Wstęp do informatyki

Wykład 9

Przeszukiwanie z nawrotami

Instytut Informatyki UWr

Temat wykładu

- Przeszukiwanie z nawrotami, na przykładach:
  - problem (ośmiu) hetmanów
  - przeszukiwanie labiryntu, obszaru, …

Problem (ośmiu) hetmanów

Problem:

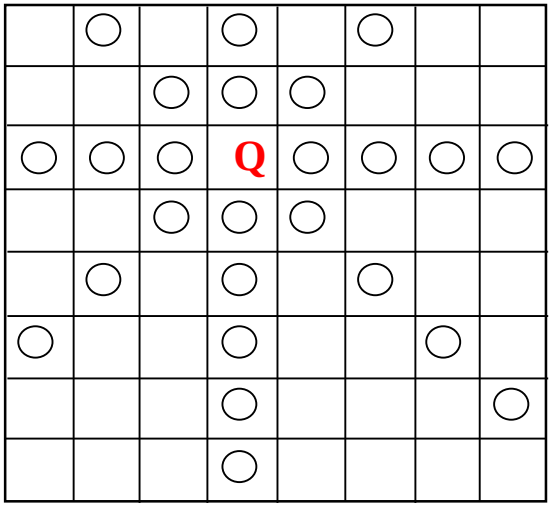
Osiem hetmanów należy rozmieścić na szachownicy w taki sposób, że żadne dwa nie atakują się nawzajem.

Uogólnienie:

n hetmanów należy rozmieścić na szachownicy rozmiaru $n \times n$ w taki sposób, że żadne dwa nie atakują się nawzajem.

Atakowanie pól

Hetman atakuje każde pole leżące w tym samym wierszu, kolumnie lub na tej samej „przekątnej”.



Przykład.

Q oznacza pozycję hetmana, kółka oznaczają pola przez niego atakowane.

Czy istnieje rozwiązanie?

n=2:

Brak rozwiązania

Q	o
o	o

n=3:

Brak rozwiązania

Q	o	o
o	o	
o		o

„wolne” pola w drugim i trzecim wierszu atakują się nawzajem

o	o	
Q	o	o
o	o	

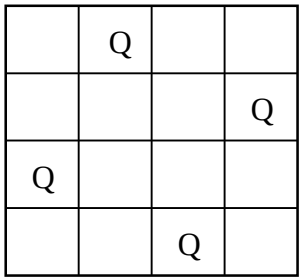
brak wolnego pola w drugiej kolumnie

Czy istnieje rozwiązanie (cd) ?

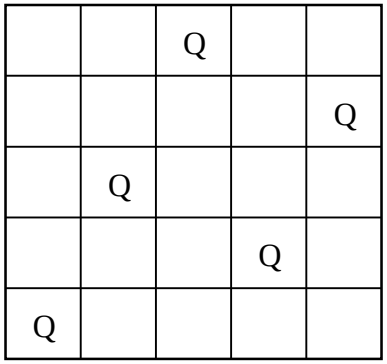
$n \geq 4$: istnieje (wiele) rozwiązań

Przykład:

n=4



n=5



Problem

Wejście

Liczba naturalna n

Wyjście

Rozwiązanie (dowolne) dla problemu hetmanów na szachownicy $n \times n$ lub informacja, że rozwiązanie nie istnieje.

Uwaga: przez rozwiązanie problemu hetmanów należy rozumieć rozmieszczenie n hetmanów zgodne ze specyfikacją problemu hetmanów.

Problem hetmanów

UWAGI

- Istnieją efektywne (szybkie) algorytmy rozwiązujące ten problem. Bazują one na obserwacji, że jeśli istnieją rozwiązania, to są wśród nich takie, w których hetmany rozmieszczone są „regularnie”.
- Na dzisiejszym wykładzie NIE będziemy szukać takich „regularności”.
- Nasz cel: znaleźć rozwiązanie (o ile istnieje) metodą „nawrotów” (czyli zorganizowanego „przeszukiwania” możliwości).

