmus von Kruskal geeignet, dessen Vorgehensweise sich wie folgt beschreibt:

- Erzeuge eine Liste L mit allen möglichen Verbindungen und sortiere diese in aufsteigender Reihenfolge nach ihren Kosten.
- Iteriere über die Liste L und füge die aktuelle Verbindung nur dann hinzu, wenn dadurch kein Zyklus entsteht.

[1] Der Leser mag sich fragen, wie ein Ausdruck der Form $a \oplus b + c$ auszuwerten ist. Per Konvention sei hier festgelegt, dass XOR stärker als Addition bindet. Es gilt also $a \oplus b + c = (a \oplus b) + c$.

Ein Zyklus liegt zum Beispiel dann vor, wenn es für drei Server A, B und C die Verbindungen $A \leftrightarrow B$, $B \leftrightarrow C$ sowie $C \leftrightarrow A$ gibt. Das gilt analog für Zyklen mit mehr als drei Servern. Auf den ersten Blick erscheint es schwierig, beim Hinzufügen einer Verbindung zu entscheiden, ob dadurch ein Zyklus entsteht. Mit der raffinierten Union-Find-Struktur ist dieses Problem aber effizient gelöst. Darunter verstehen wir eine Datenstruktur, die auf einem Wald folgende Operationen unterstützt:

- $\textsc{init}(n)$: Erzeuge einen Wald aus n Bäumen, die jeweils einzelne Knoten sind.
- $\textsc{find}(u)$: Gebe die Wurzel desjenigen Baums zurück, in dem sich u befindet.
- $\textsc{union}(u,\ v)$: Verbinde die Bäume mit den Wurzeln u und v durch $u \leftrightarrow v$.

Um auf die Entstehung eines Zyklus zu überprüfen, schauen wir für die Endknoten u und v der aktuellen Verbindung, ob $\textsc{find}(u) = \textsc{find}(v)$ gilt. Ist die Gleichheit erfüllt, sind u und v bereits im selben Baum und eine neue Verbindung erzeugt einen Zyklus. Beim Hinzufügen einer neuen Verbindung rufen wir $\textsc{union}(\textsc{find}(u),\ \textsc{find}(v))$ auf, um die Union-Find-Struktur zu aktualisieren. Implementieren lässt sich die Union-Find-Struktur sehr effizient, sodass die Laufzeit des Algorithmus durch das Sortieren der Verbindungen dominiert wird. Da es $\mathcal{O}(n^2)$ Verbindungen gibt, liegt die Laufzeit in $\mathcal{O}(n^2 \lg n)$. Das genügt für $n \leq 3000$ bzw. die Testgruppen 2 und 3.

Dritte Lösung. Ist $n \approx 10^5$, dann gibt es etwa $5 \cdot 10^9$ Verbindungen und es ist zu ineffizient, diese explizit zu generieren. Stattdessen erkennt man einige Muster durch scharfes Hinsehen und kann den folgenden zielführenden Satz formulieren.

Satz 1. Es gilt $c(n) = \sum_{k=1}^{n-1} 2^{\operatorname{lsb}(k)}$ für alle $n \in \mathbb{N}$.

Beweis. Wir beweisen die Aussage mit starker Induktion über n. Für $n = 1$ stimmt sie offensichtlich, der Induktionsanfang ist erfüllt. Nach Induktionsvoraussetzung gelte die Aussage für alle natürlichen Zahlen $\leq n$, wir zeigen sie für $n + 1$. Sei m so gewählt, dass $2^m \leq n < 2^{m+1}$ gilt. Wir teilen die Server nach ihren Nummern in zwei Gruppen: $S_1 = \{k \in \mathbb{N} \mid 0 \leq k < 2^m\}$ und $S_2 = \{k \in \mathbb{N} \mid 2^m \leq k \leq n\}$. Seien $u \in S_1$ und $v \in S_2$ die Nummern zweier Server. Wegen $\operatorname{msb}(u) < m$ und $\operatorname{msb}(v) = m$ gilt $u \oplus v \geq 2^m$. Tatsächlich gibt es die Verbindung $0 \leftrightarrow 2^m$ mit Kosten $0 \oplus 2^m = 2^m$. Es muss auch mindestens eine Verbindung zwischen S_1 und S_2 geben, sonst kann nicht jeder Server jeden anderen Server erreichen. Nach Induktionsvoraussetzung sind die minimalen Kosten bekannt, nur die Server aus S_1 bzw. S_2 miteinander zu verbinden. Insgesamt entsteht so ein zusammenhängendes Netzwerk mit den Kosten

