

Techniki projektowania algorytmów

- Dziel i zwyciężaj (*divide and conquer*) – rekursja
- Redukcja (*reduction, transform and conquer*)
- Programowanie liniowe (*linear programming*)
- Programowanie zachłanne (*greedy method*)
- Programowanie dynamiczne (*dynamic programming*)
- Algorytmy przeszukujące (*trial and error*)
- Algorytmy probabilistyczne i heurystyki (*probabilistic, heuristic*)

Rekursja

- Rekursja – zobacz: **Rekursja** ;-)
- Rekursja – jeśli ciągle nie wiesz co to jest, zobacz: **Rekursja** ;-)
- PHP – „PHP: Hypertext Preprocessor”
- GNU – „GNU's Not Unix”
- AMARA – „Amara Means A Recursive Acronym”

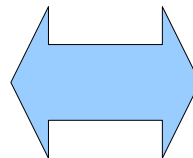
Rekursja

- $P \equiv \text{IF } B \text{ THEN } P[S,P] \text{ END}$
 - P – program
 - P – kompozycja (złożenie)
 - S – sekwencja rozkazów nie zawierająca P
- Wieże Hanoi, algorytm Euklidesa znajdowania NWD, definicja drzewa – patrz „Matematyka dyskretna”
- „Dziel i zwyciężaj” - quicksort, mergesort

Kiedy nie używać rekursji?

- $P \equiv \text{IF } B \text{ THEN } S; P \text{ END}$
(P na samym początku lub końcu)
- Silnia: $n! = n * (n-1)!$ $0! = 1$

```
PROCEDURE
F (I : INTEGER) : INTEGER;
BEGIN
  IF I > 0
  THEN
    RETURN I * F (I - 1);
  ELSE
    RETURN 1;
  END;
END F;
```

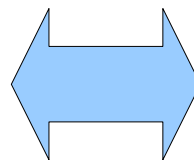


```
I := 0;
F := 1;
WHILE I < n DO
  I := I + 1;
  F := I * F;
END;
```

Kiedy nie używać rekursji?

- Ciąg Fibonacciego: $F(i) = F(i-1) + F(i-2)$
- Powtarzamy niepotrzebnie wiele obliczeń

```
PROCEDURE F (n:INTEGER) :  
INTEGER;  
BEGIN  
IF n=0  
  THEN RETURN 0;  
ELSE IF n=1  
  THEN RETURN 1;  
  ELSE  
    RETURN F (n-1) + F (n-2)  
  END;  
END;  
END F;
```



```
i := 1;  
x := 1;  
y := 0;  
WHILE i < n DO  
  x := x+y;  
  y := x-y;  
  i := i+1;  
END;
```

Krzywa Hilberta (*Hilbert curve*)

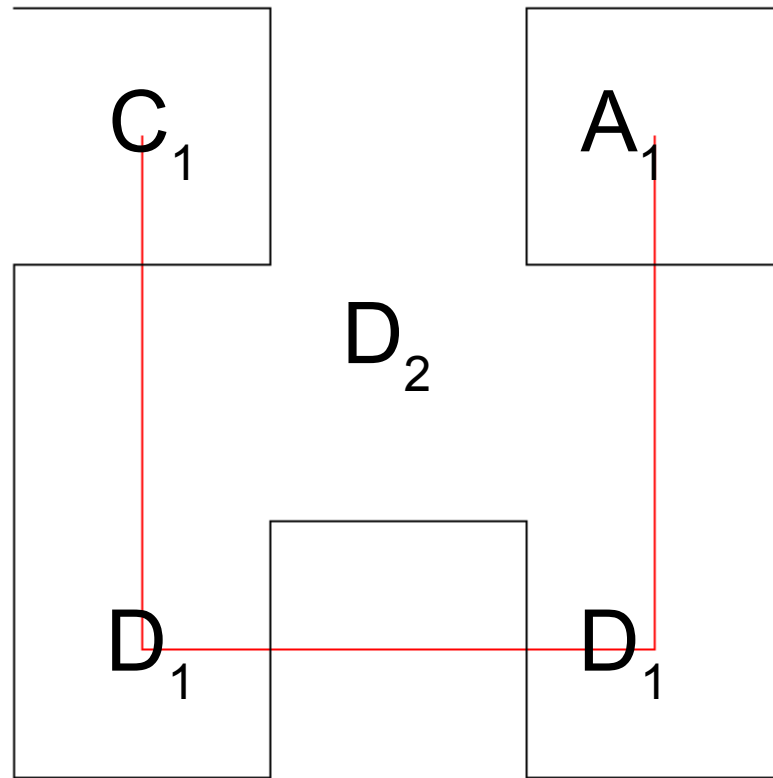
- Krzywa H_i rzędu i składa się z czterech instancji krzywych H_{i-1} dwukrotnie pomniejszonych, obróconych i połączonych odcinkami.
- H_0 jest pusta, stąd H_1 składa się tylko z trzech odcinków.
- $A_i: D_{i-1} \leftarrow A_{i-1} \downarrow A_{i-1} \rightarrow B_{i-1}$
- $B_i: C_{i-1} \uparrow B_{i-1} \rightarrow B_{i-1} \downarrow A_{i-1}$
- $C_i: B_{i-1} \rightarrow C_{i-1} \uparrow C_{i-1} \leftarrow D_{i-1}$
- $D_i: A_{i-1} \downarrow D_{i-1} \leftarrow D_{i-1} \uparrow C_{i-1}$