Rozwiązanie 1

Rozwiązanie naiwne

Struktura danych:

int b[n][n]

b[i][j] = 1 gdy hetman umieszczony na polu (i,j)

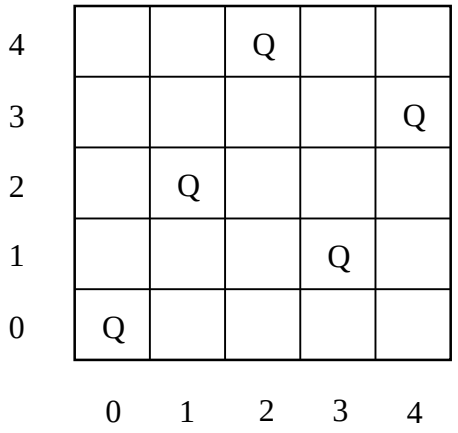
b[i][j] = 0 w przeciwnym przypadku

Rozwiązanie 1

Przykład

$n = 5$

szachownica



Struktura danych:

int b[n][n]

tablica `b[ ][ ]`

4	0	0	**1**	0	0
3	0	0	0	0	**1**
2	0	**1**	0	0	0
1	0	0	0	**1**	0
0	**1**	0	0	0	0
	0	1	2	3	4

Rozwiązanie 1

Rozwiązanie naiwne

Struktura danych:

int b[n][n]

Algorytm:

1. Dla każdego możliwego rozmieszczenia *n* hetmanów na szachownicy $n \times n$:
   - Sprawdź czy jakaś para hetmanów atakuje się nawzajem (jeśli nie – zwróć rozwiązanie i **zakończ**)
2. Zwróć informację o braku rozwiązania

Ile rozmieszczeń? Bardzo dużo… $\binom{n^2}{n}$

Rozwiązanie 2

(trochę mniej) naiwne

Struktura danych:

int b[n][n]

Algorytm:

1. Dla każdego możliwego rozmieszczenia *n* hetmanów na szachownicy *n*×*n* takiego, że w każdej kolumnie jest jeden hetman:
   - Sprawdź czy jakaś para hetmanów atakuje się nawzajem (jeśli nie – zwróć rozwiązanie i **zakończ**)
2. Zwróć informację o braku rozwiązania

Ile rozmieszczeń? Bardzo dużo… n^n

Rozwiązanie 2’

(trochę mniej) naiwne

Struktura danych:

int b[n]

b[i] jest równe pozycji (wierszowi) hetmana z kolumny i.

Algorytm:

1. Dla każdego możliwego rozmieszczenia *n* hetmanów na szachownicy *n*×*n* takiego, że w każdej kolumnie jest jeden hetman:
   – Sprawdź czy jakaś para hetmanów atakuje się nawzajem (jeśli nie – zwróć rozwiązanie i **zakończ**)
2. Zwróć informację o braku rozwiązania

Rozwiązanie 2’

Struktura danych:

int b[n]

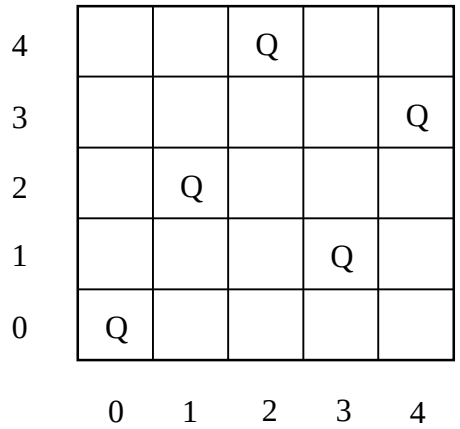
b[i] jest równe pozycji (wierszowi) hetmana z kolumny i.

Przykład

$n = 5$

i	0	1	2	3	4
***a*[*i*]**	0	2	4	1	3

Szachownica:



Rozwiązanie 3: nawroty

Algorytm (idea):

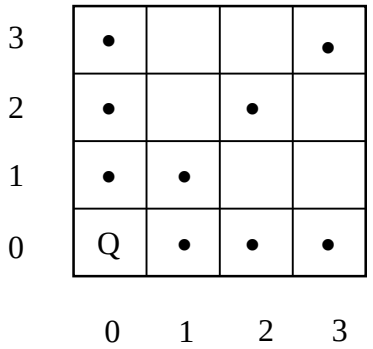
1. Próbuj stopniowo rozmieszczać hetmany w kolejnych kolumnach, zaczynając od skrajnie lewej.
2. Umieszczając hetmana w kolumnie i, znajdź pierwszą pozycję („od dołu”) która nie jest atakowana przez hetmany z kolumn 0, 1,…,i – 1
3. Jeśli nie ma takiej „wolnej” pozycji w kolumnie i, wróć do kolumny i – 1 i spróbuj ustawić hetmana na innej pozycji w tej kolumnie (wyżej niż poprzednio); jeśli to niemożliwe, wróć do kolumn i – 2, i – 3, …
4. Jeśli udało się ustawić hetmana w ostatniej kolumnie – zwróć wynik i zakończ działanie!
5. Jeśli nie ma już „nowych” (wyższych) pozycji w skrajnie lewej kolumnie – zakończ z informacją o braku ustawienia hetmanów.

