Teil (i). Christian kann Alice die Geschenke mit den Werten $1, 2, \ldots, 9$ und 15 schenken. Bob erhält die übrigen Geschenke mit den Werten $10, 11, 12, 13$ und 14. Das ist wegen $1+2+\cdots+9+15 = 10+11+12+13+14 = 60$ eine gerechte Aufteilung.

Teil (ii). Es gibt dann und nur dann eine gerechte Aufteilung, wenn n oder $n+1$ durch 4 teilbar ist. Das führt in aufsteigender Reihenfolge auf

$$3, 4, 7, 8, 11, 12, 15, 16, 19, 20, 23, 24, 27, 28, 31, 32, \ldots$$

als mögliche Werte für n, was natürlich zu beweisen ist. Wir zeigen zwei Aussagen:

- Gibt es eine gerechte Aufteilung, dann ist n oder $n+1$ durch 4 teilbar.
- Ist n oder $n+1$ durch 4 teilbar, dann gibt es eine gerechte Aufteilung.

„⇒“ Angenommen, es gibt eine gerechte Aufteilung. Mit S bezeichnen wir den Gesamtwert aller Geschenke, also die Summe der einzelnen Werte:

$$S = 1 + 2 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$$

Damit Alice und Bob jeweils im gleichen Wert $\frac{S}{2}$ beschenkt werden können, muss der Gesamtwert S eine gerade Zahl sein. Dafür muss $n(n+1)$ sogar durch 4 teilbar sein. Jetzt ist n durch 4 teilbar, $n+1$ durch 4 teilbar oder n und $n+1$ sind jeweils nur durch 2 teilbar. Letzterer Fall tritt aber nie ein, denn aufeinanderfolgende Zahlen können niemals beide gerade sein. Also ist n oder $n+1$ durch 4 teilbar.

„⇐“ Sei nun umgekehrt n oder $n+1$ durch 4 teilbar. Es ist zu beweisen, dass es eine gerechte Aufteilung gibt. Wir machen eine Fallunterscheidung und geben für alle solchen Werte von n konkrete gerechte Aufteilungen an.

Fall 1: n ist durch 4 teilbar
Dann hat n die Form $n = 4k$ mit einer positiven ganzen Zahl k. Ordne die Geschenke in aufsteigender Reihenfolge nach ihrem Wert und teile sie folgendermaßen auf. Alice bekommt die ersten k und die letzten k Geschenke, Bob die mittleren $2k$ Geschenke. Die Aufteilung ist gerecht, denn Alice erhält Geschenke im Wert von

$$\begin{aligned}
&\underbrace{1 + 2 + \cdots + k}_{\text{erste } k \text{ Geschenke}} + \underbrace{(3k+1) + (3k+2) + \cdots + (4k-1) + 4k}_{\text{letzte } k \text{ Geschenke}} \\
&= (1 + 4k) + (2 + (4k-1)) + (3 + (4k-2)) + \cdots + (k + (3k+1)) \\
&= k(4k+1)
\end{aligned}$$

und alle Geschenke zusammen haben genau den doppelten Wert

$$S = \frac{n(n+1)}{2} = \frac{4k(4k+1)}{2} = 2k(4k+1),$$

sodass Bob mit dem gleichen Wert wie Alice beschenkt wird. Somit können wir in jedem Fall eine gerechte Aufteilung finden, wenn n durch 4 teilbar ist.

Fall 2: $n+1$ ist durch 4 teilbar
Dann hat n die Form $n = 4k-1$ mit einer positiven ganzen Zahl k. Die Idee ist wie in Fall 1, Alice und Bob erhalten jedoch unterschiedlich viele Geschenke. Ordne die Geschenke wieder in aufsteigender Reihenfolge nach ihrem Wert. Alice bekommt jetzt die ersten $k-1$ und die letzten k Geschenke, deren Gesamtwert sich zu

$$\begin{aligned}&\underbrace{1+2+\cdots+(k-1)}_{\text{erste } k-1 \text{ Geschenke}}+\underbrace{(3k)+(3k+1)+\cdots+(4k-2)+(4k-1)}_{\text{letzte } k \text{ Geschenke}}\\&=(1+(4k-1))+(2+(4k-2))+\cdots+((k-1)+(3k+1))+3k\\&=4k(k-1)+3k=k(4k-1)\end{aligned}$$

berechnet. Alle Geschenke zusammen haben den doppelten Wert

$$S=\frac{n(n+1)}{2}=\frac{(4k-1)4k}{2}=2k(4k-1),$$

sodass Bob mit dem gleichen Wert wie Alice beschenkt wird. Damit ist gezeigt, dass auch dann eine gerechte Aufteilung existiert, wenn $n+1$ durch 4 teilbar ist.

