TEORETYCZNE PODSTAWY INFORMATYKI

15/10/2018

WFAiS UJ, Informatyka Stosowana
I rok studiów, I stopień

Wykład 2

Problemy algorytmiczne

- **Klasy problemów algorytmicznych**
- **Liczby Fibonacciego**
- **Przeszukiwanie tablic**
- **Największy wspólny podzielnik**
- **Algorytmy zachłanne**

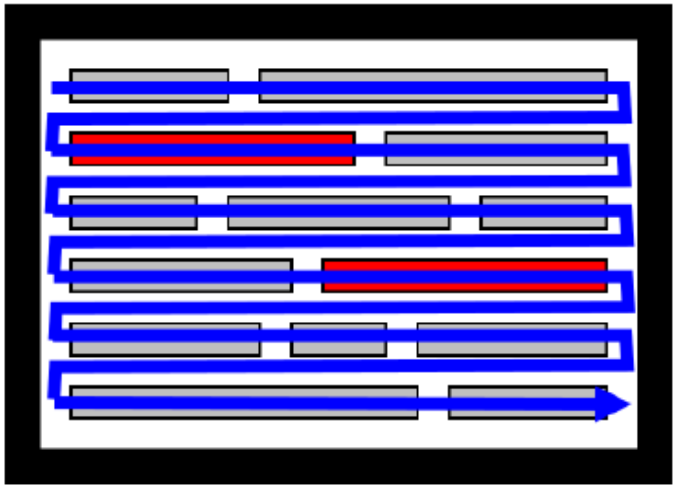
Materiał w oparciu o wykład: D. Kane, Univ. San Diego, USA „Data Structures and Algorithms, Algorithmic Toolbox"

Klasy problemów → typy algorytmów

- Mogą być prosto zaimplementowane w dowolnym języku programowania
- Proste (naturalne) rozwiązanie jest efektywne.

Prosty problem: liniowy scan

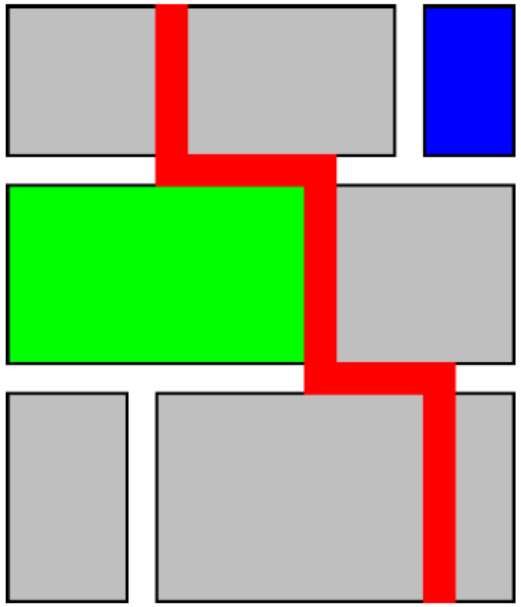
- Znajdź słowo w tekście.



- Przeszukujemy tekst liniowo
  - Prosty algorytm działa dobrze
  - Nie można/ nie ma powodu aby go poprawiać.