$$c(n) = 2^m + \sum_{k=1}^{2^m-1} 2^{\operatorname{lsb}(k)} + \sum_{k=1}^{n-2^m} 2^{\operatorname{lsb}(k)} = \sum_{k=1}^{2^m} 2^{\operatorname{lsb}(k)} + \sum_{k=2^m+1}^{n} 2^{\operatorname{lsb}(k)} = \sum_{k=1}^{n} 2^{\operatorname{lsb}(k)}$$

und das ist die zu beweisene Aussage. Man beachte, dass wir eine Reindexierung der Server in S_2 vorgenommen haben. Das ist möglich, weil stets $\operatorname{msb}(u) = m$ für $u \in S_2$ gilt und wegen $2^m \oplus 2^m = 0$ das m-te Bit in den Kosten sämtlicher Verbindungen gleich 0 ist, also ignoriert werden kann. Damit ist der Satz bewiesen. □

Um die Testfälle mit $n \leq 10^5$ (Testgruppe 4) zu lösen, genügt es, die Summe $\sum_{k=1}^{n-1} 2^{\mathrm{lsb}(k)}$ aus dem Satz direkt auszuwerten. Das gelingt in Zeit $\mathcal{O}(n \lg n)$ und kann mit bitweisen Operationen schnell und übersichtlich implementiert werden.

Vierte Lösung. Schließlich sind die Testfälle mit $n \leq 10^{12}$ (Testgruppe 5) durch eine geschickte Berechnung der genannten Summe zu bezwingen. Dafür werden hier zwei verschiedene Lösungsideen vorgestellt. Zunächst beweisen wir den folgenden Satz.

Satz 2. Es gilt $c(n) = \sum_{k=0}^{\lfloor \lg(n-1) \rfloor} 2^k \left\lfloor \frac{n+2^k-1}{2^{k+1}} \right\rfloor$ für alle $n \in \mathbb{N}$.

Beweis. Sei $g(k)$ die Anzahl der Zahlen $1 \leq m \leq n-1$ mit $\mathrm{lsb}(m) = k$. Dann gilt

$$c(n) = \sum_{k=1}^{n-1} 2^{\mathrm{lsb}(k)} = \sum_{k=1}^{\infty} g(k) \cdot 2^k = \sum_{k=1}^{\lfloor \lg(n-1) \rfloor} g(k) \cdot 2^k,$$

wobei das letzte Gleichheitszeichen wegen $g(k) = 0$ für $k > \lfloor \lg(n-1) \rfloor$ steht, denn $\mathrm{lsb}(m) > \lfloor \lg(n-1) \rfloor$ impliziert $\mathrm{lsb}(m) > \lg(n-1)$, die Zahl $n-1$ hat aber nicht mehr als $1 + \lfloor \lg(n-1) \rfloor$ Bits und daher ist $\mathrm{lsb}(m) \leq \lfloor \lg(n-1) \rfloor < \lg(n-1)$, ein Widerspruch. Um $g(k)$ zu berechnen, stellt man zunächst fest, dass jedes m mit $\mathrm{lsb}(m) = k$ ein Vielfaches von 2^k ist und $\mathrm{lsb}(2^k) = k$ gilt. Ist m ein gerades Vielfaches von 2^k, d. h. $m = 2\ell \cdot 2^k$, mit $\ell \in \mathbb{N}$, dann ist sogar $m = \ell \cdot 2^{k+1}$, also $\mathrm{lsb}(m) \geq k+1$. Sei also m ein ungerades Vielfaches von 2^k, d. h. $m = (2\ell-1) \cdot 2^k$ mit $\ell \in \mathbb{N}$. Dann ist tatsächlich immer $\mathrm{lsb}(m) = k$. Wir behaupten, dass $\ell_{max} = \left\lfloor \frac{n+2^k-1}{2^{k+1}} \right\rfloor$ das größte ℓ ist, sodass $(2\ell-1) \cdot 2^k \leq n-1$ gilt. Die Ungleichung ist leicht zu überprüfen:

$$(2\ell_{max} - 1) \cdot 2^k \overset{(\star)}{\leq} \left(\frac{n+2^k-1}{2^k} - 1 \right) \cdot 2^k = n + 2^k - 1 - 2^k = n - 1.$$