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 0



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i] = -1 oznacza brak hetmana w i-tej kolumnie):

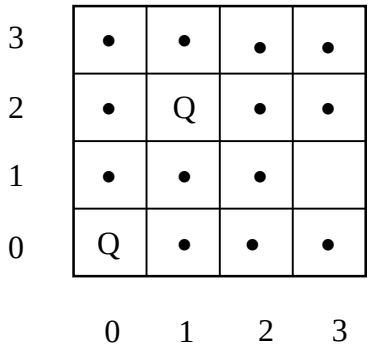
i	0	1	2	3
b[i]	0	-1	-1	-1

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 1



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i] = -1 oznacza brak hetmana w i-tej kolumnie):

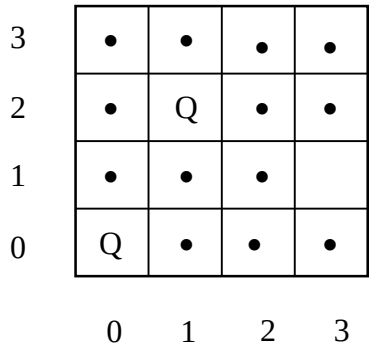
i	0	1	2	3
b[i]	0	2	-1	-1

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 2



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

i	0	1	2	3
b[i]	0	2	-1	-1

Problem!

Brak wolnej pozycji w kolumnie 2!

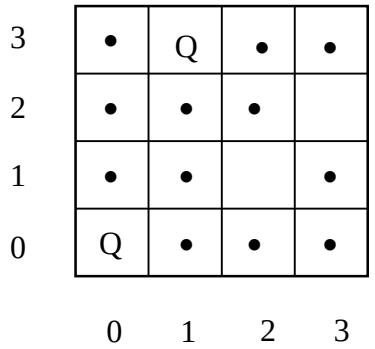
Wróć do kolumny 1!

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 1



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

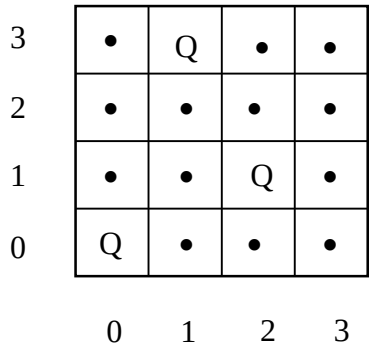
i	0	1	2	3
b[i]	0	3	-1	-1

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 2



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

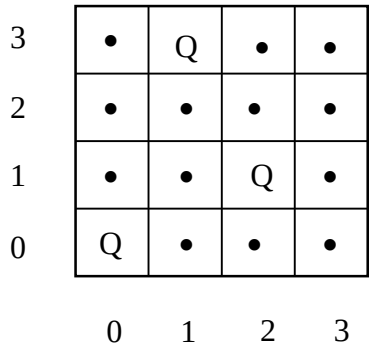
i	0	1	2	3
b[i]	0	3	1	-1

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 3



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

i	0	1	2	3
b[i]	0	3	1	-1

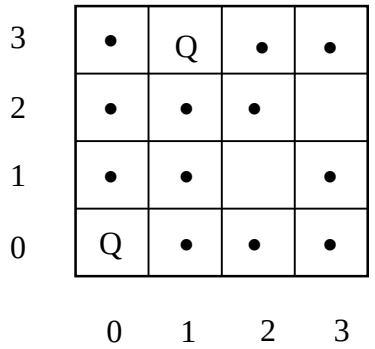
Problem!
Brak wolnej pozycji w kolumnie 3!
Wróć do kolumny 2!

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 2



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

i	0	1	2	3
b[i]	0	3	1	-1

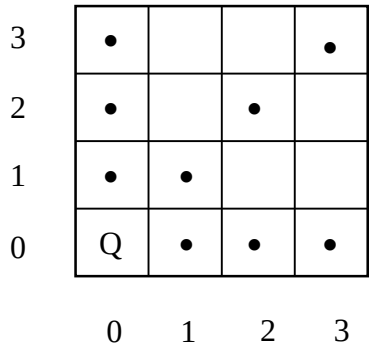
Problem!
Brak wolnej „wyższej” pozycji w kolumnie 2!
Wróć do kolumny 1!

