Rekurencja i poszukiwanie z nawrotami

zajęcia 4.

Marcin Andrychowicz

Rekurencja

Algorytm rekurencyjny:

- rozwiązuje problem przez rozwiązanie pewnej liczby prostszych przypadków tego samego problemu
- zapisujemy za pomocą funkcji rekurencyjnej, czyli wywołującej samą siebie
- przykładem może być omówione wcześniej sortowanie przez scalanie

Rekurencja

Algorytm rekurencyjny:

- rozwiązuje problem przez rozwiązanie pewnej liczby prostszych przypadków tego samego problemu
- zapisujemy za pomocą funkcji rekurencyjnej, czyli wywołującej samą siebie
- przykładem może być omówione wcześniej sortowanie przez scalanie

Rekurencja

Algorytm rekurencyjny:

- rozwiązuje problem przez rozwiązanie pewnej liczby prostszych przypadków tego samego problemu
- zapisujemy za pomocą funkcji rekurencyjnej, czyli wywołującej samą siebie
- przykładem może być omówione wcześniej sortowanie przez scalanie

Liczby Fibonacciego

Definicja

- F_n — n-ta liczba Fibonacciego
- $F_0 = F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$ dla $n > 1$
n	0	1	2	3	4	5	6	7
F_n	1	1	2	3	5	8	13	21
- związek z królikami

Obliczanie F_n — 1. podejście

```
int fib(int n){
    if(n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
```

Liczby Fibonacciego

Definicja

- F_n — n-ta liczba Fibonacciego
- $F_0 = F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$ dla $n > 1$
n	0	1	2	3	4	5	6	7
F_n	1	1	2	3	5	8	13	21
- związek z królikami

Obliczanie F_n — 1. podejście

```
int fib(int n){
    if(n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
```

Liczby Fibonacciego

Definicja

- F_n — n-ta liczba Fibonacciego
- $F_0 = F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$ dla $n > 1$
n	0	1	2	3	4	5	6	7
F_n	1	1	2	3	5	8	13	21
- związek z królikami

Obliczanie F_n — 1. podejście

```
int fib(int n){
    if(n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
```

Liczby Fibonacciego

Definicja

- F_n — n-ta liczba Fibonacciego
- $F_0 = F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$ dla $n > 1$
n	0	1	2	3	4	5	6	7
F_n	1	1	2	3	5	8	13	21
- związek z królikami

Obliczanie F_n — 1. podejście

```
int fib(int n){
    if(n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
```

Liczby Fibonacciego

Definicja

- F_n — n-ta liczba Fibonacciego
- $F_0 = F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$ dla $n > 1$
n	0	1	2	3	4	5	6	7
F_n	1	1	2	3	5	8	13	21
- związek z królikami

Obliczanie F_n — 1. podejście

```
int fib(int n){
    if(n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
```

Liczby Fibonacciego

Definicja

- F_n — n-ta liczba Fibonacciego
- $F_0 = F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$ dla $n > 1$
n	0	1	2	3	4	5	6	7
F_n	1	1	2	3	5	8	13	21
- związek z królikami

Obliczanie F_n — 1. podejście

```
int fib(int n){
    if(n <= 1) return 1;
    return fib(n-1)+fib(n-2);
}
```

Liczby Fibonacciego

Analiza złożoności

- policzmy ile razy wywołamy `fib(0)` lub `fib(1)` licząc `fib(n)`
- licząc `fib(0)` lub `fib(1)` zrobimy jedno wywołanie
- licząc `fib(2)` zrobimy 2 wywołania
- licząc `fib(3)` zrobimy 3 wywołania
- licząc `fib(4)` zrobimy 5 wywołań . . .
- licząc `fib(`n`)` zrobimy F_n wywołań
- spróbujmy jakoś oszacować tą liczbę

Liczby Fibonacciego

Analiza złożoności

- policzmy ile razy wywołamy `fib(0)` lub `fib(1)` licząc `fib(n)`
- licząc `fib(0)` lub `fib(1)` zrobimy jedno wywołanie
- licząc `fib(2)` zrobimy 2 wywołania
- licząc `fib(3)` zrobimy 3 wywołania
- licząc `fib(4)` zrobimy 5 wywołań . . .
- licząc `fib(`n`)` zrobimy F_n wywołań
- spróbujmy jakoś oszacować tą liczbę

Liczby Fibonacciego

Analiza złożoności

- policzmy ile razy wywołamy `fib(0)` lub `fib(1)` licząc `fib(n)`
- licząc `fib(0)` lub `fib(1)` zrobimy jedno wywołanie
- licząc `fib(2)` zrobimy 2 wywołania
- licząc `fib(3)` zrobimy 3 wywołania
- licząc `fib(4)` zrobimy 5 wywołań . . .
- licząc `fib(`n`)` zrobimy F_n wywołań
- spróbujmy jakoś oszacować tą liczbę

Liczby Fibonacciego

Analiza złożoności

- policzmy ile razy wywołamy `fib(0)` lub `fib(1)` licząc `fib(n)`
- licząc `fib(0)` lub `fib(1)` zrobimy jedno wywołanie
- licząc `fib(2)` zrobimy 2 wywołania
- licząc `fib(3)` zrobimy 3 wywołania
- licząc `fib(4)` zrobimy 5 wywołań . . .
- licząc `fib(`n`)` zrobimy F_n wywołań
- spróbujmy jakoś oszacować tą liczbę

Liczby Fibonacciego

Analiza złożoności

- policzmy ile razy wywołamy `fib(0)` lub `fib(1)` licząc `fib(n)`
- licząc `fib(0)` lub `fib(1)` zrobimy jedno wywołanie
- licząc `fib(2)` zrobimy 2 wywołania
- licząc `fib(3)` zrobimy 3 wywołania
- licząc `fib(4)` zrobimy 5 wywołań . . .
- licząc `fib(`n`)` zrobimy F_n wywołań
- spróbujmy jakoś oszacować tą liczbę

Liczby Fibonacciego

Analiza złożoności

- policzmy ile razy wywołamy `fib(0)` lub `fib(1)` licząc `fib(n)`
- licząc `fib(0)` lub `fib(1)` zrobimy jedno wywołanie
- licząc `fib(2)` zrobimy 2 wywołania
- licząc `fib(3)` zrobimy 3 wywołania
- licząc `fib(4)` zrobimy 5 wywołań . . .
- licząc `fib(`n`)` zrobimy F_n wywołań
- spróbujmy jakoś oszacować tą liczbę

Liczby Fibonacciego

Analiza złożoności

- policzmy ile razy wywołamy `fib(0)` lub `fib(1)` licząc `fib(n)`
- licząc `fib(0)` lub `fib(1)` zrobimy jedno wywołanie
- licząc `fib(2)` zrobimy 2 wywołania
- licząc `fib(3)` zrobimy 3 wywołania
- licząc `fib(4)` zrobimy 5 wywołań . . .
- licząc `fib(`n`)` zrobimy F_n wywołań
- spróbujmy jakoś oszacować tą liczbę

Liczby Fibonacciego

Jak szybko rosną?

Liczby Fibonacciego rosną wykładniczo. Dowód:

1. $F_{n+1} > F_n$
2. $F_{n+2} = F_{n+1} + F_n = 2F_n + F_{n-1} \geqslant 2F_n$
3. $F_{2n+2} \geqslant F_{2n+1} \geqslant 2^n F_1 = 2^n$ (czyli F_n rosną co najmniej wykładniczo)
4. $F_{n+1} \leqslant 2F_n$, więc $F_n \leqslant 2^n$ (czyli F_n rosną co najwyżej wykładniczo)

Liczby Fibonacciego

Jak szybko rosną?

Liczby Fibonacciego rosną wykładniczo. Dowód:

1. $F_{n+1} > F_n$
2. $F_{n+2} = F_{n+1} + F_n = 2F_n + F_{n-1} \geqslant 2F_n$
3. $F_{2n+2} \geqslant F_{2n+1} \geqslant 2^n F_1 = 2^n$ (czyli F_n rosną co najmniej wykładniczo)
4. $F_{n+1} \leqslant 2F_n$, więc $F_n \leqslant 2^n$ (czyli F_n rosną co najwyżej wykładniczo)

Liczby Fibonacciego

Jak szybko rosną?