H_1

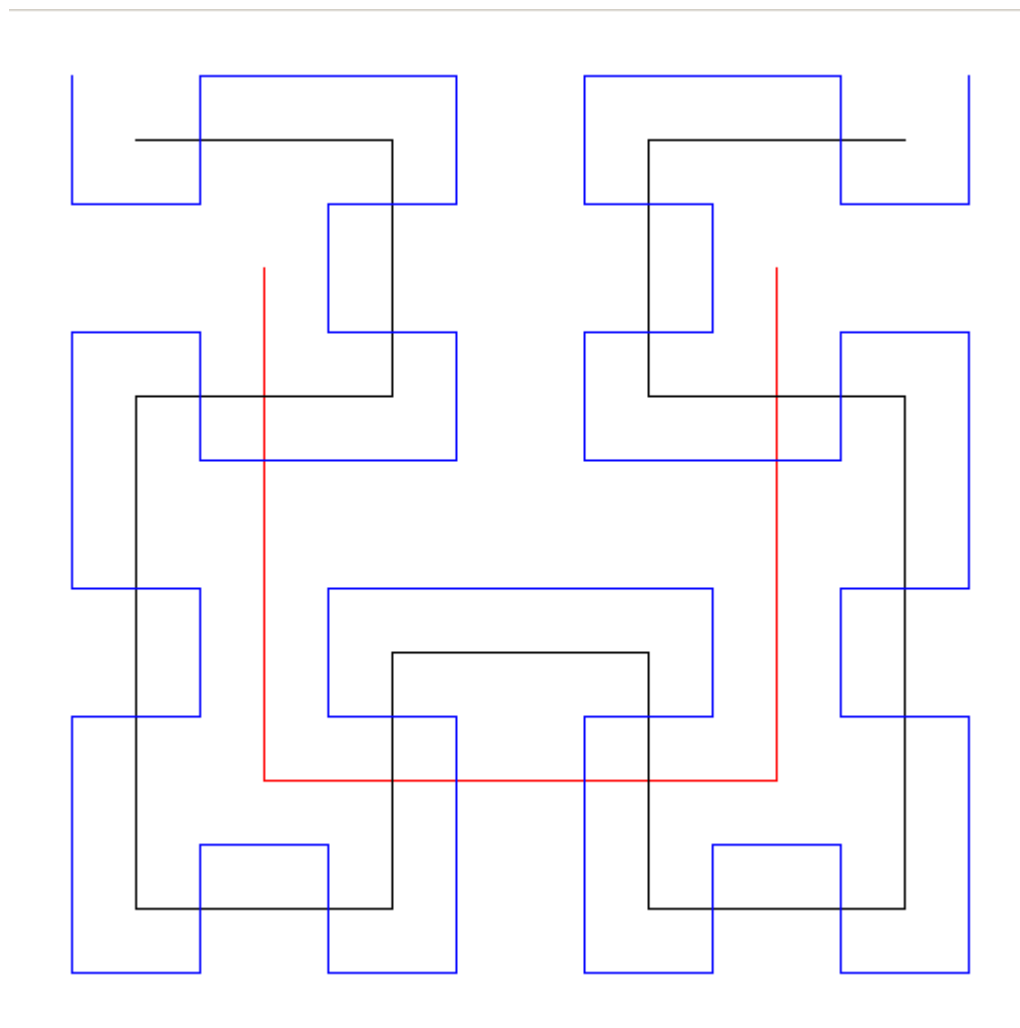


D_1

H_2



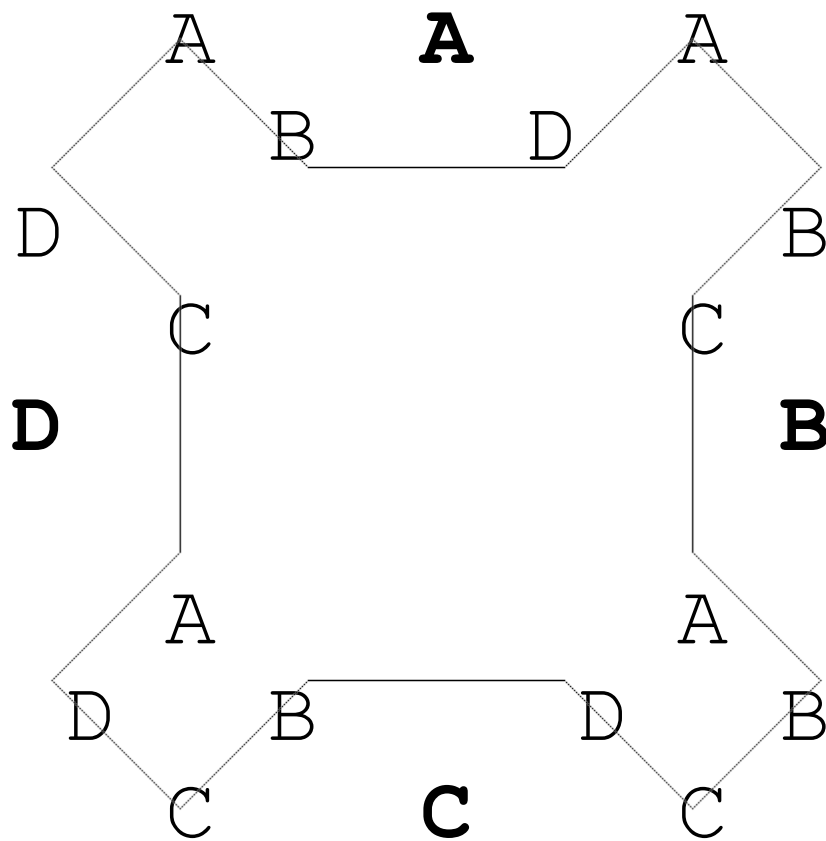
H_3



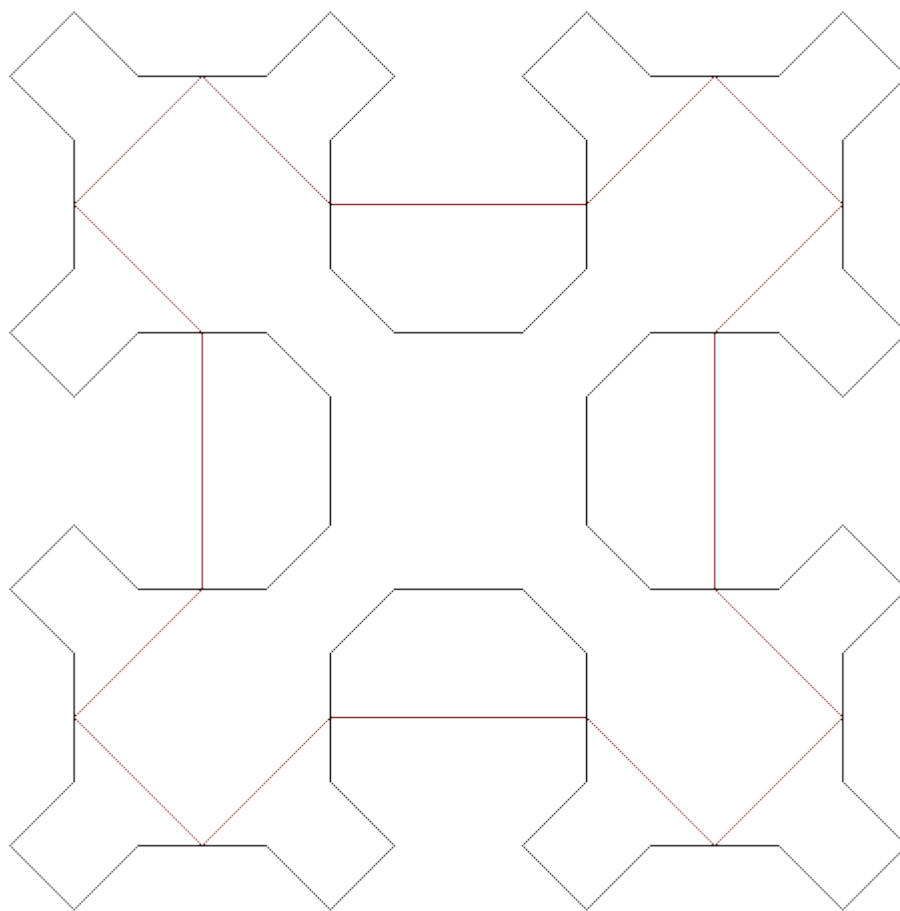
Krzywa Sierpińskiego

- Główny schemat różni się od schematów rekursji
- $S_n: A_n \searrow B_n \swarrow C_n \nwarrow D_n \nearrow$
- $A_i: A_{i-1} \searrow B_{i-1} \rightarrow D_{i-1} \nearrow A_{i-1}$
- $B_i: B_{i-1} \swarrow C_{i-1} \downarrow A_{i-1} \searrow B_{i-1}$
- $C_i: C_{i-1} \nwarrow D_{i-1} \leftarrow B_{i-1} \swarrow C_{i-1}$
- $D_i: D_{i-1} \nearrow A_{i-1} \uparrow C_{i-1} \nwarrow D_{i-1}$
- S_0 – kwadrat obrócony o 45 stopni (A_0, B_0, C_0, D_0 są puste)