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 1



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

i	0	1	2	3
b[i]	0	3	-1	-1

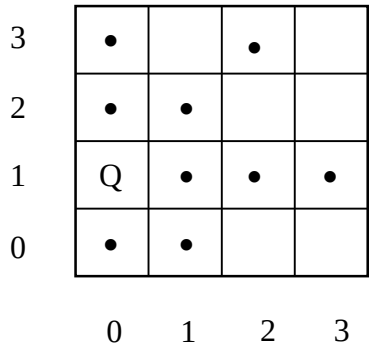
Problem!
Brak wolnej „wyższej” pozycji w kolumnie 1!
Wróć do kolumny 0!

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 0



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

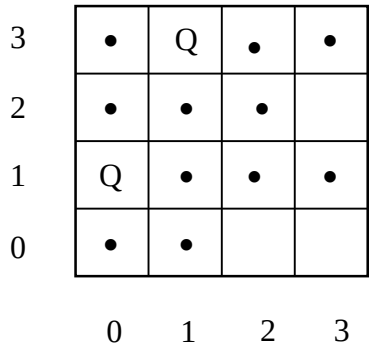
i	0	1	2	3
b[i]	1	-1	-1	-1

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 1



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

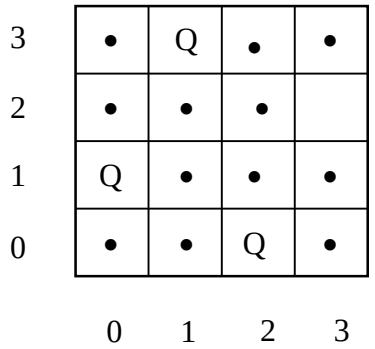
i	0	1	2	3
b[i]	1	3	-1	-1

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 2



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

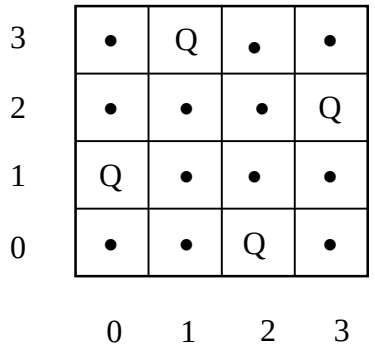
i	0	1	2	3
b[i]	1	3	0	-1

Rozwiązanie 3: nawroty

Przykład

$n = 4$

Kolumna: 3



Znak „•” oznacza pozycję atakowaną przez któryś z ustawionych już hetmanów

Struktura danych (b[i]=-1 oznacza brak hetmana w i-tej kolumnie):

i	0	1	2	3
b[i]	1	3	0	2

Przeszukiwanie z nawrotami

Algorytm (idea trochę dokładniej):

1. Umieść -1 w komórkach b[0], b[1], …, b[n – 1] //hetmany „pod szachownicą”
2. k ← 0; // k oznacza „bieżącą” kolumnę
3. dopóki (k < n oraz k ≥ 0) // k ==n: znaleziono rozw.; k<0: brak rozw.
   1. Jeśli istnieje wolne pole i w kolumnie k powyżej pozycji b[k]:
      - Umieść hetmana na (najniższym wolnym powyżej b[k]) polu i w kolumnie k, przejdź do kolejnej kolumny:

        b[k] ← i

        k ← k + 1
   2. w przeciwnym przypadku:
      - Usuń hetmana z kolumny k, wróć do kolumny k – 1:

        b[k] ← – 1 //usuń hetmana z kolumny k

        k ← k – 1 //wróć do poprzedniej kolumny

Przypomnienie: pole jest wolne, gdy nie jest atakowane przez żadnego hetmana.

Problem hetmanów - implementacja

Jak sprawdzić czy pole jest atakowane przez hetmana?

Wejście

(x, y) – sprawdzana pozycja;

(a, b) – pozycja hetmana, a ≠ x. (inna kolumna)