Teil (iii). Die Antwort ist 5 830 034 720, Christian kann bei nur 40 Geschenken eine gerechte Aufteilung aus fast 6 Milliarden Möglichkeiten wählen.

Sei S die Summe der ganzen Zahlen von 1 bis n wie in Teil (ii) und $M=\{1,2,\ldots,n\}$ die Menge dieser Zahlen. Die Elementsumme einer endlichen Menge von Zahlen ist die Summe ihrer Elemente, z. B. ist die Elementsumme von $\{2,3,5,9\}$ gleich $2+3+5+9=19$ und die Elementsumme der leeren Menge $\varnothing$ ist 0. Jetzt gilt folgende Aussage: Die Anzahl der gerechten Aufteilungen ist gleich der Anzahl von Teilmengen von M mit Elementsumme $S/2$. Warum? Es genügt, die Geschenke für Alice auszuwählen. Dann bekommt Bob alle übrigen Geschenke. Jede Teilmenge von M beschreibt eine Wahl der Geschenke für Alice. Aber da die Aufteilung gerecht sein soll, müssen Alice und Bob jeweils Geschenke im gleichen Wert von $S/2$ erhalten.

Unsere neue Fragestellung: Wie viele Teilmengen von M haben Elementsumme $S/2$?

Erste Lösung. Die Idee ist, alle Teilmengen von M explizit aufzuzählen, für jede Teilmenge die Elementsumme zu berechnen und einen Zähler zu erhöhen, wenn sie gleich $S/2$ ist. Das ist ein klassischer Brute-Force-Ansatz. Die Teilmengen von M lassen sich kompakt durch Binärcodes mit n Bits darstellen. Das i-te Bit (von $i=1$ beginnend) gibt an, ob die Zahl i in der zugehörigen Teilmenge enthalten ist.

Beispiel: Ist $M=\{1,2,3,4\}$, dann hat die Teilmenge $\{1,2,3\}$ den Binärcode 0111, die Teilmenge $\{1,4\}$ den Binärcode 1001, $\{3\}$ den Binärcode 0100, $\{1,2,3,4\}$ den Binärcode 1111 und die leere Menge $\varnothing$ den Binärcode 0000.

Offenbar können wir jeder Teilmenge von M einen Binärcode der Länge n zuordnen und umgekehrt an jedem Binärcode der Länge n eine solche Teilmenge ablesen. Insbesondere gibt es genauso viele Teilmengen von M wie Binärcodes der Länge n und die lassen sich leicht abzählen: Pro Bit gibt es zwei Möglichkeiten und wir können alle Bits unabhängig wählen und dann kombinieren, also gibt es $\underbrace{2 \cdot 2 \cdot \dots \cdot 2}_{\text{n Bits}} = 2^n$ solche Binärcodes und damit auch Teilmengen von M. Das ist ein Problem: Ist $n = 40$, dann untersuchen wir $2^{40} > 1\,000\,000\,000\,000$ Teilmengen und müssen jeweils die Elementsumme berechnen. Unsere Computer erledigen etwa 100 Millionen triviale Operationen pro Sekunde, die Berechnung dauert mindestens 2 bis 3 Tage.

Zeitkomplexität der Lösung: $\mathcal{O}(n \cdot 2^n)$

Beispielprogramm in C++:

```
1  int solve(int n) {
2      if (n % 4 != 0 && n % 4 != 3) return 0;
3      int result = 0;
4      for (int mask = 0; mask < (1 << n); ++mask) {
5          int sum = 0;
6          for (int i = 0; i < n; ++i) {
7              if ((mask & (1 << i)) != 0) sum += (i + 1);
8          }
9
10         if (4 * sum == n * (n + 1)) ++result;
11     }
12
13     return result;
14 }
```

Hinweis: Ab etwa $n = 30$ kann es zu Überläufen kommen. Durch 64-Bit-Datentypen sind Überläufe bis etwa $n = 60$ vermeidbar. Danach bedarf es raffinierter Methoden zur Darstellung großer Zahlen im Arbeitsspeicher, die nicht Gegenstand der Aufgabe sind. Dieser Hinweis gilt für alle Programme in dieser Lösung.