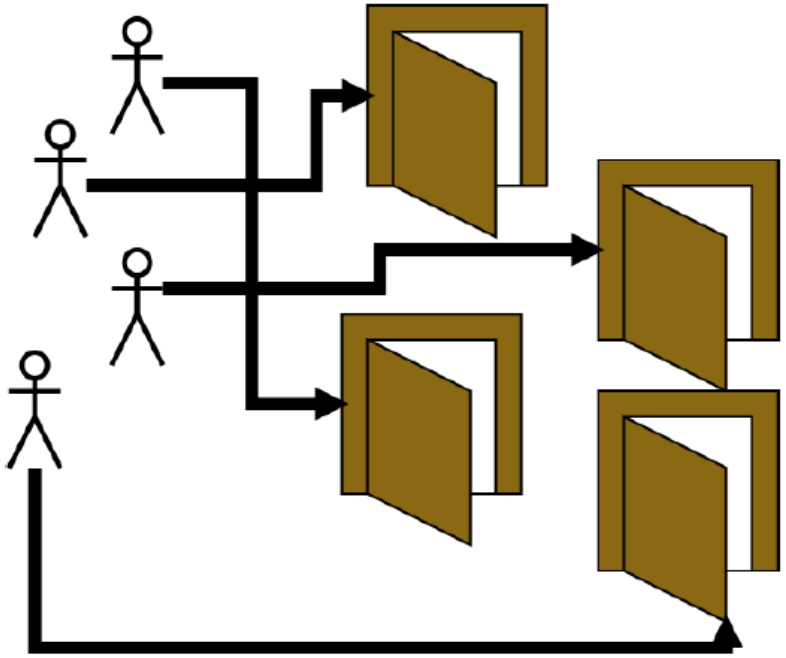
Algorytmiczny problem:

- Znajdź najkrótszą drogę pomiędzy dwoma punktami.



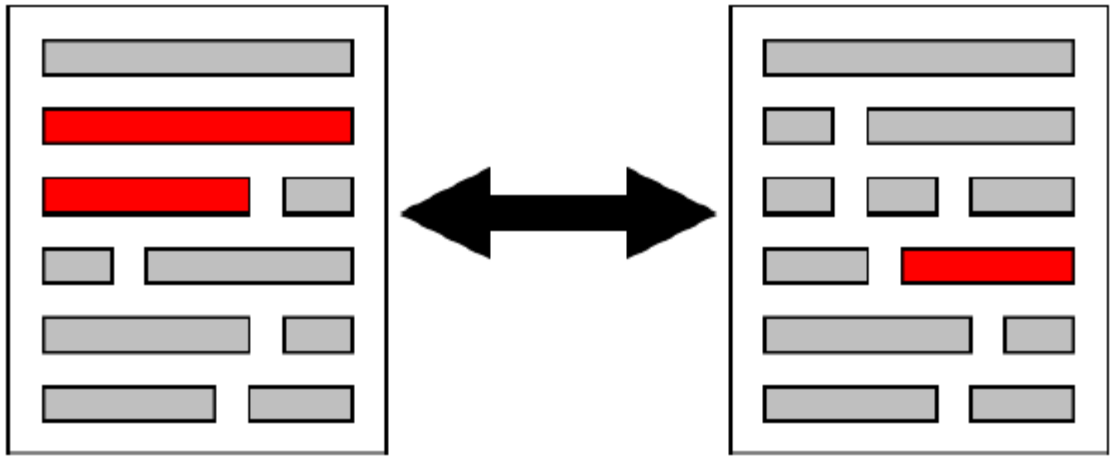
Algorytmiczny problem:

- Znajdź najlepsze przyporządkowanie studentów do pokoi w akademiku.



Algorytmiczny problem:

- Zdefiniuj miarę podobieństwa dwóch dokumentów.

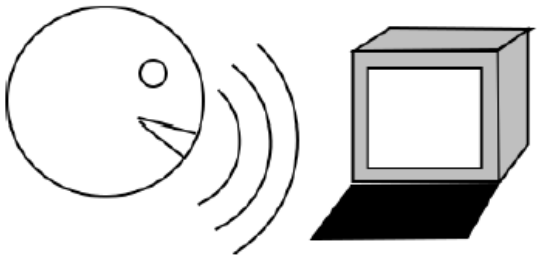


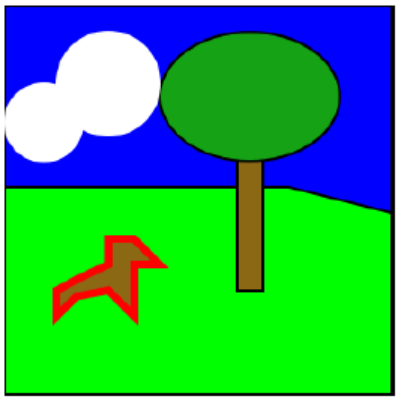
Algorytmiczny problem:

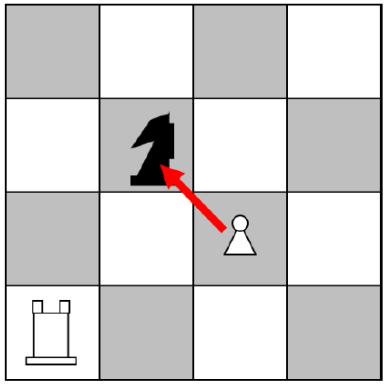
- Nie jest oczywiste jak rozwiązać.
- Proste pomysły są zbyt wolne.
- Można optymalizować rozwiązanie.

Problemy ze sztucznej inteligencji

- Rozumienie wypowiadanych słów
- Rozpoznawanie obrazów
- Gra w gry planszowe lub inne
- Itd…







Liczby Fibonacciego

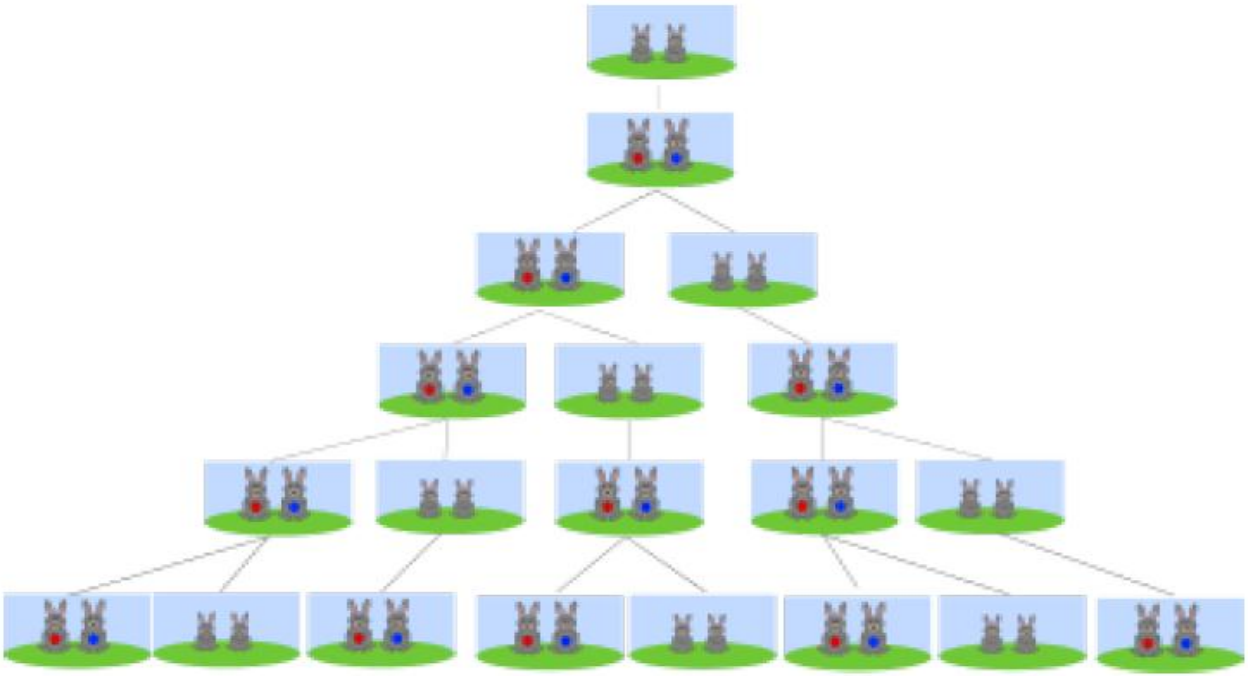
- Definicja

$$F_n = \begin{cases} 0, & n = 0\,, \\ 1, & n = 1\,, \\ F_{n-1} + F_{n-2}, & n > 1\,. \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Liczby Fibonacciego