Wyjście

1: gdy hetman na (a, b) atakuje (x,y),

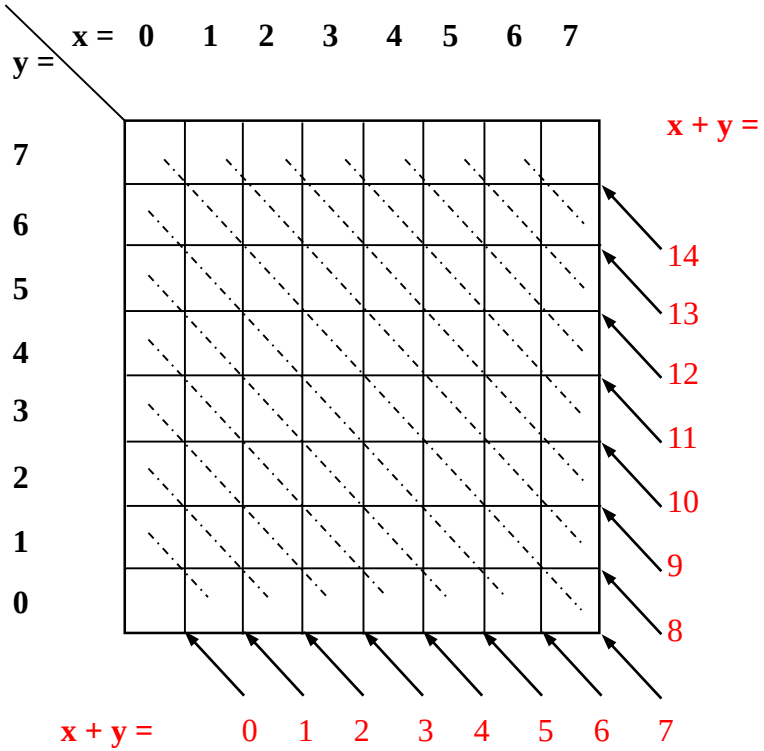
0: w przeciwnym przypadku.

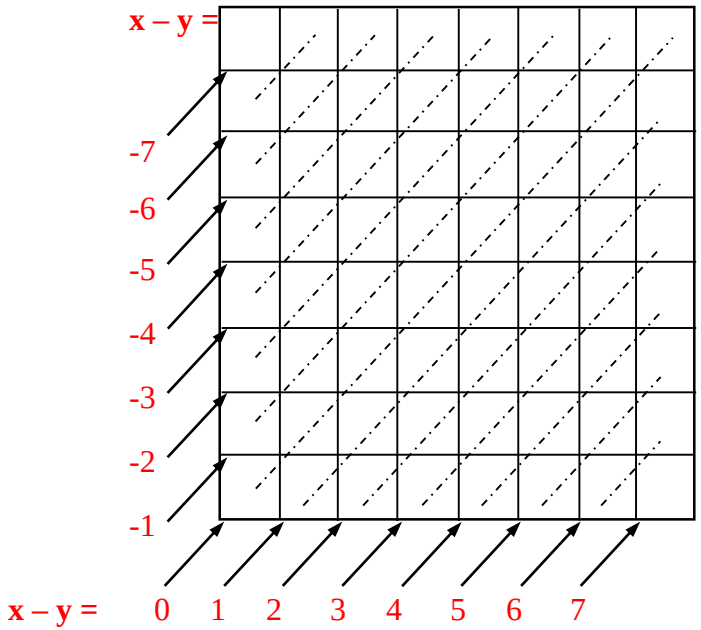
Musimy sprawdzić:

- czy (x,y) i (a,b) są w tym samym wierszu: y==b?
- czy (x,y) i (a,b) są na tej samej przekątnej???

Problem hetmanów - nawroty - implementacja

Jak sprawdzić czy (x,y) i (a,b) są na tej samej przekątnej?





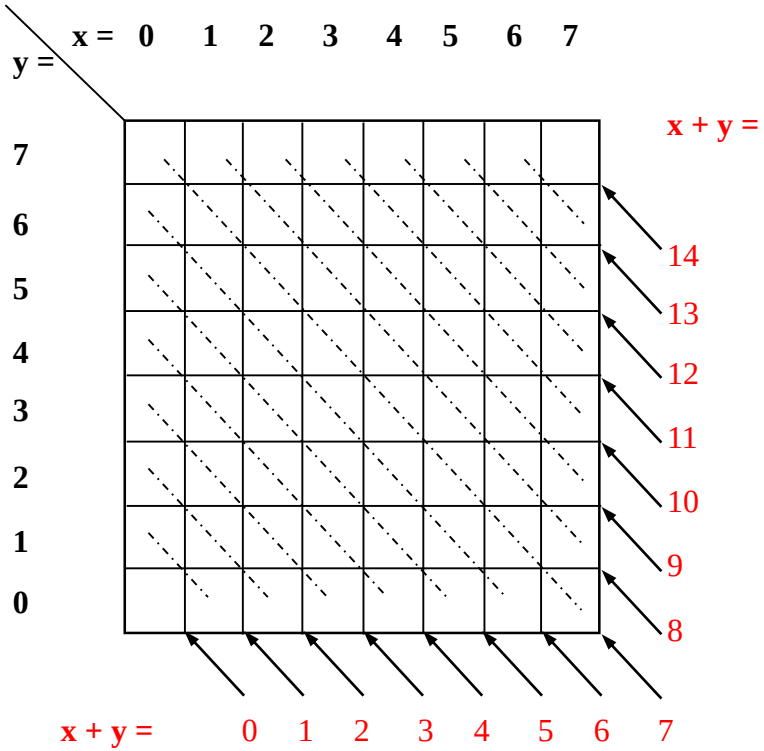
Obserwacja:

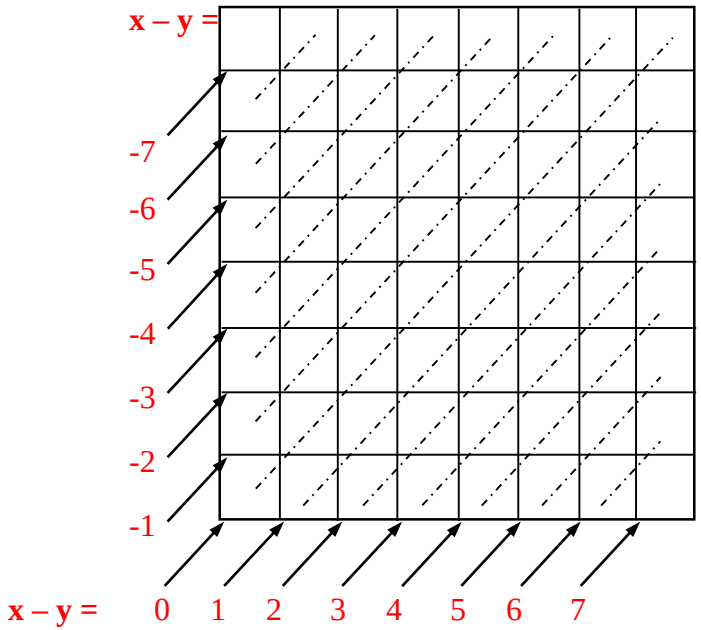
Wszystkie pola na (tej samej) przekątnej mają:

- równą ***sumę*** współrzędnych (lewa przekątna) **LUB**
- równą ***różnicę*** współrzędnych (prawa przekątna)

Problem hetmanów - nawroty - implementacja

Jak sprawdzić czy (x,y) i (a,b) są na tej samej przekątnej?





Wniosek

(x,y) i (a,b) znajdują się na tej samej przekątnej ⇔

$x + y = a + b$ **LUB** $x - y = a - b$

Problem hetmanów - nawroty - implementacja

Inicjalizacja

```
void init()
//Umieść -1 w komórkach b[0], b[1], …, b[n – 1]
{   int i;
    for (i=0; i<n; i++) b[i]=-1; //wszystkie hetmany są „pod” szachownicą
}
```

```
def init():
#Umieść -1 w komórkach b[0], b[1], …, b[n – 1]
    for i in range(n):
        b[i]=-1; #wszystkie hetmany są „pod” szachownicą
```

Problem hetmanów - nawroty - implementacja

Nawroty:

1. $k \leftarrow 0$; // k oznacza „bieżącą” kolumnę
2. dopóki ($k < n$ oraz $k \geq 0$) // k ==n: znaleziono rozw.; k<0: brak rozw.
   1. Jeśli jest wolne pole i w kolumnie k powyżej pozycji b[k]:
      - Umieść hetmana na (najniższym wolnym powyżej b[k]) polu i w kolumnie k, przejdź do kolejnej kolumny:

        $b[k] \leftarrow i$

        $k \leftarrow k + 1$
   2. w przeciwnym przypadku:
      - Usuń hetmana z kolumny k, wróć do kolumny k – 1:

        $b[k] \leftarrow -1$ //usuń hetmana z kolumny k

        $k \leftarrow k - 1$ //wróć do poprzedniej kolumny

Problem hetmanów - nawroty - implementacja

Nawroty

```
int queens()
// algorytm z nawrotami
// queens() = n gdy znalezione rozwiązanie
// queens() = -1 gdy brak rozwiązania
{   int k;
    b[0]=0;  // umieść pierwszego hetmana w „lewym dolnym” narożniku (na dole kolumny 0)
    k=1;  // przejdź do kolumny numer 1
    while (k<n && k>=0)
    {   do  //poszukiwanie „wolnego” pola w kolumnie k, idąc w górę od b[k]
         { b[k]++; }
        while (b[k]<n && !isfree(k,b[k]));
        if (b[k]<n) k++;            //wolne pole znalezione, przejście do nast. kolumny!!!
        else {b[k]=-1; k--;}        //brak wolnego pola, powrót do poprzedniej kolumny!
    }
  return k;
}
```

Uwaga: *k* oznacza aktualnie „badaną” kolumnę.