Liczby Fibonacciego rosną wykładniczo. Dowód:

1. $F_{n+1} > F_n$
2. $F_{n+2} = F_{n+1} + F_n = 2F_n + F_{n-1} \geqslant 2F_n$
3. $F_{2n+2} \geqslant F_{2n+1} \geqslant 2^n F_1 = 2^n$ (czyli F_n rosną co najmniej wykładniczo)
4. $F_{n+1} \leqslant 2F_n$, więc $F_n \leqslant 2^n$ (czyli F_n rosną co najwyżej wykładniczo)

Liczby Fibonacciego

Jak szybko rosną?

Liczby Fibonacciego rosną wykładniczo. Dowód:

1. $F_{n+1} > F_n$
2. $F_{n+2} = F_{n+1} + F_n = 2F_n + F_{n-1} \geqslant 2F_n$
3. $F_{2n+2} \geqslant F_{2n+1} \geqslant 2^n F_1 = 2^n$ (czyli F_n rosną co najmniej wykładniczo)
4. $F_{n+1} \leqslant 2F_n$, więc $F_n \leqslant 2^n$ (czyli F_n rosną co najwyżej wykładniczo)

Liczby Fibonacciego

Jak szybko rosną?

Liczby Fibonacciego rosną wykładniczo. Dowód:

1. $F_{n+1} > F_n$
2. $F_{n+2} = F_{n+1} + F_n = 2F_n + F_{n-1} \geqslant 2F_n$
3. $F_{2n+2} \geqslant F_{2n+1} \geqslant 2^n F_1 = 2^n$ (czyli F_n rosną co najmniej wykładniczo)
4. $F_{n+1} \leqslant 2F_n$, więc $F_n \leqslant 2^n$ (czyli F_n rosną co najwyżej wykładniczo)

Liczby Fibonacciego

Jak to poprawić?

- nie trzeba liczyć tych samych wartości wielokrotnie
- możemy zapisywać już policzone wartości w tablicy

Obliczanie F_n — 2. podejście

```
int fib[MAXN+1];
fib[0]=fib[1]=1;
for(int n=2;n<=MAXN;n++)
    fib[n]=fib[n-1]+fib[n-2];
```

Złożoność: $O(n)$

Wniosek: rekurencja nie zawsze jest najlepszą metodą

Liczby Fibonacciego

Jak to poprawić?

- nie trzeba liczyć tych samych wartości wielokrotnie
- możemy zapisywać już policzone wartości w tablicy

Obliczanie F_n — 2. podejście

```
int fib[MAXN+1];
fib[0]=fib[1]=1;
for(int n=2;n<=MAXN;n++)
    fib[n]=fib[n-1]+fib[n-2];
```

Złożoność: $O(n)$

Wniosek: rekurencja nie zawsze jest najlepszą metodą

Liczby Fibonacciego

Jak to poprawić?

- nie trzeba liczyć tych samych wartości wielokrotnie
- możemy zapisywać już policzone wartości w tablicy

Obliczanie F_n — 2. podejście

```
int fib[MAXN+1];
fib[0]=fib[1]=1;
for(int n=2;n<=MAXN;n++)
    fib[n]=fib[n-1]+fib[n-2];
```

Złożoność: $O(n)$

Wniosek: rekurencja nie zawsze jest najlepszą metodą

Liczby Fibonacciego

Jak to poprawić?

- nie trzeba liczyć tych samych wartości wielokrotnie
- możemy zapisywać już policzone wartości w tablicy

Obliczanie F_n — 2. podejście

```
int fib[MAXN+1];
fib[0]=fib[1]=1;
for(int n=2;n<=MAXN;n++)
    fib[n]=fib[n-1]+fib[n-2];
```

Złożoność: $O(n)$

Wniosek: rekurencja nie zawsze jest najlepszą metodą

Liczby Fibonacciego

Jak to poprawić?

- nie trzeba liczyć tych samych wartości wielokrotnie
- możemy zapisywać już policzone wartości w tablicy

Obliczanie F_n — 2. podejście

```
int fib[MAXN+1];
fib[0]=fib[1]=1;
for(int n=2;n<=MAXN;n++)
    fib[n]=fib[n-1]+fib[n-2];
```

Złożoność: $O(n)$
Wniosek: rekurencja nie zawsze jest najlepszą metodą

Największy wspólny dzielnik

Największy wspólny dzielnik

- *Największym Wspólnym Dzielnikiem (NWD)* dwóch liczb naturalnych a i b nazywamy największą liczbę naturalną dzielącą zarówno a jak i b.
- przykładowo $NWD(35, 20) = 5$
- jak policzyć $NWD(a, b)$?
- można sprawdzić wszystkie liczby $1, 2, \ldots, min(a, b)$ — złożoność $O(min(a, b))$

Największy wspólny dzielnik

Największy wspólny dzielnik

- *Największym Wspólnym Dzielnikiem (NWD)* dwóch liczb naturalnych a i b nazywamy największą liczbę naturalną dzielącą zarówno a jak i b.
- przykładowo $NWD(35, 20) = 5$
- jak policzyć $NWD(a, b)$?
- można sprawdzić wszystkie liczby $1, 2, \ldots, min(a, b)$ — złożoność $O(min(a, b))$

Największy wspólny dzielnik

Największy wspólny dzielnik

- *Największym Wspólnym Dzielnikiem (NWD)* dwóch liczb naturalnych a i b nazywamy największą liczbę naturalną dzielącą zarówno a jak i b.
- przykładowo $NWD(35, 20) = 5$
- jak policzyć $NWD(a, b)$?
- można sprawdzić wszystkie liczby $1, 2, \ldots, min(a, b)$ — złożoność $O(min(a, b))$

Największy wspólny dzielnik

Największy wspólny dzielnik

- *Największym Wspólnym Dzielnikiem (NWD)* dwóch liczb naturalnych a i b nazywamy największą liczbę naturalną dzielącą zarówno a jak i b.
- przykładowo $NWD(35, 20) = 5$
- jak policzyć $NWD(a, b)$?
- można sprawdzić wszystkie liczby $1, 2, \ldots, min(a, b)$ — złożoność $O(min(a, b))$

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Przykład działania

```
nwd(20,35) =nwd(35,20) =nwd(20,15) =nwd(15,5)
=nwd(5,0) =5
```

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Przykład działania

nwd(20,35) =nwd(35,20) =nwd(20,15) =nwd(15,5) =nwd(5,0) =5

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Przykład działania

```
nwd(20,35) =nwd(35,20) =nwd(20,15) =nwd(15,5)
=nwd(5,0) =5
```

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Przykład działania

```
nwd(20,35) =nwd(35,20) =nwd(20,15) =nwd(15,5)
=nwd(5,0) =5
```

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Przykład działania

```
nwd(20,35) =nwd(35,20) =nwd(20,15) =nwd(15,5)
=nwd(5,0) =5
```

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Przykład działania

```
nwd(20,35) =nwd(35,20) =nwd(20,15) =nwd(15,5)
=nwd(5,0) =5
```

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Przykład działania

```
nwd(20,35) =nwd(35,20) =nwd(20,15) =nwd(15,5)
=nwd(5,0) =5
```

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Dlaczego to działa?

- nie licząc pierwszego wywołania zachodzi $b < a$
- wówczas wartość b maleje przy każdym wywołaniu, więc algorytm się zatrzymuje
- zachodzi $NWD(a, b) = NWD(b, a \bmod b)$; dowód:
  1. niech $a = b \cdot n + m$, gdzie $m < b$
  2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
  3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
  4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Dlaczego to działa?

- nie licząc pierwszego wywołania zachodzi $b < a$
- wówczas wartość b maleje przy każdym wywołaniu, więc algorytm się zatrzymuje
- zachodzi $NWD(a, b) = NWD(b, a \bmod b)$; dowód:
  1. niech $a = b \cdot n + m$, gdzie $m < b$
  2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
  3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
  4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Dlaczego to działa?