- Przykład: studiowanie populacji królików



Liczby Fibonacciego

- Bardzo szybko rosną

$$F_n \geq 2^{n/2} \text{ for } n \geq 6\,.$$

- Dowód indukcyjny

By induction
Base case: $n = 6, 7$ (by direct computation).
Inductive step:

$$F_n = F_{n-1} + F_{n-2} \geq 2^{(n-1)/2} + 2^{(n-2)/2} \geq 2 \cdot 2^{(n-2)/2} = 2^{n/2}. \quad \square$$

Liczby Fibonacciego

- Przykład

$$
\begin{aligned}
F_{20} &= 6765 \\
F_{50} &= 12586269025 \\
F_{100} &= 354224848179261915075 \\
F_{500} &= 139423224561697880139724382870407283950070256587697307264108962948325571622863290691557658876222521294125
\end{aligned}
$$

Naiwny algorytm: rekurencyjny

- Funkcja rekurencyjna (pseudokod)

```
FibRecurs(n)

if n ≤ 1:
  return n
else:
  return FibRecurs(n − 1) + FibRecurs(n − 2)
```

- Czas wykonania: T(n) oznacza liczbę linii kodu które są wykonywane dla danej wartości n.

Naiwny algorytm: rekurencyjny

If $n \leq 1$

FibRecurs(n)

```
if n ≤ 1:
  return n
else:
  return FibRecurs(n − 1) + FibRecurs(n − 2)
```

$$T(n) = 2.$$

Naiwny algorytm: rekurencyjny

If $n \geq 2$

FibRecurs(n)

```
if n ≤ 1:
  return n
else:
  return FibRecurs(n − 1) + FibRecurs(n − 2)
```

$$T(n) = 3 + T(n-1) + T(n-2).$$

Naiwny algorytm: rekurencyjny

- Czas wykonania algorytmu: złożoność obliczeniowa

$$T(n) = \begin{cases} 2 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + 3 & \text{else.} \end{cases}$$