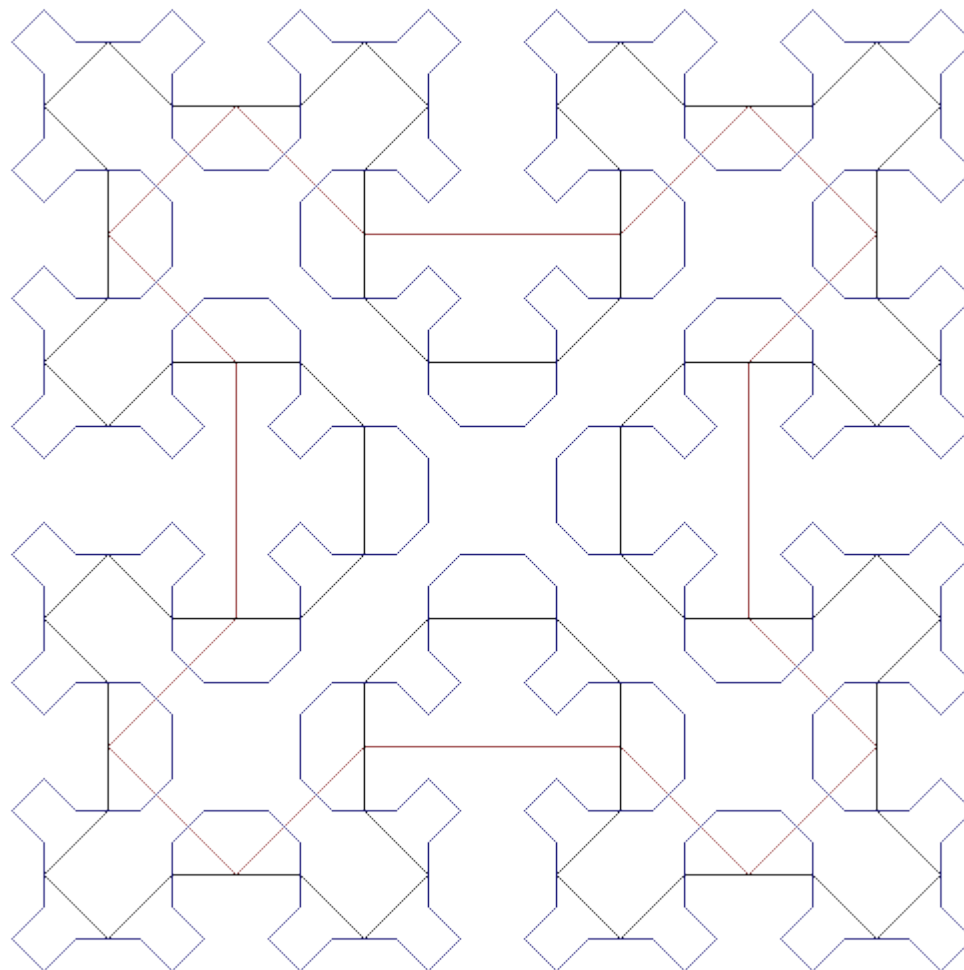
S_1



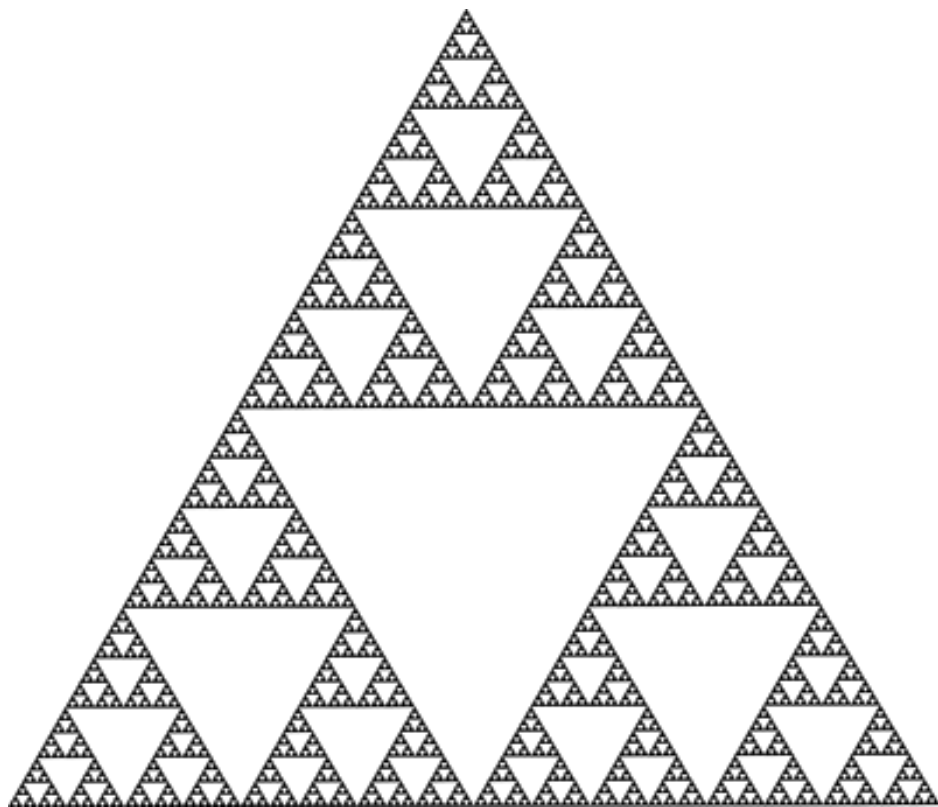
S_2



S_3

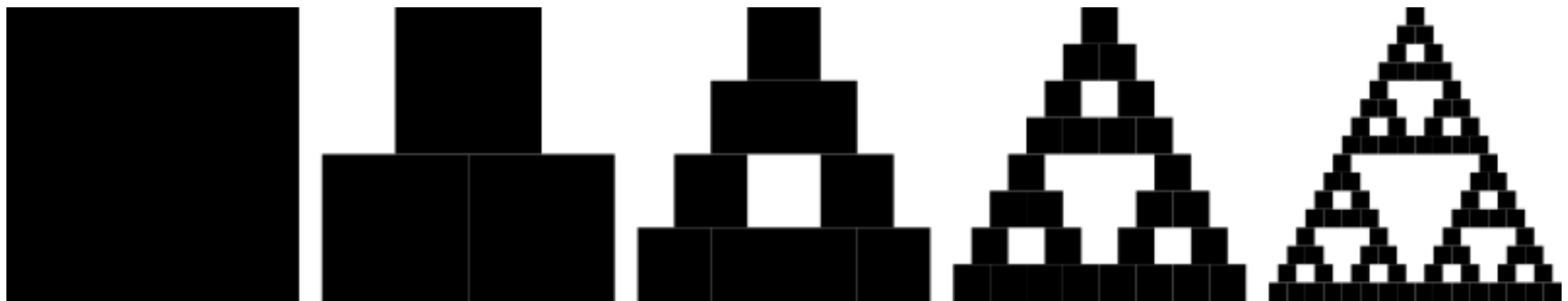


Trójkąt Sierpińskiego



- Weź dowolny kształt (zwykle trójkąt).
- Jeśli procedura osiągnęła założony poziom narysuj kształt. W przeciwnym przypadku zmniejsz rozmiar kształtu dwukrotnie. Wywołaj procedurę trzykrotnie na planie trójkąta.