- nie licząc pierwszego wywołania zachodzi $b < a$
- wówczas wartość b maleje przy każdym wywołaniu, więc algorytm się zatrzymuje
- zachodzi $NWD(a, b) = NWD(b, a \bmod b)$; dowód:
  1. niech $a = b \cdot n + m$, gdzie $m < b$
  2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
  3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
  4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Dlaczego to działa?

- nie licząc pierwszego wywołania zachodzi $b < a$
- wówczas wartość b maleje przy każdym wywołaniu, więc algorytm się zatrzymuje
- zachodzi $NWD(a, b) = NWD(b, a \bmod b)$; dowód:
  1. niech $a = b \cdot n + m$, gdzie $m < b$
  2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
  3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
  4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Dlaczego to działa?

- nie licząc pierwszego wywołania zachodzi $b < a$
- wówczas wartość b maleje przy każdym wywołaniu, więc algorytm się zatrzymuje
- zachodzi $NWD(a, b) = NWD(b, a \bmod b)$; dowód:
  1. niech $a = b \cdot n + m$, gdzie $m < b$
  2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
  3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
  4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Dlaczego to działa?

- nie licząc pierwszego wywołania zachodzi $b < a$
- wówczas wartość b maleje przy każdym wywołaniu, więc algorytm się zatrzymuje
- zachodzi $NWD(a, b) = NWD(b, a \bmod b)$; dowód:
  1. niech $a = b \cdot n + m$, gdzie $m < b$
  2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
  3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
  4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Dlaczego to działa?

- nie licząc pierwszego wywołania zachodzi $b < a$
- wówczas wartość b maleje przy każdym wywołaniu, więc algorytm się zatrzymuje
- zachodzi $NWD(a, b) = NWD(b, a \bmod b)$; dowód:
  1. niech $a = b \cdot n + m$, gdzie $m < b$
  2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
  3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
  4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

Największy wspólny dzielnik

Algorytm Euklidesa

```
int nwd(int a,int b){
    if(b == 0) return a;
    return nwd(b, a%b);
}
```

Dlaczego to działa?

- nie licząc pierwszego wywołania zachodzi $b < a$
- wówczas wartość b maleje przy każdym wywołaniu, więc algorytm się zatrzymuje
- zachodzi $NWD(a, b) = NWD(b, a \bmod b)$; dowód:
  1. niech $a = b \cdot n + m$, gdzie $m < b$
  2. chcemy wykazać, że $NWD(b \cdot n + m, b) = NWD(b, m)$
  3. wspólne dzielniki $b \cdot n + m$ i b dzielą też m
  4. wspólne dzielniki b i m dzielą też $b \cdot n + m$

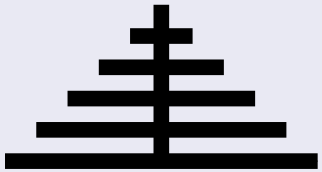
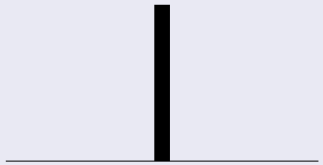
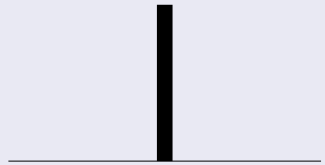
Największy Wspólny Dzielnik

Złożoność

Algorytm Euklidesa działa w czasie $O(\log min(a, b))$. Dowód pomijamy.

Wieże Hanoi

Wieże Hanoi

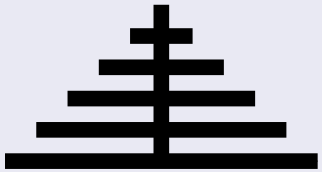
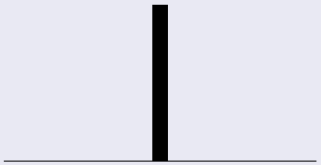
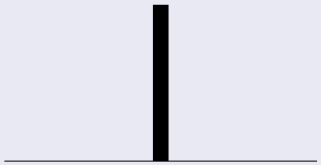




Problem

- mamy 3 kołki i n krążków różnej średnicy
- sytuację początkową przedstawia rysunek
- zadanie pole na przełożeniu wszystkich krążków na prawy kołek
- można przełożyć tylko jeden krążek na raz
- nie można kłaść krążka większego na mniejszy
- oznaczmy krążki `1, 2, ... n; 1` — najmniejszy

Wieże Hanoi

Wieże Hanoi

Problem

- mamy 3 kołki i n krążków różnej średnicy
- sytuację początkową przedstawia rysunek
- zadanie pole na przełożeniu wszystkich krążków na prawy kołek
- można przełożyć tylko jeden krążek na raz
- nie można kłaść krążka większego na mniejszy
- oznaczmy krążki `1, 2, ... n; 1` — najmniejszy

Wieże Hanoi

Wieże Hanoi

Problem

- mamy 3 kołki i *n* krążków różnej średnicy
- sytuację początkową przedstawia rysunek
- zadanie pole na przełożeniu wszystkich krążków na prawy kołek
- można przełożyć tylko jeden krążek na raz
- nie można kłaść krążka większego na mniejszy
- oznaczmy krążki `1, 2, ... n; 1` — najmniejszy

Wieże Hanoi

Wieże Hanoi

Problem

- mamy 3 kołki i n krążków różnej średnicy
- sytuację początkową przedstawia rysunek
- zadanie pole na przełożeniu wszystkich krążków na prawy kołek
- można przełożyć tylko jeden krążek na raz
- nie można kłaść krążka większego na mniejszy
- oznaczmy krążki `1, 2, ... n; 1` — najmniejszy

Wieże Hanoi

Wieże Hanoi

Problem

- mamy 3 kołki i n krążków różnej średnicy
- sytuację początkową przedstawia rysunek
- zadanie pole na przełożeniu wszystkich krążków na prawy kołek
- można przełożyć tylko jeden krążek na raz
- nie można kłaść krążka większego na mniejszy
- oznaczmy krążki `1, 2, ... n`; `1` — najmniejszy

Wieże Hanoi

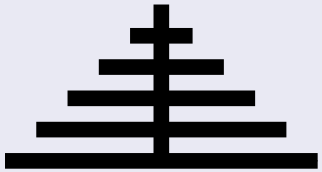
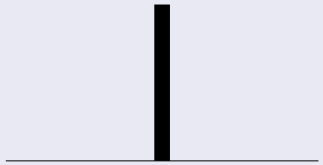
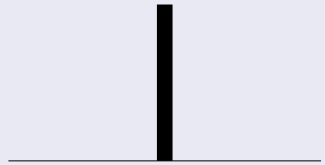
Wieże Hanoi

Problem

- mamy 3 kołki i n krążków różnej średnicy
- sytuację początkową przedstawia rysunek
- zadanie pole na przełożeniu wszystkich krążków na prawy kołek
- można przełożyć tylko jeden krążek na raz
- nie można kłaść krążka większego na mniejszy
- oznaczmy krążki `1`, `2`, ... `n`; `1` — najmniejszy

Wieże Hanoi

Wieże Hanoi

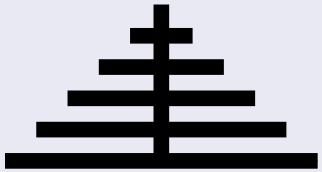
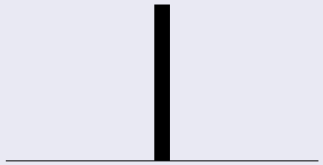
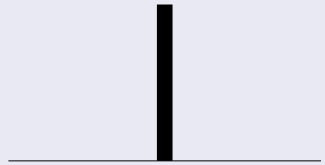




Rozwiązanie

- 1 możemy przemieszczać bez problemu
- aby przesunąć 12 na 3. kołek, możemy:
  1. przesunąć 1 na 2. kołek
  2. przesunąć 2 na 3. kołek
  3. przesunąć 1 na 3. kołek
- aby przesunąć 123 na 3. kołek, możemy:
  1. przesunąć 12 na 2. kołek (rekurencyjnie)
  2. przesunąć 3 na 3. kołek
  3. przesunąć 12 na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi

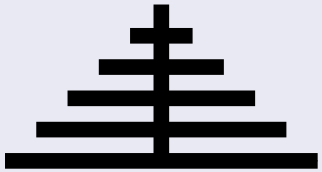
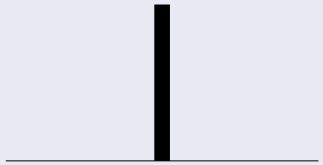
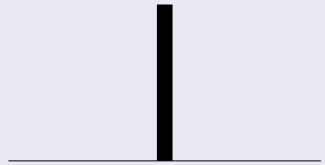




Rozwiązanie

- 1 możemy przemieszczać bez problemu
- aby przesunąć 12 na 3. kołek, możemy:
  1. przesunąć 1 na 2. kołek
  2. przesunąć 2 na 3. kołek
  3. przesunąć 1 na 3. kołek
- aby przesunąć 123 na 3. kołek, możemy:
  1. przesunąć 12 na 2. kołek (rekurencyjnie)
  2. przesunąć 3 na 3. kołek
  3. przesunąć 12 na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi

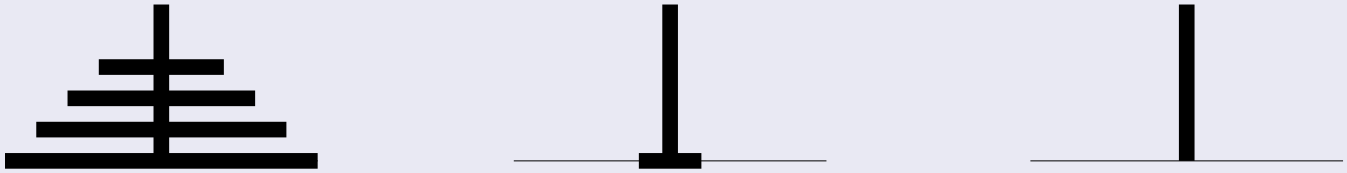




Rozwiązanie

- `1` możemy przemieszczać bez problemu
- aby przesunąć `12` na 3. kołek, możemy:
  1. przesunąć `1` na 2. kołek
  2. przesunąć `2` na 3. kołek
  3. przesunąć `1` na 3. kołek
- aby przesunąć `123` na 3. kołek, możemy:
  1. przesunąć `12` na 2. kołek (rekurencyjnie)
  2. przesunąć `3` na 3. kołek
  3. przesunąć `12` na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi

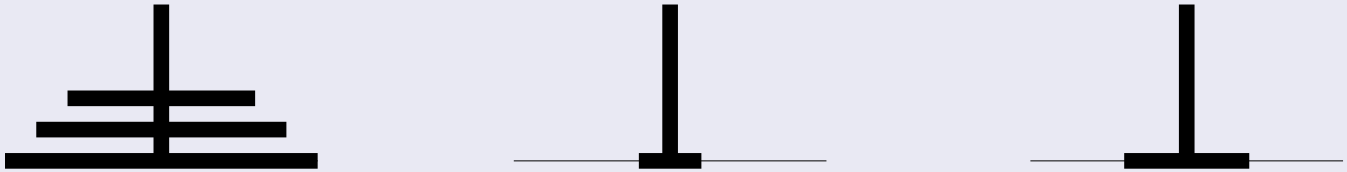


Rozwiązanie

- `1` możemy przemieszczać bez problemu
- aby przesunąć `12` na 3. kołek, możemy:
  1. przesunąć `1` na 2. kołek
  2. przesunąć `2` na 3. kołek
  3. przesunąć `1` na 3. kołek
- aby przesunąć `123` na 3. kołek, możemy:
  1. przesunąć `12` na 2. kołek (rekurencyjnie)
  2. przesunąć `3` na 3. kołek
  3. przesunąć `12` na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi



Rozwiązanie

- `1` możemy przemieszczać bez problemu
- aby przesunąć `12` na 3. kołek, możemy:
  1. przesunąć `1` na 2. kołek
  2. przesunąć `2` na 3. kołek
  3. przesunąć `1` na 3. kołek
- aby przesunąć `123` na 3. kołek, możemy:
  1. przesunąć `12` na 2. kołek (rekurencyjnie)
  2. przesunąć `3` na 3. kołek
  3. przesunąć `12` na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi

Rozwiązanie

- `1` możemy przemieszczać bez problemu
- aby przesunąć `12` na 3. kołek, możemy:
  1. przesunąć `1` na 2. kołek
  2. przesunąć `2` na 3. kołek
  3. przesunąć `1` na 3. kołek
- aby przesunąć `123` na 3. kołek, możemy:
  1. przesunąć `12` na 2. kołek (rekurencyjnie)
  2. przesunąć `3` na 3. kołek
  3. przesunąć `12` na 3. kołek (rekurencyjnie)

Wieże Hanoi

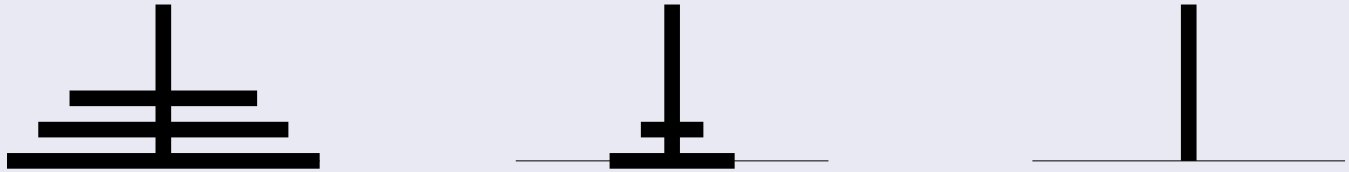
Wieże Hanoi

Rozwiązanie

- `1` możemy przemieszczać bez problemu
- aby przesunąć `12` na 3. kołek, możemy:
  1. przesunąć `1` na 2. kołek
  2. przesunąć `2` na 3. kołek
  3. przesunąć `1` na 3. kołek
- aby przesunąć `123` na 3. kołek, możemy:
  1. przesunąć `12` na 2. kołek (rekurencyjnie)
  2. przesunąć `3` na 3. kołek
  3. przesunąć `12` na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi

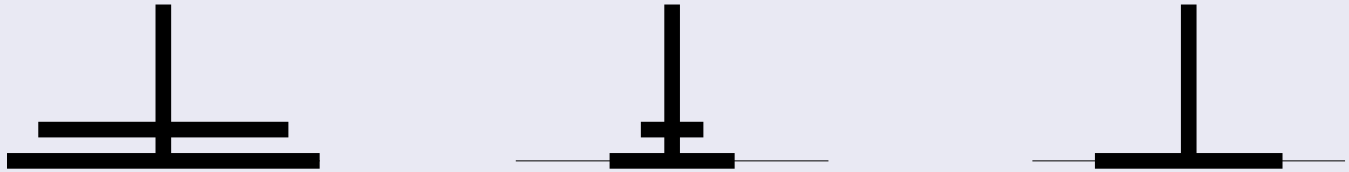


Rozwiązanie

- `1` możemy przemieszczać bez problemu
- aby przesunąć `12` na 3. kołek, możemy:
  1. przesunąć `1` na 2. kołek
  2. przesunąć `2` na 3. kołek
  3. przesunąć `1` na 3. kołek
- aby przesunąć `123` na 3. kołek, możemy:
  1. przesunąć `12` na 2. kołek (rekurencyjnie)
  2. przesunąć `3` na 3. kołek
  3. przesunąć `12` na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi



Rozwiązanie

- `1` możemy przemieszczać bez problemu
- aby przesunąć `12` na 3. kołek, możemy:
  1. przesunąć `1` na 2. kołek
  2. przesunąć `2` na 3. kołek
  3. przesunąć `1` na 3. kołek
- aby przesunąć `123` na 3. kołek, możemy:
  1. przesunąć `12` na 2. kołek (rekurencyjnie)
  2. przesunąć `3` na 3. kołek
  3. przesunąć `12` na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi

Rozwiązanie

- `1` możemy przemieszczać bez problemu
- aby przesunąć `12` na 3. kołek, możemy:
  1. przesunąć `1` na 2. kołek
  2. przesunąć `2` na 3. kołek
  3. przesunąć `1` na 3. kołek
- aby przesunąć `123` na 3. kołek, możemy:
  1. przesunąć `12` na 2. kołek (rekurencyjnie)
  2. przesunąć `3` na 3. kołek
  3. przesunąć `12` na 3. kołek (rekurencyjnie)