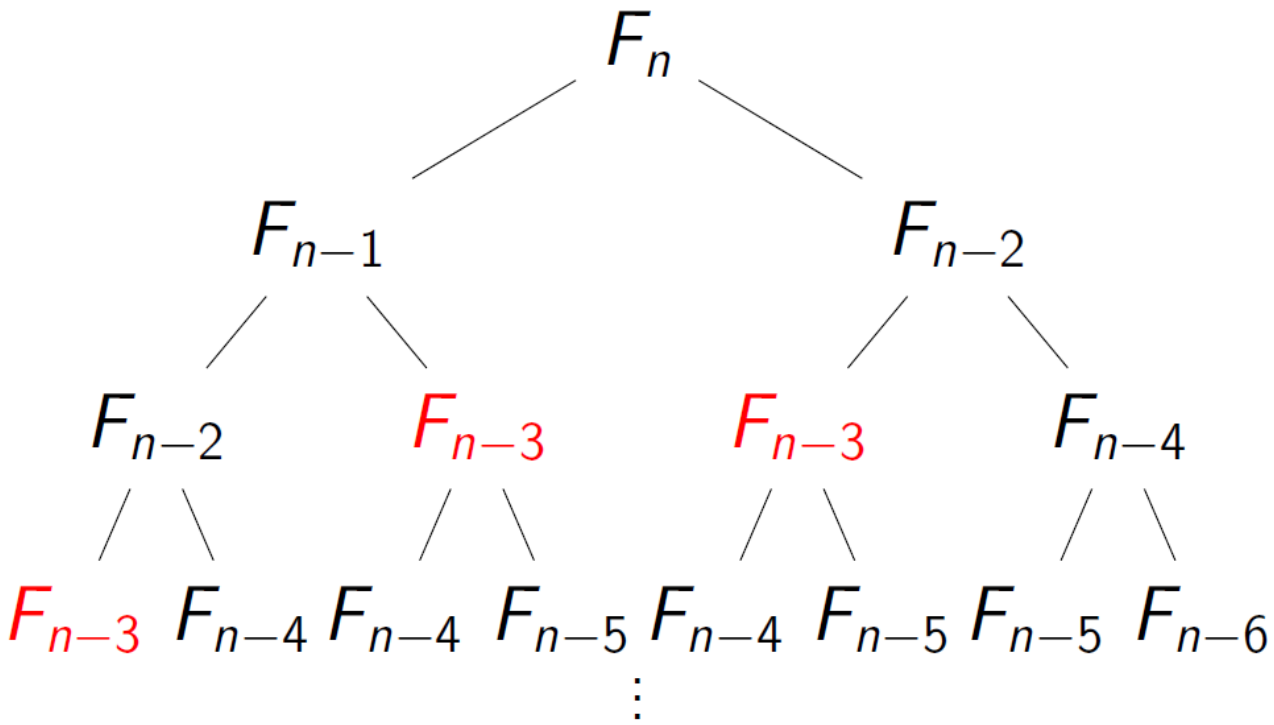
Therefore $T(n) \geq F_n$

$T(100) \approx 1.77 \cdot 10^{21}$ (1.77 sextillion)

Takes 56,000 years at 1GHz.