Trójkąt Sierpińskiego



Redukcja (*reduction*)

- *transform and conquer* – „transformuj i zwyciężaj”
- np. zadanie znalezienia mediany w zbiorze liczb możemy rozwiązać następująco:
 - sortujemy zbiór (droga operacja)
 - wybieramy element środkowy (tania operacja)
- Chcemy pomnożyć dwie liczby. Mamy gotowe układy które potrafią: dodawać, odejmować, podnosić do kwadratu, dzielić przez 2.
- $a*b = ((a+b)^2 - a^2 - b^2)/2$

Programowanie liniowe (*linear programming*)

- Optymalizacja liniowej funkcji celu podlegającej liniowym ograniczeniom w postaci równości lub nierówności.
- Forma kanoniczna:

Maksymalizuj: $c^T x$
przy ograniczeniach: $Ax \leq b$
gdzie: $x \geq 0$
 x – wektor zmiennych
 c, b – wektory współczynników
 A – macierz współczynników

- Forma standardowa składa się z trzech części:
 - liniowej funkcji celu, np. maksymalizuj $c_1x_1+c_2x_2$
 - ograniczeń, np.

$$a_{11}x_1+a_{12}x_2\leq b_1$$

$$a_{21}x_1+a_{22}x_2\leq b_2$$

$$a_{31}x_1+a_{32}x_2\leq b_3$$
 - nieujemnych zmiennych np. $x_1\geq 0, x_2\geq 0$
- Ograniczenia inne niż mniejszościowe (\leq) możemy zamienić na mniejszościowe:
 - ograniczenie równościowe na dwa \leq i \geq
 - ograniczenie większościowe przez negację zmiennych.

Przykład

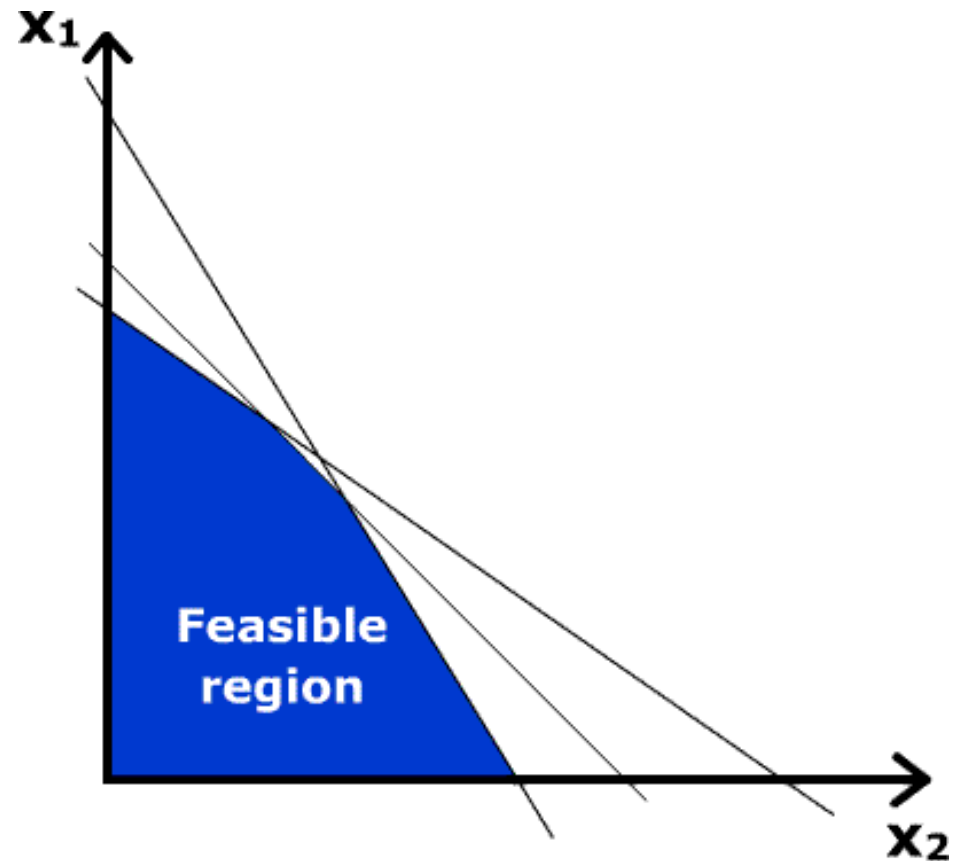
Rolnik ma pole o powierzchni A hektarów na którym może posadzić pszenicę lub jęczmień w dowolnej proporcji. Rolnik może zużyć maksymalnie F ton nawozu i P ton środka owadobójczego. Do uprawy pszenicy potrzeba F_1 a do jęczmienia F_2 ton nawozu na hektar. Do uprawy pszenicy potrzeba P_1 a do jęczmienia P_2 ton środka owadobójczego na hektar. Uprawa pszenicy daje S_1 , a jęczmienia S_2 złotych z hektara. Ile hektarów ma rolnik obsadzić pszenicą, a ile jęczmieniem aby uzyskać maksymalny zysk?