Wieże Hanoi

Wieże Hanoi

Rozwiązanie rekurencyjne:

- `hanoi(m,k)` — przeniesienie `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie)
- chcemy wykonać `hanoi(n,-1)`
- `hanoi(m,k)`:
  1. jeśli $m = 1$, to przesuń `1` i skończ
  2. `hanoi(m-1,-k)`
  3. przesuń `m` w kierunku k
  4. `hanoi(m-1,-k)`

Wieże Hanoi

Wieże Hanoi

Rozwiązanie rekurencyjne:

- `hanoi(m,k)` — przeniesienie `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie)
- chcemy wykonać `hanoi(n,-1)`
- `hanoi(m,k)`:
  1. jeśli $m = 1$, to przesuń `1` i skończ
  2. `hanoi(m-1,-k)`
  3. przesuń `m` w kierunku k
  4. `hanoi(m-1,-k)`

Wieże Hanoi

Wieże Hanoi

Rozwiązanie rekurencyjne:

- `hanoi(m,k)` — przeniesienie `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie)
- chcemy wykonać `hanoi(n,-1)`
- `hanoi(m,k)`:
  1. jeśli $m = 1$, to przesuń `1` i skończ
  2. `hanoi(m-1,-k)`
  3. przesuń `m` w kierunku k
  4. `hanoi(m-1,-k)`

Wieże Hanoi

Wieże Hanoi

Rozwiązanie rekurencyjne:

- `hanoi(m,k)` — przeniesienie `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie)
- chcemy wykonać `hanoi(n,-1)`
- `hanoi(m,k)`:
  1. jeśli $m = 1$, to przesuń `1` i skończ
  2. `hanoi(m-1,-k)`
  3. przesuń `m` w kierunku k
  4. `hanoi(m-1,-k)`

Wieże Hanoi

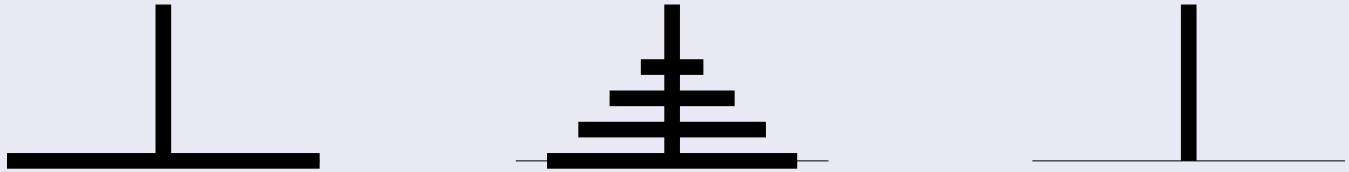
Wieże Hanoi

Rozwiązanie rekurencyjne:

- `hanoi(m,k)` — przeniesienie `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie)
- chcemy wykonać `hanoi(n,-1)`
- `hanoi(m,k)`:
  1. jeśli $m = 1$, to przesuń `1` i skończ
  2. `hanoi(m-1,-k)`
  3. przesuń `m` w kierunku k
  4. `hanoi(m-1,-k)`

Wieże Hanoi

Wieże Hanoi

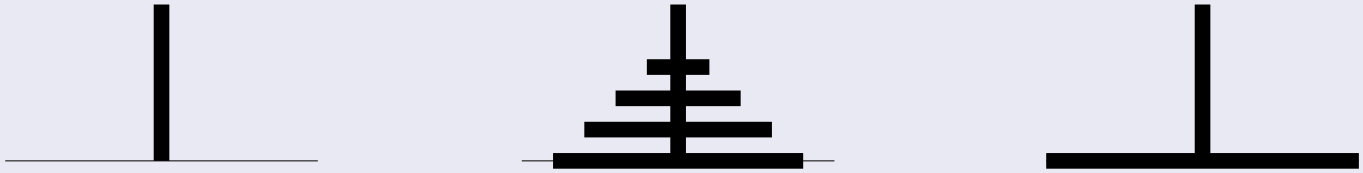


Rozwiązanie rekurencyjne:

- `hanoi(m,k)` — przeniesienie `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie)
- chcemy wykonać `hanoi(n,-1)`
- `hanoi(m,k)`:
  1. jeśli $m = 1$, to przesuń `1` i skończ
  2. `hanoi(m-1,-k)`
  3. przesuń `m` w kierunku k
  4. `hanoi(m-1,-k)`

Wieże Hanoi

Wieże Hanoi



Rozwiązanie rekurencyjne:

- `hanoi(m,k)` — przeniesienie `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie)
- chcemy wykonać `hanoi(n,-1)`
- `hanoi(m,k)`:
  1. jeśli $m = 1$, to przesuń `1` i skończ
  2. `hanoi(m-1,-k)`
  3. przesuń `m` w kierunku k
  4. `hanoi(m-1,-k)`

Wieże Hanoi

Wieże Hanoi

Rozwiązanie rekurencyjne:

- `hanoi(m,k)` — przeniesienie `12...m` w kierunku k, gdzie $k = 1$ oznacza przesunięcie na prawo a $k = -1$ na lewo od obecnej pozycji (cyklicznie)
- chcemy wykonać `hanoi(n,-1)`
- `hanoi(m,k)`:
  1. jeśli $m = 1$, to przesuń `1` i skończ
  2. `hanoi(m-1,-k)`
  3. przesuń `m` w kierunku k
  4. `hanoi(m-1,-k)`

Wieże Hanoi

Ile ruchów wykonujemy?

- T_n — liczba ruchów które wykonujemy przy przenoszeniu `12...n`
- $T_1 = 1$
- $T_n = 2 \cdot T_n - 1 + 1$ dla $n > 1$
-

n	1	2	3	4	5
T_n	1	3	7	15	31

- teza: $T_n = 2^n - 1$
- dowód przez indukcję:
  - baza: $T_1 = 1 = 2^1 - 1$
  - krok indukcyjny:
    $T_n = 2 \cdot T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Wieże Hanoi

Ile ruchów wykonujemy?

- T_n — liczba ruchów które wykonujemy przy przenoszeniu `12...n`
- $T_1 = 1$
- $T_n = 2 \cdot T_n - 1 + 1$ dla $n > 1$
n	1	2	3	4	5
T_n	1	3	7	15	31
- teza: $T_n = 2^n - 1$
- dowód przez indukcję:
  - baza: $T_1 = 1 = 2^1 - 1$
  - krok indukcyjny:
    $T_n = 2 \cdot T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Wieże Hanoi

Ile ruchów wykonujemy?

- T_n — liczba ruchów które wykonujemy przy przenoszeniu `12...n`
- $T_1 = 1$
- $T_n = 2 \cdot T_n - 1 + 1$ dla $n > 1$
n	1	2	3	4	5
T_n	1	3	7	15	31
- teza: $T_n = 2^n - 1$
- dowód przez indukcję:
  - baza: $T_1 = 1 = 2^1 - 1$
  - krok indukcyjny:
    $T_n = 2 \cdot T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Wieże Hanoi

Ile ruchów wykonujemy?

- T_n — liczba ruchów które wykonujemy przy przenoszeniu `12...n`
- $T_1 = 1$
- $T_n = 2 \cdot T_n - 1 + 1$ dla $n > 1$
n	1	2	3	4	5
T_n	1	3	7	15	31
- teza: $T_n = 2^n - 1$
- dowód przez indukcję:
  - baza: $T_1 = 1 = 2^1 - 1$
  - krok indukcyjny:
    $T_n = 2 \cdot T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Wieże Hanoi

Ile ruchów wykonujemy?

- T_n — liczba ruchów które wykonujemy przy przenoszeniu `12...n`
- $T_1 = 1$
- $T_n = 2 \cdot T_n - 1 + 1$ dla $n > 1$
n	1	2	3	4	5
T_n	1	3	7	15	31
- teza: $T_n = 2^n - 1$
- dowód przez indukcję:
  - baza: $T_1 = 1 = 2^1 - 1$
  - krok indukcyjny:
    $T_n = 2 \cdot T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Wieże Hanoi

Ile ruchów wykonujemy?