Problem hetmanów - nawroty - implementacja

Nawroty

```
def queens():
# algorytm z nawrotami
# queens() = n gdy znalezione rozwiązanie
# queens() = -1 gdy brak rozwiązania
    b[0]=0; #umieść pierwszego hetmana w „lewym dolnym” narożniku (na dole kolumny 0)
    k=1 # przejdź do kolumny numer 1
    while k<n and k>=0:
        #poszukiwanie „wolnego” pola w kolumnie k, idąc w górę od b[k]
        b[k]+=1
        while b[k]<n and not isfree(k,b[k]):
            b[k]+=1
        if b[k]<n: k+=1              #wolne pole znalezione, przejście do następnej kolumny!!!
        else:
            b[k]=-1; k-=1            #brak wolnego pola, powrót do poprzedniej kolumny
    return k
```

Uwaga: *k* oznacza aktualnie „badaną” kolumnę.

Problem hetmanów - nawroty - implementacja

Nawroty

```
int queens()
{    int k;
   b[0]=0;
   k=1;
   while (k<n && k>=0)
   {  do //poszukiwanie wolnego pola w kolumnie k, zaczynając „od dołu”
       { b[k]++; }
      while (b[k]<n && !isfree(k,b[k]));
      if (b[k]<n) k++;
      else {b[k]=-1; k--;}
   }
  return k;
}
```

Uwaga: wartość b[k] po „niebieskiej” pętli jest równa:

- najniższemu polu (powyżej pól dotychczas „sprawdzonych”) w kolumnie k, które nie jest atakowane przez hetmany z wcześniejszych kolumn, LUB
- *n*, gdy nie ma „wolnego” pola w kolumnie k (powyżej pól dotychczas „sprawdzonych”), czyli nieatakowanego przez hetmany z wcześniejszych kolumn.

Problem hetmanów - nawroty - implementacja

Nawroty

```
int queens()
{    int k;
   b[0]=0; k=1;
   while (k<n && k>=0)
   {  do
        { b[k]++; }
      while (b[k]<n && !isfree(k,b[k]));
      if (b[k]<n) k++;          //wolne pole znalezione!
      else {b[k]=-1; k--;}      //brak wolnego pola, powrót do poprzedniej kolumny
   }
  return k;
}
```

Uwagi:

- `k++` i `k--` opisują przejście do następnej/poprzedniej kolumny
- Po przejściu do „poprzedniej” kolumny (z k do k – 1), zaczynamy ponowne „sprawdzanie” wszystkich pól w tej kolumnie. Stąd `b[k]=-1;` (umieść hetmana z kolumny k „pod” szachownicą)

Problem hetmanów - nawroty - implementacja

Sprawdź czy pozycja (x,y) jest atakowana przez hetmany z kolumn 0,1,…,x-1:

```
int isfree(int x, int y)
// isfree(x,y) = 1 wtw (x,y) nie jest atakowane
// przez hetmany z kolumn 0,1,2,...,x-1
{ int i;
  for (i=0; i<x; i++)
    if (b[i]-i==y-x || b[i]+i==y+x || b[i]==y) return 0;
  return 1;
}
```

```
def isfree(x, y):
# isfree(x,y) = 1 wtw (x,y) nie jest atakowane
# przez hetmany z kolumn 0,1,2,...,x-1
    for i in range(x):
        if b[i]-i==y-x or b[i]+i==y+x or b[i]==y:
            return 0
    return 1
```

Przypomnienie

Pozycja hetmana z kolumny *i* jest równa (*i*, b[*i*]).

Problem hetmanów - nawroty - implementacja

Prezentacja wyników

```
void drawresult()
{ int i, j;
  printf("\n");
  for (i=0; i<n; i++)  // prosty opis
    printf("%d ", b[i]);

  printf("\n\n");
  for(i=0; i<n; i++)  // cała szachownica
   { for(j=0; j<n; j++)
        if (b[j]==i) printf("x ");
        else printf("o ");
     printf("\n");
   }
}
```

```
def drawresult():
  print("\n");
  for i in range(n): # prosty opis
     print b[i],

  print "\n"
  for i in range(n): #cała szach.
      for j in range(n):
       if b[j]==i: print "x ",
       else: print "o ",
    print
```