Przykład

- maksymalizuj: $S_1x_1 + S_2x_2$ (funkcja celu = zysk)
- ograniczenia:
 - $x_1 + x_2 \leq A$ (wielkość pola)
 - $F_1x_1 + F_2x_2 \leq F$ (ilość nawozu)
 - $P_1x_1 + P_2x_2 \leq P$ (ilość środka owadobójczego)
- nieujemne zmienne (nie da się uprawiać pola o ujemnej powierzchni):
 - $x_1 \geq 0$ (powierzchnia uprawy pszenicy)
 - $x_2 \geq 0$ (powierzchnia uprawy jęczmienia)

Programowanie liniowe – metody rozwiązywania

- Algorytm simplex – w najgorszym przypadku złożoność wykładnicza
- Metoda elipsoid – złożoność wielomianowa, ale w praktyce gorszy niż simplex
- Algorytm Karmarkara – złożoność wielomianowa, lepszy w praktyce niż simplex



Bajka o trzech złodziejach

Złodziej włamuje się do domu i widzi cztery wartościowe rzeczy:

- biżuterię o wadze 1 kg i wartości 15 PLN,
- żyrandol o wadze 5 kg i wartości 10 PLN,
- obraz o wadze 3 kg i wartości 9 PLN,
- radio o wadze 4 kg i wartości 5 PLN.

Jego plecak może udźwignąć max. 8 kg (obie ręce musi mieć wolne, żeby wyjść przez okno na piętrze). Co powinien zapakować do plecaka?

Mamy trzech złodziei: chciwego, powolnego i sprytnego.

Problem plecakowy

(wersja 0-1)

Maksymalizuj:
$$\sum_{i=1}^n p_i x_i$$

przy ograniczeniach:

$$\sum_{i=1}^n w_i x_i \leq c \quad x_i \in \{0,1\} \quad i=1,\dots,n$$

W wersji ciągłej można dzielić przedmioty pakowane do plecaka na części. Wówczas algorytm zachłanny jest optymalny.

Algorytm zachłanny (*greedy*)

- Wrzucamy przedmioty do plecaka zaczynając od przedmiotu o największej wartości (albo stosunku wartość/waga – lepsze, dla problemu ciągłego daje rozwiązanie optymalne).
- Złodziej wrzuca więc kolejno biżuterię (1 kg, 15 PLN) i żyrandol (5 kg, 10 PLN). Plecak może jeszcze udźwignąć 2 kg, ale nie ma tak lekkich przedmiotów, więc złodziej ucieka, unosząc przedmioty o wartości 25 PLN.

Algorytm *brute force*

- *Brute force* – brutalna siła, sprawdzamy wszystkie rozwiązania.
- Powolny złodziej zaczyna sprawdzać wszystkie możliwe rozwiązania. Złożoność problemu wynosi $O(2^n)$, gdyż tyle mamy możliwych do utworzenia podzbiorów. Zanim złodziej policzył odpowiednie sumy wartości i wag przedmiotów, wpadła policja i go aresztowano.

Programowanie dynamiczne (*dynamic programming*)

$A(i,j)$ definiujemy jako maksymalną wartość plecaka o wielkości j , rozpatrując tylko i pierwszych przedmiotów. Szukamy $A(n,c)$. Możemy je znaleźć z następującej zależności rekurencyjnej:

$$A(i, j) = \begin{cases} 0 & \text{dla } i=0 \text{ lub } j=0 \\ A(i-1, j) & \text{dla } w_i > j \\ \max\{A(i-1, j), p_i + A(i-1, j-w_i)\} & \text{dla } w_i \leq j \end{cases}$$

A więc badamy, czy warto dołożyć i -ty przedmiot do plecaka o w_i mniejszego

Programowanie dynamiczne (*dynamic programming*)

| p_i/w_i | | 15/1 | 10/5 | 9/3 | 5/4 |
|-----------|------------|------------|------------|------------|------------|
| j | i=0 | i=1 | i=2 | i=3 | i=4 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 15 | 15 | 15 | 15 |
| 2 | 0 | 15 | 15 | 15 | 15 |
| 3 | 0 | 15 | 15 | 15< | 15 |
| 4 | 0 | 15 | 15 | 24 | 24< |
| 5 | 0 | 15 | 15< | 24 | 24< |
| 6 | 0 | 15 | 25 | 25< | 25< |
| 7 | 0 | 15 | 25 | 25< | 25< |
| 8 | 0 | 15 | 25 | 25< | 29 |

Programowanie dynamiczne (*dynamic programming*)