Zweite Lösung. Der Brute-Force-Ansatz ist prinzipiell geeignet: Gelingt es uns, nur die Teilmengen mit Elementsumme $S/2$ aufzuzählen, dann sind wir in weniger als einer Minute fertig. Selbstverständlich sind Elementsummen zu berechnen und es ist schwierig, die gesuchten Teilmengen zu charakterisieren. Mit Backtracking und einer guten Heuristik ist die Laufzeit dennoch auf etwa 15 Minuten zu reduzierbar.

Wir entwickeln einen rekursiven Algorithmus. Dieser basiert auf der Idee, für jedes Geschenk zu entscheiden, ob es ausgewählt oder nicht ausgewählt wird und die Anzahlen für beide Fälle zu summieren. Ein Parameter gibt den Bereich $1, 2, \dots, k$ derjenigen Geschenke an, für die noch eine Entscheidung zu treffen ist. Initial ist $k = n$ und mit jedem rekursiven Aufruf reduzieren wir k um 1. Im Basisfall $k = 0$ ist zu überprüfen, ob die Elementsumme gleich $S/2$ ist.

Überdies genügt es, die Summe der Werte der ausgewählten Geschenke als Parameter zu übergeben. Informationen über die einzelnen Geschenke selbst sind irrelevant. In dieser Form benötigt der Algorithmus 2^n Funktionsaufrufe und ist deutlich zu langsam. Die Laufzeit lässt sich durch zwei Heuristiken erstaunlich reduzieren:

- Untere Schranke: Ist die als Parameter übergebene Summe größer als $S/2$, dann kann abgebrochen und 0 zurückgegeben werden.
- Obere Schranke: Ist die als Parameter übergebene Summe plus $k(k+1)/2$ kleiner als $S/2$, dann kann ebenfalls abgebrochen und 0 zurückgegeben werden. In einem solchen Fall ist die Summe $S/2$ nicht mehr zu erreichen – selbst wenn alle noch übrigen Geschenke ausgewählt werden.

Besonders die obere Schranke reduziert die neue Laufzeit auf einen Bruchteil der ursprünglichen Laufzeit. Mit beiden Schranken zusammen gab das Programm auf dem Testsystem für $n = 40$ das korrekte Ergebnis nach etwa 13 Minuten aus.

Beispielprogramm in C++:

```
1  typedef unsigned long long num;
2  num solve(num n, num idx, num sum) {
3      if (idx == 0) {
4          if (4 * sum == n * (n + 1)) return 1;
5          return 0;
6      }
7
8      if (4 * sum > n * (n + 1)
9          || 4 * sum + 2 * idx * (idx + 1) < n * (n + 1)) return 0;
10
11     return solve(n, idx - 1, sum) + solve(n, idx - 1, sum + idx);
12 }
```

Dritte Lösung. Schließlich kommen wir zu einer Lösung, die auf dem Prinzip der dynamischen Programmierung basiert und effizienter als ihre Vorgänger ist. Dafür müssen zwei Voraussetzungen erfüllt sein:

- Optimale Teilstruktur: Eine Lösung für eine Instanz des Problems kann aus den Teillösungen für kleinere Instanzen konstruiert werden.
- Überlappende Teilprobleme: Das Problem lässt sich in kleinere Teilprobleme zerlegen, die redundant sind bzw. mehrfach auftreten.

In der Regel überlegt man sich eine geeignete Zustandsmenge und eine Reihenfolge, in der die Teillösungen für alle Zustände berechnet werden. Dazu darf auf bereits berechnete Zustände zurückgegriffen werden.

Hinweis für den interessierten Leser. Ist V die Zustandsmenge, dann können wir den gerichteten Graphen $G = (V, E)$ mit $E = \{(u, v) \in V^2 \mid u \neq v \wedge u \to v\}$ definieren, wobei $u \to v$ ausdrückt, dass v zur Berechnung von u benötigt wird.

Die Berechnungsreihenfolge der Zustände existiert genau dann, wenn G ein Wald ist. In der Tat ist sie eine topologische Sortierung von der Zustandsmenge V.