- T_n — liczba ruchów które wykonujemy przy przenoszeniu `12...n`
- $T_1 = 1$
- $T_n = 2 \cdot T_n - 1 + 1$ dla $n > 1$
-

n	1	2	3	4	5
T_n	1	3	7	15	31

- teza: $T_n = 2^n - 1$
- dowód przez indukcję:
  - baza: $T_1 = 1 = 2^1 - 1$
  - krok indukcyjny:
    $T_n = 2 \cdot T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Wieże Hanoi

Ile ruchów wykonujemy?

- T_n — liczba ruchów które wykonujemy przy przenoszeniu `12...n`
- $T_1 = 1$
- $T_n = 2 \cdot T_n - 1 + 1$ dla $n > 1$
-

n	1	2	3	4	5
T_n	1	3	7	15	31

- teza: $T_n = 2^n - 1$
- dowód przez indukcję:
  - baza: $T_1 = 1 = 2^1 - 1$
  - krok indukcyjny:
    $T_n = 2 \cdot T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Wieże Hanoi

Ile ruchów wykonujemy?

- T_n — liczba ruchów które wykonujemy przy przenoszeniu `12...n`
- $T_1 = 1$
- $T_n = 2 \cdot T_n - 1 + 1$ dla $n > 1$
-

n	1	2	3	4	5
T_n	1	3	7	15	31

- teza: $T_n = 2^n - 1$
- dowód przez indukcję:
  - baza: $T_1 = 1 = 2^1 - 1$
  - krok indukcyjny:
    $T_n = 2 \cdot T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$

Rekurencja

Problem

Z szachownicy 2^nx2^n wycięto jedno pole. Jak za pomocą klocków w kształcie litery L (zajmujących 3 pola) pokryć taką szachownicę?

Rozwiązanie

1. jeśli $n = 1$, to zadanie jest łatwe
2. jeśli $n > 1$, to podzielmy naszą szachownicę na cztery szachownice 2^{n-1}x2^{n-1} (pozioma i pionowa linia przechodząca przez środek planszy)
3. w środkowym kwadracie 2x2 układamy L w taki sposób, aby w każdej mniejszej szachownicy dokładnie jedno pole było zajęte (licząc to wycięte na wstępie)
4. cztery małe szachownice pokrywamy rekurencyjnie

Rekurencja

Problem

Z szachownicy 2^nx2^n wycięto jedno pole. Jak za pomocą klocków w kształcie litery L (zajmujących 3 pola) pokryć taką szachownicę?

Rozwiązanie

1. jeśli $n = 1$, to zadanie jest łatwe
2. jeśli $n > 1$, to podzielmy naszą szachownicę na cztery szachownice 2^{n-1}x2^{n-1} (pozioma i pionowa linia przechodząca przez środek planszy)
3. w środkowym kwadracie 2x2 układamy L w taki sposób, aby w każdej mniejszej szachownicy dokładnie jedno pole było zajęte (licząc to wycięte na wstępie)
4. cztery małe szachownice pokrywamy rekurencyjnie

Rekurencja

Problem

Z szachownicy 2^nx2^n wycięto jedno pole. Jak za pomocą klocków w kształcie litery L (zajmujących 3 pola) pokryć taką szachownicę?

Rozwiązanie

1. jeśli $n = 1$, to zadanie jest łatwe
2. jeśli $n > 1$, to podzielmy naszą szachownicę na cztery szachownice 2^{n-1}x2^{n-1} (pozioma i pionowa linia przechodząca przez środek planszy)
3. w środkowym kwadracie 2x2 układamy L w taki sposób, aby w każdej mniejszej szachownicy dokładnie jedno pole było zajęte (licząc to wycięte na wstępie)
4. cztery małe szachownice pokrywamy rekurencyjnie

Rekurencja

Problem

Z szachownicy 2^nx2^n wycięto jedno pole. Jak za pomocą klocków w kształcie litery L (zajmujących 3 pola) pokryć taką szachownicę?

Rozwiązanie

1. jeśli $n = 1$, to zadanie jest łatwe
2. jeśli $n > 1$, to podzielmy naszą szachownicę na cztery szachownice 2^{n-1}x2^{n-1} (pozioma i pionowa linia przechodząca przez środek planszy)
3. w środkowym kwadracie 2x2 układamy L w taki sposób, aby w każdej mniejszej szachownicy dokładnie jedno pole było zajęte (licząc to wycięte na wstępie)
4. cztery małe szachownice pokrywamy rekurencyjnie

Rekurencja

Problem

Z szachownicy 2^nx2^n wycięto jedno pole. Jak za pomocą klocków w kształcie litery L (zajmujących 3 pola) pokryć taką szachownicę?

Rozwiązanie

1. jeśli $n = 1$, to zadanie jest łatwe
2. jeśli $n > 1$, to podzielmy naszą szachownicę na cztery szachownice 2^{n-1}x2^{n-1} (pozioma i pionowa linia przechodząca przez środek planszy)
3. w środkowym kwadracie 2x2 układamy L w taki sposób, aby w każdej mniejszej szachownicy dokładnie jedno pole było zajęte (licząc to wycięte na wstępie)
4. cztery małe szachownice pokrywamy rekurencyjnie

Przeszukiwanie przestrzeni rozwiązań

Problem 8 hetmanów

Czy da się ustawić na szachownicy 8 hetmanów, tak aby nie atakowały się wzajemnie? Hetman atakuje wszystkie pola leżące w tym samym wierszu, kolumnie lub na tej samej przekątnej.

Zbiór rozwiązań

Jest to typowy problem w którym mamy pewien zbiór możliwych rozwiązań — zbiór ustawień 8 hetmanów na szachownicy i szukamy jednego lub wszystkich rozwiązań, które spełniają jakiś warunek — w tym przypadku hetmany się nie atakują.

Przeszukiwanie przestrzeni rozwiązań

Problem 8 hetmanów

Czy da się ustawić na szachownicy 8 hetmanów, tak aby nie atakowały się wzajemnie? Hetman atakuje wszystkie pola leżące w tym samym wierszu, kolumnie lub na tej samej przekątnej.

Zbiór rozwiązań

Jest to typowy problem w którym mamy pewien zbiór możliwych rozwiązań — zbiór ustawień 8 hetmanów na szachownicy i szukamy jednego lub wszystkich rozwiązań, które spełniają jakiś warunek — w tym przypadku hetmany się nie atakują.

Problem 8 hetmanów

Rozwiązanie

- najprostszym rozwiązaniem jest sprawdzenie wszystkich możliwych rozstawień hetmanów
- zauważmy jednak, że jeśli pierwsze dwa postawione hetmany się atakują, to nie ma sensu próbować rozstawić pozostałych 6, gdyż taki układ nigdy nie spełni naszych warunków
- możemy konstruować rozwiązanie, zaczynając od pustej planszy i rozbudowując je o kolejne hetmany
- kiedy jednak dwa już postawione hetmany będą się biły to możemy pominąć taką konfigurację

Problem 8 hetmanów

Rozwiązanie

- najprostszym rozwiązaniem jest sprawdzenie wszystkich możliwych rozstawień hetmanów
- zauważmy jednak, że jeśli pierwsze dwa postawione hetmany się atakują, to nie ma sensu próbować rozstawić pozostałych 6, gdyż taki układ nigdy nie spełni naszych warunków
- możemy konstruować rozwiązanie, zaczynając od pustej planszy i rozbudowując je o kolejne hetmany
- kiedy jednak dwa już postawione hetmany będą się biły to możemy pominąć taką konfigurację

Problem 8 hetmanów

Rozwiązanie

- najprostszym rozwiązaniem jest sprawdzenie wszystkich możliwych rozstawień hetmanów
- zauważmy jednak, że jeśli pierwsze dwa postawione hetmany się atakują, to nie ma sensu próbować rozstawić pozostałych 6, gdyż taki układ nigdy nie spełni naszych warunków
- możemy konstruować rozwiązanie, zaczynając od pustej planszy i rozbudowując je o kolejne hetmany
- kiedy jednak dwa już postawione hetmany będą się biły to możemy pominąć taką konfigurację

