1. **Сијалице**

*Решење:*

Можемо индукцијом показати да ако желимо да угасимо сијалице које образују улазну ниску 1000...00 са N сијалица, онда нам је потребно $2^N$-1 потеза.

Даље можемо наставити познавањем својства Грај кодова или динамичким програмирањем или рекурзивно-похлепним приступом.

Прикажимо решење добијено познавањем својства Грај кодова.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
   ios::sync_with_stdio(false);
   string s; cin >> s;
   long long sol = 0, v = 0;
   for (char x : s)
   {
      v ^= x - '0';
      sol = 2 * sol + v;
   }
   cout << sol << endl;
}
```

Прикажимо решење добијено похлепном стратегијом.

Конструишемо метод `solve` који добија два параметра: позиција `i` која одговара суфиксу ниске сијалица и бинарна цифра `target`. Овај метод враће минимални број потребних потеза да сијалица `i` постигне стање 0 или 1 које памти `target`, и искључује све наредне сијалице. Резултат се може добити позивом метода са `solve(1, 0)`.

Дакле, за метод `solve(i, target)` важи:

- ако почетна вредност сијалице `i` је једнака `target`, довољно је вратити као резултат `solve(i+1, 0)`

- Иначе, најпре позовемо ` solve (i+1, 1)`. Као резултат добијамо број потеза потребан да се достигне стање у ком можемо променити осветљеност сијалице `i`. Дакле, обрада те сијалице је завршена и потребно нам је још $2^(N-i)-1$ потеза да искључимо сијалицу `i+1`.

```
#include <cassert>
#include <cstdio>
#include <iostream>
#include <string>
using namespace std;
const int MAX_N = 50;
int n;
string s;
```

```
long long solve(int i, char target) {
if (i == n-1) {
   if (s[i] == target) return 0;
   else return 1;
}
if (s[i] == target) return solve(i + 1, '0');
return 1 + solve(i + 1, '1') + ((1LL << (n - i - 1)) - 1);
}

int main() {
  cin >> s;
  n = s.size();
  assert(1 <= n && n <= MAX_N);
  for (auto ch : s) {
    assert(ch == '0' || ch == '1');
  }
 cout << solve(0, '0');
}
```

2. **Оптимално множење матрица**

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

using namespace std;

long long minBrojMnozenja(const vector<int>& dimenzije, int l, int d,
                    vector<vector<long long>>& memo) {
 if (memo[l][d] != -1)
  return memo[l][d];
 int n = d - l + 1;
 if (n == 2)
  return 0;
 long long min = numeric_limits<long long>::max();
 for (int i = l+1; i <= d-1; i++) {
  long long broj = minBrojMnozenja(dimenzije, l, i, memo) +
               minBrojMnozenja(dimenzije, i, d, memo) +
               dimenzije[l] * dimenzije[i] * dimenzije[d];
  if (broj < min)
    min = broj;
 }
 return memo[l][d] = min;
}

long long minBrojMnozenja(const vector<int>& dimenzije) {
 int n = dimenzije.size();
 vector<vector<long long>> memo(n);
 for (int i = 0; i < n; i++)
  memo[i].resize(n, -1);
 return minBrojMnozenja(dimenzije, 0, dimenzije.size() - 1, memo);
}
```

```
int main() {
 int n;
 cin >> n;
 vector<int> dimenzije(n);
 for (int i = 0; i < n; i++)
  cin >> dimenzije[i];
 cout << minBrojMnozenja(dimenzije) << endl;
 return 0;
}
```

3. **Обилазак графа**

```
#include<iostream>
using namespace std;
int a,b,c,d,i,j;
int nasao =0;

void obidji(int x, int y, int potez, int granica){
    if (x<0 || y<0 || x>7 || y>7)return;
    if(x==c && y==d) {nasao = 1;   return ;}

   if(potez > granica) return;
    obidji(x-1,y-2,potez + 1,granica);
    obidji(x-2,y-1,potez + 1,granica);
    obidji(x+1,y+2,potez + 1,granica);
    obidji(x+2,y+1,potez + 1,granica);
    obidji(x-1,y+2,potez + 1,granica);
    obidji(x-2,y+1,potez + 1,granica);
    obidji(x+1,y-2,potez + 1,granica);
    obidji(x+2,y-1,potez + 1,granica);
}
int main(){
    cin >> a >> b >> c >> d;
    a--; b--; c--; d--;
    if(a==c && b==d) {cout<< 0 ; return 0;}
    int granica;

    for(granica =0 ; nasao ==0; granica++)
        obidji(a,b,0,granica);

    cout<<granica;

    return 0;

}
```