Da bei $(\star)$ Gleichheit eintreten kann und die linke Seite in ℓ_{max} streng monoton steigend ist, ist ℓ_{max} auch das größte ℓ. Wegen $\ell \in [1, \ell_{max}]$ gibt ebendiese Zahl zugleich die Anzahl möglicher Werte für ℓ an, was wiederum gleich der Anzahl möglicher Werte für m – also $g(k)$ – ist und damit ist der Satz bewiesen. □

Die neue Formel ermöglicht eine Berechnung von $c(n)$ mit Laufzeit in $\mathcal{O}(\lg n)$. Das ist nicht nur eine bemerkenswerte Verbesserung gegenüber den bereits vorgestellten Lösungen, sondern lässt sich in wenigen Zeilen Quellcode mühelos implementieren.

Fünfte Lösung. Einen anderen Zugang zur effizienten Berechnung der Summe bieten Rekursionsgleichungen. Wir beweisen einen weiteren Satz.

Satz 3. Für alle $n \in \mathbb{N}$ mit $n \geq 2$ gelten folgende Rekursionsgleichungen:

$$c(2n) = 1 + c(2n-1), \tag{1}$$

$$c(2n+1) = n + 2 \cdot c(n+1). \tag{2}$$

Beweis. Gleichung (1) ist klar, denn $(2n-2) \oplus (2n-1) = 1$. Für Gleichung (2) bemerkt man zunächst, dass

$$c(2n+1) = n + 2 \cdot c(n+1) \Leftrightarrow \sum_{k=1}^{2n} 2^{\mathrm{lsb}(k)} = n + 2 \sum_{k=1}^{n} 2^{\mathrm{lsb}(k)}$$

nach Satz 1 gilt und das ist nach einfachen Umformungen zu

$$\sum_{k=n+1}^{2n} 2^{\mathrm{lsb}(k)} = n + \sum_{k=1}^{n} 2^{\mathrm{lsb}(k)}$$

äquivalent. Wir beweisen diese Gleichung mit vollständiger Induktion. Mit dem Fall $n = 1$ ist der Induktionsanfang klar. Die Gleichung gelte nun für ein n, wir zeigen sie für $n+1$ durch folgende Umformungen:

$$\begin{aligned}
\sum_{k=n+2}^{2n+2} 2^{\mathrm{lsb}(k)} &= \sum_{k=n+1}^{2n} 2^{\mathrm{lsb}(k)} + 2^{\mathrm{lsb}(2n+1)} + 2^{\mathrm{lsb}(2n+2)} - 2^{\mathrm{lsb}(n+1)} \\
&= n + \sum_{k=1}^{n} 2^{\mathrm{lsb}(k)} + 1 + 2 \cdot 2^{\mathrm{lsb}(n+1)} - 2^{\mathrm{lsb}(n+1)} \\
&= n + 1 + \sum_{k=1}^{n+1} 2^{\mathrm{lsb}(k)}
\end{aligned}$$

Der letzte Ausdruck ist die zu beweisene Gleichung und damit sind wir fertig. □

Mit den Rekursionsgleichungen gelingt die Berechnung der minimalen Kosten auch in $\mathcal{O}(\lg n)$ Zeit und es bietet sich eine rekursive Implementierung an.

Bonus. Die fünfte Lösung bringt sogar weitere Erkenntnisse über das Wachstum von $c(n)$. Der Leser ist eingeladen, als Übung zu beweisen, dass für alle $k \in \mathbb{N}$ gilt:

$$c(2^k) = k \cdot 2^{k-1}$$

Sei $2^{k-1} < n \leq 2^k$. Dann können wir die minimalen Kosten $c(n)$ durch

$$c(n) \leq c(2^k) = k \cdot 2^{k-1} < (1 + \lg n) \cdot n = n \lg n + n \in \mathcal{O}(n \lg n)$$

abschätzen. Die tatsächlichen minimalen Kosten sind mindestens um den Faktor 3 kleiner als in Teil (ii) zu zeigen war.