Problem 8 hetmanów

Rozwiązanie

- najprostszym rozwiązaniem jest sprawdzenie wszystkich możliwych rozstawień hetmanów
- zauważmy jednak, że jeśli pierwsze dwa postawione hetmany się atakują, to nie ma sensu próbować rozstawić pozostałych 6, gdyż taki układ nigdy nie spełni naszych warunków
- możemy konstruować rozwiązanie, zaczynając od pustej planszy i rozbudowując je o kolejne hetmany
- kiedy jednak dwa już postawione hetmany będą się biły to możemy pominąć taką konfigurację

Backtracking — przeszukiwanie z nawrotami:

- to ogólna technika pozwalająca przeglądać możliwe rozwiązania i szukać tych, które spełniają pewne warunki
- polega ona na stopniowym rozbudowywaniu rozwiązania (dodawanie hetmanów)
- jeśli jednak aktualne rozwiązanie nie może być rozbudowane (2 hetmany się szachują), to następuje powrót do poprzedniego kroku, gdzie podejmowana jest próba znalezienia innej możliwości
- proces ten można zapisać rekurencyjnie — aby przetworzyć rozwiązanie x:
  1. jeśli x jest dobrym rozwiązaniem to je wypisz
  2. jeśli x nie może być rozbudowane to skończ
  3. spróbuj wszystkich możliwości rozbudowania x i przetwórz każdą z nich

Backtracking — przeszukiwanie z nawrotami:

- to ogólna technika pozwalająca przeglądać możliwe rozwiązania i szukać tych, które spełniają pewne warunki
- polega ona na stopniowym rozbudowywaniu rozwiązania (dodawanie hetmanów)
- jeśli jednak aktualne rozwiązanie nie może być rozbudowane (2 hetmany się szachują), to następuje powrót do poprzedniego kroku, gdzie podejmowana jest próba znalezienia innej możliwości
- proces ten można zapisać rekurencyjnie — aby przetworzyć rozwiązanie x:
  1. jeśli x jest dobrym rozwiązaniem to je wypisz
  2. jeśli x nie może być rozbudowane to skończ
  3. spróbuj wszystkich możliwości rozbudowania x i przetwórz każdą z nich

Backtracking — przeszukiwanie z nawrotami:

- to ogólna technika pozwalająca przeglądać możliwe rozwiązania i szukać tych, które spełniają pewne warunki
- polega ona na stopniowym rozbudowywaniu rozwiązania (dodawanie hetmanów)
- jeśli jednak aktualne rozwiązanie nie może być rozbudowane (2 hetmany się szachują), to następuje powrót do poprzedniego kroku, gdzie podejmowana jest próba znalezienia innej możliwości
- proces ten można zapisać rekurencyjnie — aby przetworzyć rozwiązanie x:
  1. jeśli x jest dobrym rozwiązaniem to je wypisz
  2. jeśli x nie może być rozbudowane to skończ
  3. spróbuj wszystkich możliwości rozbudowania x i przetwórz każdą z nich

Backtracking — przeszukiwanie z nawrotami:

- to ogólna technika pozwalająca przeglądać możliwe rozwiązania i szukać tych, które spełniają pewne warunki
- polega ona na stopniowym rozbudowywaniu rozwiązania (dodawanie hetmanów)
- jeśli jednak aktualne rozwiązanie nie może być rozbudowane (2 hetmany się szachują), to następuje powrót do poprzedniego kroku, gdzie podejmowana jest próba znalezienia innej możliwości
- proces ten można zapisać rekurencyjnie — aby przetworzyć rozwiązanie x:
  1. jeśli x jest dobrym rozwiązaniem to je wypisz
  2. jeśli x nie może być rozbudowane to skończ
  3. spróbuj wszystkich możliwości rozbudowania x i przetwórz każdą z nich

Backtracking — przeszukiwanie z nawrotami:

- to ogólna technika pozwalająca przeglądać możliwe rozwiązania i szukać tych, które spełniają pewne warunki
- polega ona na stopniowym rozbudowywaniu rozwiązania (dodawanie hetmanów)
- jeśli jednak aktualne rozwiązanie nie może być rozbudowane (2 hetmany się szachują), to następuje powrót do poprzedniego kroku, gdzie podejmowana jest próba znalezienia innej możliwości
- proces ten można zapisać rekurencyjnie — aby przetworzyć rozwiązanie x:
  1. jeśli x jest dobrym rozwiązaniem to je wypisz
  2. jeśli x nie może być rozbudowane to skończ
  3. spróbuj wszystkich możliwości rozbudowania x i przetwórz każdą z nich

Backtracking — przeszukiwanie z nawrotami:

- to ogólna technika pozwalająca przeglądać możliwe rozwiązania i szukać tych, które spełniają pewne warunki
- polega ona na stopniowym rozbudowywaniu rozwiązania (dodawanie hetmanów)
- jeśli jednak aktualne rozwiązanie nie może być rozbudowane (2 hetmany się szachują), to następuje powrót do poprzedniego kroku, gdzie podejmowana jest próba znalezienia innej możliwości
- proces ten można zapisać rekurencyjnie — aby przetworzyć rozwiązanie x:
  1. jeśli x jest dobrym rozwiązaniem to je wypisz
  2. jeśli x nie może być rozbudowane to skończ
  3. spróbuj wszystkich możliwości rozbudowania x i przetwórz każdą z nich

Backtracking — przeszukiwanie z nawrotami:

- to ogólna technika pozwalająca przeglądać możliwe rozwiązania i szukać tych, które spełniają pewne warunki
- polega ona na stopniowym rozbudowywaniu rozwiązania (dodawanie hetmanów)
- jeśli jednak aktualne rozwiązanie nie może być rozbudowane (2 hetmany się szachują), to następuje powrót do poprzedniego kroku, gdzie podejmowana jest próba znalezienia innej możliwości
- proces ten można zapisać rekurencyjnie — aby przetworzyć rozwiązanie x:
  1. jeśli x jest dobrym rozwiązaniem to je wypisz
  2. jeśli x nie może być rozbudowane to skończ
  3. spróbuj wszystkich możliwości rozbudowania x i przetwórz każdą z nich

Problem 8 hetmanów

Szczegóły implementacyjne

- w każdym wierszu może stać co najwyżej jeden hetman
- aktualne rozwiązanie możemy trzymać jako tablicę 8 liczb; $het[i]$ oznacza położenie hetmana w i-tym wierszu
- będziemy dostawiać kolejne hetmany w kolejnych wierszach, dzięki czemu każde rozstawienie hetmanów będziemy mogli otrzymać tylko na jeden sposób

Problem 8 hetmanów

Szczegóły implementacyjne

- w każdym wierszu może stać co najwyżej jeden hetman
- aktualne rozwiązanie możemy trzymać jako tablicę 8 liczb; $het[i]$ oznacza położenie hetmana w i-tym wierszu
- będziemy dostawiać kolejne hetmany w kolejnych wierszach, dzięki czemu każde rozstawienie hetmanów będziemy mogli otrzymać tylko na jeden sposób

Problem 8 hetmanów

Szczegóły implementacyjne

- w każdym wierszu może stać co najwyżej jeden hetman
- aktualne rozwiązanie możemy trzymać jako tablicę 8 liczb; $het[i]$ oznacza położenie hetmana w i-tym wierszu
- będziemy dostawiać kolejne hetmany w kolejnych wierszach, dzięki czemu każde rozstawienie hetmanów będziemy mogli otrzymać tylko na jeden sposób

Implementacja

Implementacja

Program rozstawiający 8 hetmanów na szachownicy znajduje się w notatkach.

Backtracking

Dlaczego warto stosować backtracking?

- dzięki pomijaniu całych grup rozwiązań wystarczy przejrzeć o wiele mniej konfiguracji niż w rozwiązaniu brutalnym
- w przypadku problemu 8 hetmanów używając przeszukiwania z nawrotami można przy odrobinie cierpliwości znaleźć poprawne ustawienie bez użycia komputera
- pomimo trudności oszacowania złożoności rozwiązań opartych na poszukiwaniu z nawrotami, zazwyczaj działają one dużo szybciej niż rozwiązania brutalne

Backtracking

Dlaczego warto stosować backtracking?