Wir stellen eine Rekursionsgleichung auf. Sei $f(k, s)$ die Anzahl der Teilmengen von $\{1, 2, \ldots, k\}$ mit Elementsumme s. Unser Ergebnis wird

$$f\left(n, \frac{n(n+1)}{4}\right)$$

sein. Um $f(k, s)$ zu berechnen, gehen wir wie in der zweiten Lösung vor, betrachten also beide Möglichkeiten, das Geschenk k zu wählen bzw. nicht zu wählen:

- Geschenk k wird gewählt: $f(k-1, s-k)$ Teilmengen von $\{1, 2, \ldots, k-1\}$ haben Elementsumme $s-k$. Da k gewählt wird, ist die Gesamtsumme s.
- Geschenk k wird nicht gewählt: $f(k-1, s)$ Teilmengen von $\{1, 2, \ldots, k-1\}$ haben Elementsumme s, was auch die Lösung für $\{1, 2, \ldots, k\}$ ist.

Der erste Fall gilt nur, wenn $s \geq k$ ist. Sonst ist es nicht möglich, nach der Wahl des Geschenks k noch die Summe s zu erreichen. Die Rekursionsgleichung lautet

$$f(k, s) = f(k-1, s) + \begin{cases} 0 & \text{falls } s < k, \\ f(k-1, s-k) & \text{sonst.} \end{cases}$$

Es fehlen noch die Basisfälle. Ist $k = 0$, dann haben wir $f(0, s) = 0$ für $s > 0$. Für $s = 0$ gilt $f(k, 0) = 1$ wegen der leeren Mengen. Insbesondere ist $f(0, 0) = 1$.

Wie viele Zustände sind zu berechnen? Der Parameter k liegt im Intervall $[0; n]$, es gibt also $n + 1$ verschiedene Werte für k. Der Parameter s ist durch

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

nach oben beschränkt und kann ebenfalls den Wert 0 annehmen. Da die Parameter s und k in allen Kombinationen auftreten, gibt es insgesamt

$$(n+1)\left(\frac{n(n+1)}{2} + 1\right)$$

Zustände und der Rechenaufwand pro Zustand ist in etwa konstant. Für $n = 40$ gibt es 33 661 Zustände. Das Programm läuft in deutlich weniger als einer Sekunde durch und ist damit mehr als 1 000-mal so schnell wie die Backtracking-Lösung und mehr als 200 000-mal so schnell wie die Brute-Force-Lösung. Bemerkenswert ist, dass wir die Teilmengen mit Elementsumme $S/2$ gar nicht explizit konstruieren.

Zeitkomplexität der Lösung: $\mathcal{O}(n^3)$

Zur Implementierung. Zum Speichern der Teillösungen für die Zustände hat sich in der Praxis ein zweidimensionales Array, also eine Tabelle bzw. Matrix, etabliert.

Beispielprogramm in C++:

```cpp
typedef unsigned long long num;
num solve(num n) {
    if (n % 4 != 0 && n % 4 != 3) return 0;

    num dp[n + 1][n * (n + 1) / 2 + 1];

    // Basisfälle
    for (num k = 0; k <= n; ++k) dp[k][0] = 1;
    for (num s = 1; s <= n * (n + 1) / 2; ++s) dp[0][s] = 0;

    // Hauptberechnung
    for (num k = 1; k <= n; ++k) {
        for (num s = 1; s <= n * (n + 1) / 2; ++s) {
            dp[k][s] = dp[k - 1][s];
            if (s >= k) dp[k][s] += dp[k - 1][s - k];
        }
    }

    return dp[n][n * (n + 1) / 4];
}
```

Bemerkung. Diese Aufgabe ist eine Spezialfall des enumerativen Partitionsproblems, für das bislang kein effizienter Algorithmus bekannt ist. Das Problem ist $\mathcal{NP}$-schwer, was Richard M. Karp in seiner berühmten Arbeit *Reducibility Among Combinatorial Problems* 1972 für dieses und 20 weitere Probleme beweisen konnte. Tatsächlich ist laut OEIS[1] selbst für diesen Spezialfall keine explizite Formel bekannt.

Alternative Lösung oder Fehler gefunden?
Wir freuen uns über Verbesserungen und neue Lösungsideen.
Kontakt: **itag-goethe@protonmail.com**

[1] URL der Folge: http://oeis.org/A063865