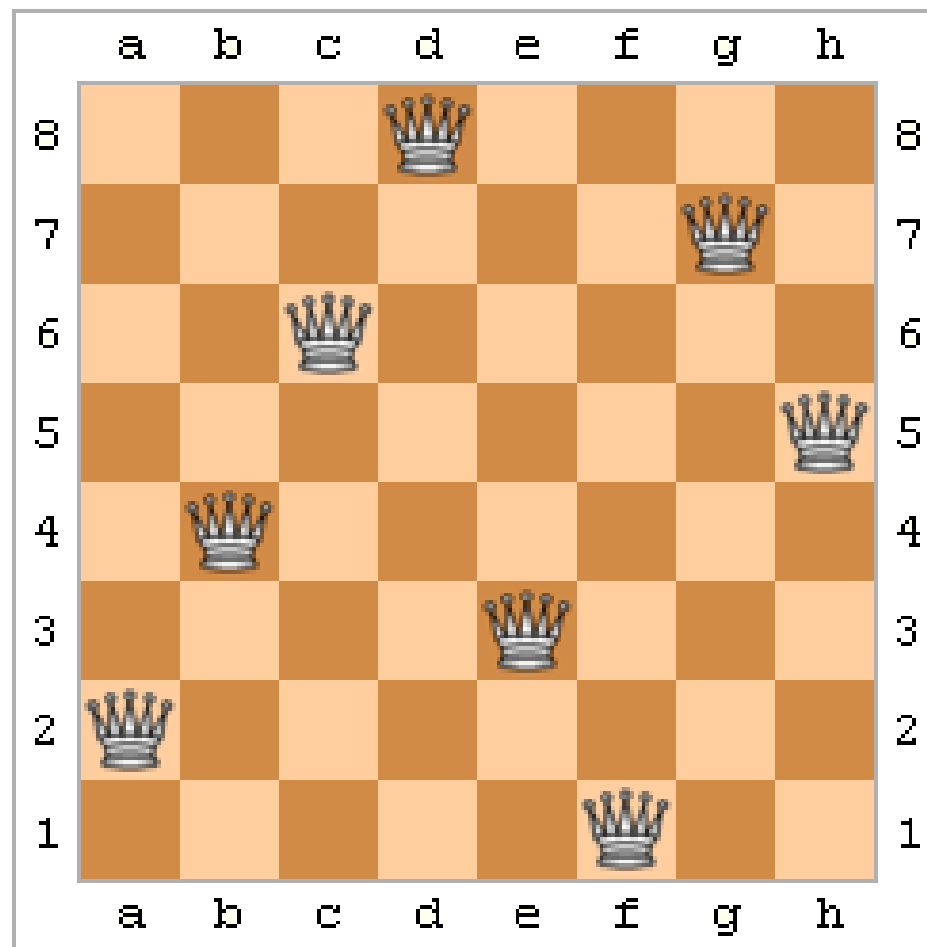
- Sprytny złodziej uzyskał plecak o wartości 29 PLN (4 PLN lepiej niż jego chciwy kolega), zabierając biżuterię, obraz i radio, ważące w sumie 8 kg.
- Złożoność wynosi $O(n \cdot c)$ w porównaniu z $O(2^n)$ algorytmu *brute force*.
- Programowanie dynamiczne możemy wykorzystać np. także w problemie obliczania ciągu Fibonacciego, poprzez zapamiętywanie kolejnych wyrazów ciągu.

Algorytmy przeszukujące przestrzeń rozwiązań (*trial and error*)

- *Brute force* – brutalna siła, sprawdzamy wszystkie rozwiązania.
- *Backtracking* – przeszukiwanie z nawrotami, wykorzystujemy własności zadania aby ograniczyć przeszukiwaną przestrzeń.
- *Branch and bound* – algorytm podziału i ograniczeń, wykorzystujemy własności zadania aby ograniczyć przeszukiwaną przestrzeń.

Problem ośmiu hetmanów

- Rozstawić osiem hetmanów na tradycyjnej szachownicy 8x8 tak, aby wzajemnie się nie atakowały (w pionie, poziomie i po przekątnej)
- Są 92 rozwiązania, 12 jeśli wykluczymy symetryczne i obrócone



Źródło: wikipedia.org

Rozwiązanie problemu – *brute force*

- Generujemy na ślepo wszystkie $64^8 = 2^{48} = 281,474,976,710,656$ możliwych ustawień hetmanów.
- Ponieważ dwa (i więcej) hetmany nie mogą zajmować jednego pola, możemy ograniczyć ilość możliwych ustawień do $64!/56 = 178,462,987,637,760$
- Jeszcze lepiej wygenerować wszystkie permutacje (pozycje w danym wierszu lub kolumnie) których jest $8! = 40320$, co eliminuje ataki w pionie i poziomie (jak dla wieży)

Rozwiązanie problemu – *backtracking*

- Przeszukujemy drzewo rozwiązań oparte na permutacjach. Stosujemy metodę przeszukiwania w głąb. Wystąpienie konfliktu po przekątnej powoduje „odcięcie” całego poddrzewa rozwiązań, dzięki czemu przeszukujemy tylko 15720 ustawień.

Algorytm podziału i ograniczeń (*branch and bound*)

- Dzielimy problem na podproblemy (*branching*), a następnie obliczamy górne i dolne ograniczenia (*bounding*) i „obcinamy” (*pruning*) niektóre gałęzie drzewa przeszukiwań.
- Dla problemu maksymalizacji: jeśli dla danego poddrzewa T jego górne ograniczenie (*upper bound*) jest mniejsze od dolnego ograniczenia (*lower bound*) dowolnego innego poddrzewa, to nie ma sensu testować rozwiązań w T .
- Dolne ograniczenie to zwykle najlepsze dotąd znalezione rozwiązanie.

```

PROCEDURE Try(i, tw, av: INTEGER);
VAR av1: INTEGER;
BEGIN (*dołączamy przedmiot i do rozwiązania*)
  IF tw + obj[i].weight <= limw THEN
    s := s + {i};
    IF i < n THEN Try(i+1, tw + obj[i].weight, av)
    ELSIF av > maxv THEN maxv := av; opts := s
    END ;
    s := s - {i}
  END ;
  (*wyłączamy przedmiot i z rozwiązania*)
  IF av - obj[i].value > maxv THEN
    IF i < n THEN Try(i+1, tw, av - obj[i].value)
    ELSE maxv := av - obj[i].value; opts := s
    END
  END
END

```

END Try;