- dzięki pomijaniu całych grup rozwiązań wystarczy przejrzeć o wiele mniej konfiguracji niż w rozwiązaniu brutalnym
- w przypadku problemu 8 hetmanów używając przeszukiwania z nawrotami można przy odrobinie cierpliwości znaleźć poprawne ustawienie bez użycia komputera
- pomimo trudności oszacowania złożoności rozwiązań opartych na poszukiwaniu z nawrotami, zazwyczaj działają one dużo szybciej niż rozwiązania brutalne

Backtracking

Dlaczego warto stosować backtracking?

- dzięki pomijaniu całych grup rozwiązań wystarczy przejrzeć o wiele mniej konfiguracji niż w rozwiązaniu brutalnym
- w przypadku problemu 8 hetmanów używając przeszukiwania z nawrotami można przy odrobinie cierpliwości znaleźć poprawne ustawienie bez użycia komputera
- pomimo trudności oszacowania złożoności rozwiązań opartych na poszukiwaniu z nawrotami, zazwyczaj działają one dużo szybciej niż rozwiązania brutalne

Porównanie 3 technik

Porównanie 3 technik

- omówiliśmy już 3 ważne techniki programowania:
  - backtracking
  - programowanie dynamiczne
  - algorytmy zachłanne
- backtracking jest zazwyczaj najwolniejszy, ale ma najszersze zastosowania
- programowanie dynamiczne jest zazwyczaj szybsze, ale skonstruowanie rozwiązania opartego o tą technikę może być trudniejsze lub czasami niemożliwe
- algorytmy zachłanne są zazwyczaj najszybsze, ale mają najbardziej ograniczone zastosowanie

Porównanie 3 technik

Porównanie 3 technik

- omówiliśmy już 3 ważne techniki programowania:
  - backtracking
  - programowanie dynamiczne
  - algorytmy zachłanne
- backtracking jest zazwyczaj najwolniejszy, ale ma najszersze zastosowania
- programowanie dynamiczne jest zazwyczaj szybsze, ale skonstruowanie rozwiązania opartego o tą technikę może być trudniejsze lub czasami niemożliwe
- algorytmy zachłanne są zazwyczaj najszybsze, ale mają najbardziej ograniczone zastosowanie

Porównanie 3 technik

Porównanie 3 technik

- omówiliśmy już 3 ważne techniki programowania:
  - backtracking
  - programowanie dynamiczne
  - algorytmy zachłanne
- backtracking jest zazwyczaj najwolniejszy, ale ma najszersze zastosowania
- programowanie dynamiczne jest zazwyczaj szybsze, ale skonstruowanie rozwiązania opartego o tą technikę może być trudniejsze lub czasami niemożliwe
- algorytmy zachłanne są zazwyczaj najszybsze, ale mają najbardziej ograniczone zastosowanie

Porównanie 3 technik

Porównanie 3 technik

- omówiliśmy już 3 ważne techniki programowania:
  - backtracking
  - programowanie dynamiczne
  - algorytmy zachłanne
- backtracking jest zazwyczaj najwolniejszy, ale ma najszersze zastosowania
- programowanie dynamiczne jest zazwyczaj szybsze, ale skonstruowanie rozwiązania opartego o tą technikę może być trudniejsze lub czasami niemożliwe
- algorytmy zachłanne są zazwyczaj najszybsze, ale mają najbardziej ograniczone zastosowanie

Porównanie 3 technik

Porównanie 3 technik

- omówiliśmy już 3 ważne techniki programowania:
  - backtracking
  - programowanie dynamiczne
  - algorytmy zachłanne
- backtracking jest zazwyczaj najwolniejszy, ale ma najszersze zastosowania
- programowanie dynamiczne jest zazwyczaj szybsze, ale skonstruowanie rozwiązania opartego o tą technikę może być trudniejsze lub czasami niemożliwe
- algorytmy zachłanne są zazwyczaj najszybsze, ale mają najbardziej ograniczone zastosowanie

Porównanie 3 technik

Porównanie 3 technik

- omówiliśmy już 3 ważne techniki programowania:
  - backtracking
  - programowanie dynamiczne
  - algorytmy zachłanne
- backtracking jest zazwyczaj najwolniejszy, ale ma najszersze zastosowania
- programowanie dynamiczne jest zazwyczaj szybsze, ale skonstruowanie rozwiązania opartego o tą technikę może być trudniejsze lub czasami niemożliwe
- algorytmy zachłanne są zazwyczaj najszybsze, ale mają najbardziej ograniczone zastosowanie

Porównanie 3 technik

Porównanie 3 technik

- omówiliśmy już 3 ważne techniki programowania:
  - backtracking
  - programowanie dynamiczne
  - algorytmy zachłanne
- backtracking jest zazwyczaj najwolniejszy, ale ma najszersze zastosowania
- programowanie dynamiczne jest zazwyczaj szybsze, ale skonstruowanie rozwiązania opartego o tą technikę może być trudniejsze lub czasami niemożliwe
- algorytmy zachłanne są zazwyczaj najszybsze, ale mają najbardziej ograniczone zastosowanie

Sudoku

Problem

W jaki sposób można szukać rozwiązań Sudoku 9x9? Zwróć uwagę na brak możliwości zastosowania algorytmu brutalnego z powodu olbrzymiej ilości potencjalnych rozwiązań.

Rozwiązanie

- konfiguracje możemy trzymać jako dwuwymiarową tablicę 9x9 zawierającą liczby w kolejnych komórkach
- w kolejnych krokach będziemy dopisywać liczby w kolejnych wolnych polach wierszami od góry (a w obrębie wiersza od lewej)
- przed wpisaniem liczby w puste pole, będziemy wyznaczali liczby, które możemy tam wpisać przeglądając odpowiedni wiersz, kolumnę i kwadrat 3x3

Sudoku

Problem

W jaki sposób można szukać rozwiązań Sudoku 9x9? Zwróć uwagę na brak możliwości zastosowania algorytmu brutalnego z powodu olbrzymiej ilości potencjalnych rozwiązań.

Rozwiązanie

- konfiguracje możemy trzymać jako dwuwymiarową tablicę 9x9 zawierającą liczby w kolejnych komórkach
- w kolejnych krokach będziemy dopisywać liczby w kolejnych wolnych polach wierszami od góry (a w obrębie wiersza od lewej)
- przed wpisaniem liczby w puste pole, będziemy wyznaczali liczby, które możemy tam wpisać przeglądając odpowiedni wiersz, kolumnę i kwadrat 3x3

Sudoku

Problem

W jaki sposób można szukać rozwiązań Sudoku 9x9? Zwróć uwagę na brak możliwości zastosowania algorytmu brutalnego z powodu olbrzymiej ilości potencjalnych rozwiązań.

Rozwiązanie

- konfiguracje możemy trzymać jako dwuwymiarową tablicę 9x9 zawierającą liczby w kolejnych komórkach
- w kolejnych krokach będziemy dopisywać liczby w kolejnych wolnych polach wierszami od góry (a w obrębie wiersza od lewej)
- przed wpisaniem liczby w puste pole, będziemy wyznaczali liczby, które możemy tam wpisać przeglądając odpowiedni wiersz, kolumnę i kwadrat 3x3

Sudoku

Problem

W jaki sposób można szukać rozwiązań Sudoku 9x9? Zwróć uwagę na brak możliwości zastosowania algorytmu brutalnego z powodu olbrzymiej ilości potencjalnych rozwiązań.

Rozwiązanie

- konfiguracje możemy trzymać jako dwuwymiarową tablicę 9x9 zawierającą liczby w kolejnych komórkach
- w kolejnych krokach będziemy dopisywać liczby w kolejnych wolnych polach wierszami od góry (a w obrębie wiersza od lewej)
- przed wpisaniem liczby w puste pole, będziemy wyznaczali liczby, które możemy tam wpisać przeglądając odpowiedni wiersz, kolumnę i kwadrat 3x3