Auf den folgenden Seiten befinden sich Implementierungen der Lösungen in C++.

Erste Lösung (Brute-Force) mit Bitmasks und Tiefensuche.

```cpp
#include <bits/stdc++.h>
#define INF 1000000000
typedef long long ll;
using namespace std;

ll solve(ll n) {
    ll curId = 0;
    ll edgeId[n][n];
    for (ll i = 0; i < n; ++i) {
        for (ll j = i + 1; j < n; ++j) {
            edgeId[i][j] = edgeId[j][i] = curId++;
        }
    }

    ll rs = INF;
    bool vis[n];
    for (ll mask = 0; mask < (1LL << (n * (n - 1) / 2)); ++mask) {
        ll cost = 0;
        vector<ll> graph[n];
        for (ll i = 0; i < n; ++i) {
            for (ll j = i + 1; j < n; ++j) {
                if (mask & (1 << edgeId[i][j])) {
                    graph[i].push_back(j);
                    graph[j].push_back(i);
                    cost += i ^ j;
                }
            }
        }

        stack<ll> s; s.push(0);
        fill(vis, vis + n, 0);
        while (!s.empty()) {
            ll u = s.top(); s.pop();
            vis[u] = true;
            for (ll v : graph[u]) {
                if (!vis[v]) s.push(v);
            }
        }

        bool ok = true;
        for (ll i = 0; i < n; ++i) ok &= vis[i];
        if (ok) rs = min(rs, cost);
    }

    return rs;
}

int main() {
    ll n; cin >> n;
    cout << solve(n) << endl;
    return 0;
}
```

Zweite Lösung (Spannbaum) mit Kruskal und Union-Find-Struktur.

```cpp
#include <bits/stdc++.h>
typedef long long ll;
#define MAX_N 3000
using namespace std;

ll dsu[MAX_N];
ll dsusz[MAX_N];

ll dsu_find(ll u) {
    while (dsu[u] != u) u = dsu[u];
    return u;
}

void dsu_union(ll u, ll v) {
    ll ru = dsu_find(u);
    ll rv = dsu_find(v);
    if (ru == rv) return;
    if (dsusz[ru] > dsusz[rv]) swap(ru, rv);
    dsu[ru] = rv; dsusz[rv] += dsusz[ru];
}

ll solve(ll n) {
    for (ll i = 0; i < n; ++i) {
        dsu[i] = i; dsusz[i] = 1;
    }

    vector<tuple<ll, ll, ll>> edges;
    for (ll i = 0; i < n; ++i) {
        for (ll j = i + 1; j < n; ++j) {
            edges.push_back(make_tuple(i ^ j, i, j));
        }
    }

    ll rs = 0;
    sort(edges.begin(), edges.end());
    for (ll i = 0; i < edges.size(); ++i) {
        ll u = get<1>(edges[i]);
        ll v = get<2>(edges[i]);
        if (dsu_find(u) != dsu_find(v)) {
            rs += u ^ v;
            dsu_union(u, v);
        }
    }

    return rs;
}

int main() {
    ll n; cin >> n;
    cout << solve(n) << endl;
    return 0;
}
```

Dritte Lösung mit Auswertung einer Summe.

```
#include <bits/stdc++.h>
typedef long long ll;
using namespace std;

ll solve(ll n) {
    ll rs = 0;
    for (ll k = 1; k < n; ++k) {
        ll lsb = 0;
        while (k % (1LL << (lsb + 1)) == 0) ++lsb;
        rs += 1LL << lsb;
    }

    return rs;
}

int main() {
    ll n; cin >> n;
    cout << solve(n) << endl;
    return 0;
}
```

Vierte Lösung mit geschickter Auswertung der Summe.

```
ll solve(ll n) {
    ll rs = 0;
    for (ll k = 0; (1LL << k) < n; ++k) {
        ll q = (n + (1LL << k) - 1) / (1LL << (k + 1));
        rs += q * (1LL << k);
    }

    return rs;
}
```

Fünfte Lösung mit Rekursionsgleichungen zur Auswertung der Summe.

```
ll solve(ll n) {
    if (n == 1) return 0;
    if (n % 2) return n / 2 + 2 * solve(1 + n / 2);
    return 1 + solve(n - 1);
}
```