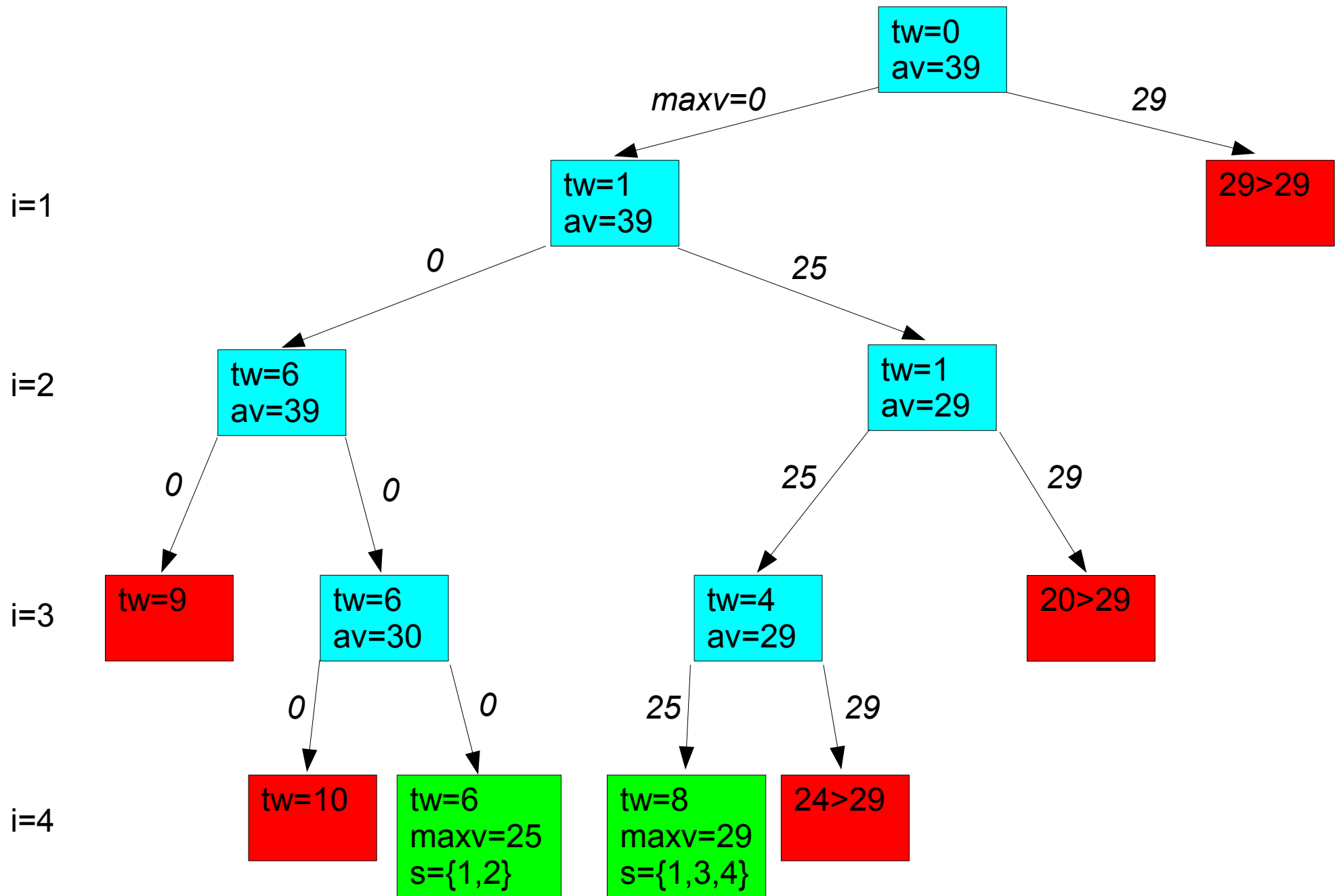
maxv - najlepsze rozwiązanie (dolne ograniczenie)

totv - suma wszystkich przedmiotów

av - górne ograniczenie (możliwe maksimum)

limw - wielkość plecaka

Wywołanie: maxv := 0; s := {}; opts := {};
 Try(1, 0, totv);



- Jak widać, przeszukiwanie przestrzeni rozwiązań metodą podziału i ograniczeń możliwe jest tylko dla problemów optymalizacyjnych. Nie nadaje się na przykład dla problemu ośmiu hetmanów.
- Można stosować inne metody obliczania ograniczeń. Np. można posortować przedmioty według stosunku wartość/waga, a następnie jako górne ograniczenie przyjąć wynik algorytmu zachłannego dla ciągłego problemu plecakowego.

- Złożoność algorytmu w najgorszym przypadku jest taka, jak *brute force*. Zwykle nadaje się do średniej wielkości instancji problemu.
- Dla problemu plecakowego metoda ta ma tę zaletę, iż pojemność plecaka i wagi przedmiotów nie muszą być całkowitoliczbowe (w przeciwieństwie np. do programowania dynamicznego).

Heurystyki

- Heurystyka - algorytm, który nie gwarantuje znalezienia optymalnego rozwiązania.
- Metaheurystyka – ogólna metoda rozwiązywania szerokiej klasy problemów, nie gwarantująca znalezienia optymalnego rozwiązania. Algorytm polega zwykle na przeszukiwaniu w odpowiedni sposób przestrzeni rozwiązań.

Metaheurystyki

- Przeszukiwanie losowe
- Przeszukiwanie lokalne (*local search*)
 - *Simple hill climbing*
 - *Steepest ascent hill climbing*
 - *Gradient descent/ascent*
- Przeszukiwanie tabu (*tabu search*)
- Symulowane wyżarzanie (*simulated annealing*)
- Algorytmy genetyczne (*genetic algorithms*)

No free lunch

- Średnia wydajność dowolnej pary algorytmów na wszystkich możliwych problemach jest identyczna.