Problem hetmanów - nawroty - implementacja

```
#define n 6  //rozmiar szachownicy
int b[n];
void init()
{int i;
 for (i=0; i<n; i++) b[i]=-1;
}
int isfree(int x, int y)
{int i;
 for (i=0; i<x; i++)
   if (b[i]-i==y-x || b[i]+i==y+x ||
      b[i]==y) return 0;
 return 1;
}
int queens()
{int k=1; b[0]=0;
 while (k<n && k>=0)
 {do
   {b[k]++;}
  while (b[k]<n && !isfree(k,b[k]));
  if (b[k]<n) k++;  else {b[k]=-1;k--;}
 }
 return k;
}
```

```
void drawresult()
{ int i, j;
  printf("\n");
  for (i=0; i<n; i++)
    printf("%d ", b[i]);
  printf("\n\n");
  for(i=0; i<n; i++)
   { for(j=0; j<n; j++)
        if (b[j]==i) printf("x ");
        else printf("o ");
     printf("\n");
   }
}

main()
{
  init();
  if (queens()==n) drawresult();
  else printf("brak rozwiazania");
  system("pause");
}
```

Problem hetmanów - nawroty - implementacja

```
def init():
     for i in range(n):
        b[i]=-1;

def isf( x, y):
  for i in range(x):
    if b[i]-i==y-x or b[i]+i==y+x or
       b[i]==y:
         return 0;
  return 1;

def queens():
  b[0]=0
  k=1
  while k<n and k>=0:
    b[k]+=1
    while b[k]<n and not isf(k,b[k]):
      b[k]+=1
    if b[k]<n: k+=1
    else: b[k]=-1; k-=1
  return k
```

```
def drawresult():
  print("\n");
  for i in range(n):
    print b[i],

  print
  for i in range(n):
    for j in range(n):
       if b[j]==i: print "x",
       else: print "o",
    print



n=6  # rozmiar szachownicy
b=Array(n)
init()
if queens()==n:
    drawresult()
else: print("brak rozwiazania")
```

Hetmany – przeszukiwanie z nawrotami - złożoność

$O(n^n)$… można lepiej oszacować (przez mniejszą funkcję)?

Problem hetmanów – rekurencyjnie…

Idea rozwiązania:

- ustawiamy hetmany w kolejnych kolumnach
- ustawienie w kolumnach k, k+1,…,n to:
  - ustawienie w kolumnie k-tej
  - ustawienie w kolumnach k+1,k+2,…,n (rekurencja) – o ile k nie jest ostatnią kolumną

Wejście:

- n – rozmiar szachownicy
- k – pozycja kolumny od której jeszcze nie ustawiliśmy hetmanów
- a – tablica opisująca pozycje już ustawionych hetmanów (czyli w kolumnach 0, 1,…, k – 1)

Problem hetmanów – rekurencyjnie…

Idea rozwiązania – dokładniej:

1. ustawiamy hetmany w kolejnych kolumnach
2. ustawienie w kolumnach k, k+1,…,n gdy $k < n$:
   dla kolejnych poprawnych ustawień w kolumnie k-tej….
   - sprawdzamy rekurencyjnie czy możliwe jest poprawne ustawienie w kolumnach k+1,k+2,…,n.
3. ustawienie w kolumnach k, k+1,…,n gdy $k = n$:
   ustawiliśmy już wszystkie hetmany, koniec!

Uwaga: numery kolumn to 0, 1, 2,…., n – 1.

Problem hetmanów – rekurencyjnie…

```
int hetmany(int n, int k, int a[])
{
  if (k==n) return 1;
  for(int i=0; i<n; i++)  {
    if isf(k,i, a) {
     a[k]=i;
      if (hetmany(n,k+1,a)) return 1;
   }
  }
  return 0;
}
```

```
def hetmany(n, k, a):
  if k==n: return 1
  for i in range(n):
    if isf(k,i, a):
      a[k]=i
      if (hetmany(n,k+1,a)): return 1
  return 0
```

Uwaga! Funkcja `isf` różni się od funkcji `isfree` tym, że tablica z pozycjami hetmanów jest parametrem wywołania!

Problem hetmanów – rekurencyjnie…

Pytania:

- Jak zastąpić w naszym programie funkcję queens przez funkcję hetmany?
- Z jakimi parametrami wywołać funkcję hetmany?
- Która wersja szybsza?
- Która wersja łatwiejsza w implementacji?

```
int hetmany(int n, int k, int a[])

int queens()
```

Problem hetmanów – rekurencyjnie…

Pytania:

- Jak zastąpić w naszym programie funkcję queens przez funkcję hetmany?
- Z jakimi parametrami wywołać funkcję hetmany?

```
int queens()
{   int k;
    b[0]=0; k=1;
    while (k<n && k>=0)
    {   do
          { b[k]++; }
        while (b[k]<n && !isfree(k,b[k]));
        if (b[k]<n) k++;
        else {b[k]=-1; k--;}
    }
  return k;
}
```

```
int queensRek()
{     int k;
      if hetmany(n,0,b)
          return n;
      else return -1;
}
```