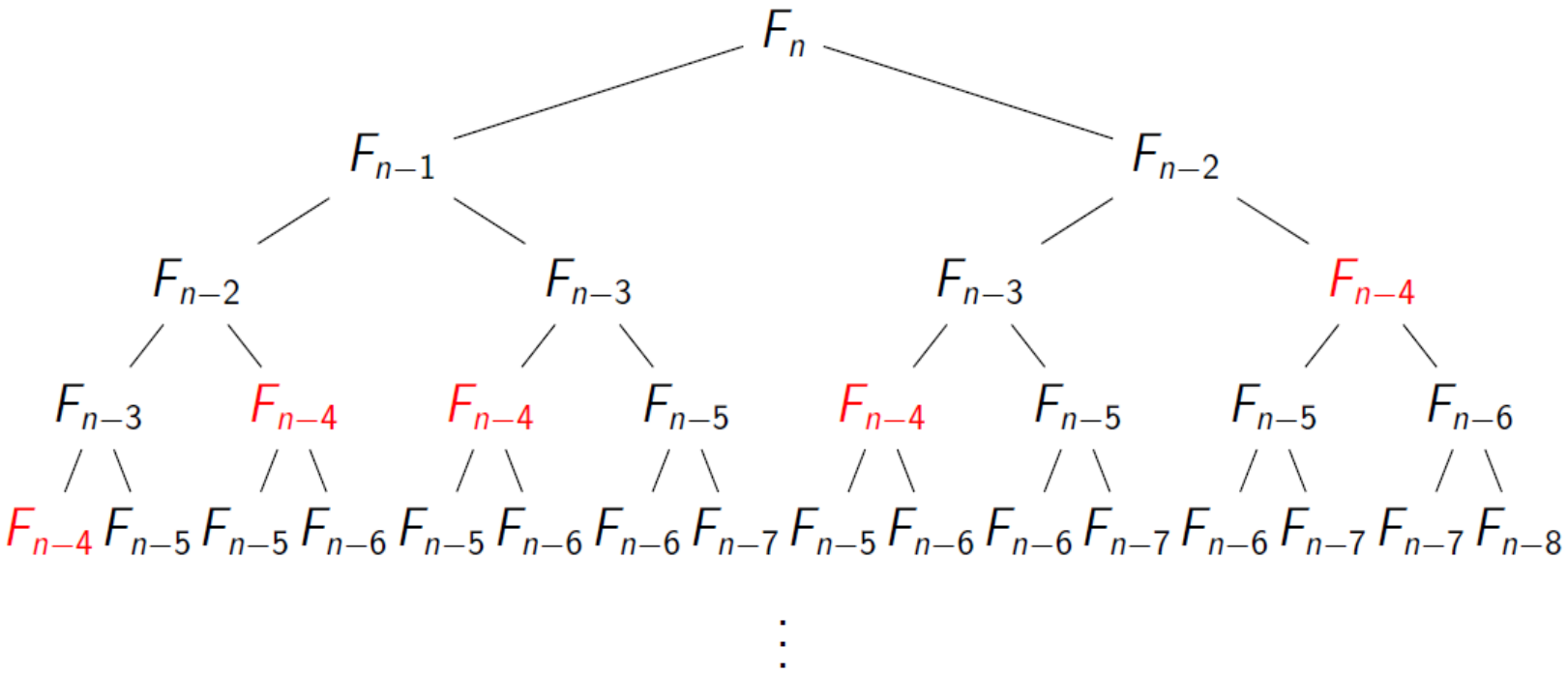
Naiwny algorytm: rekurencyjny

- Dlaczego tak czasochłonny?



Naiwny algorytm: rekurencyjny

- Dlaczego tak czasochłonny?



Efektywny algorytm: iteracja

- Spróbujmy ręcznie policzyć

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8$$

$$0+1=1$$
$$1+1=2$$
$$1+2=3$$
$$2+3=5$$
$$3+5=8$$

Definicja

$$F_n = \begin{cases} 0, & n=0, \\ 1, & n=1, \\ F_{n-1}+F_{n-2}, & n>1. \end{cases}$$

Efektywny algorytm: iteracja

- Funkcja: iteracyjne wypełnianie tablicy (pseudokod)

```
FibList(n)
create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
   F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

$$T(n) = 2n + 2. \text{ So } T(100) = 202.$$

Liczby Fibonaciego

- Naiwny algorytm (z definicji): prosty, elegancki, nieakceptowalnie wolny
- Iteracyjny algorytm: bardzo szybki

Zastosowanie ulepszonego (iteracyjnego) algorytmu umożliwia wykonanie obliczeń.

Przeszukiwanie: obrazu, tablicy

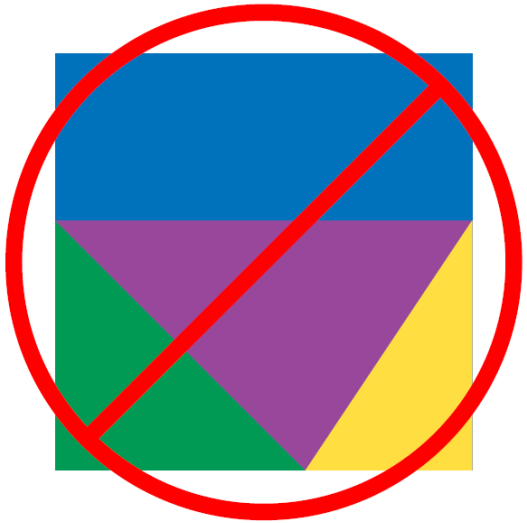
Metoda: „dziel i zwyciężaj”

- Podziel na problemy tego same typu, obszary nie mogą się pokrywać.

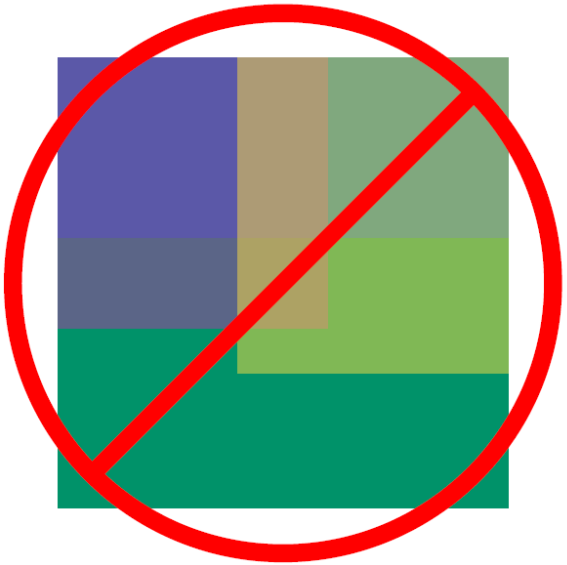
Metoda: „dziel i zwyciężaj”

- Podziel na problemy tego same typu, obszary nie mogą się pokrywać.

Nie są tego samego typu

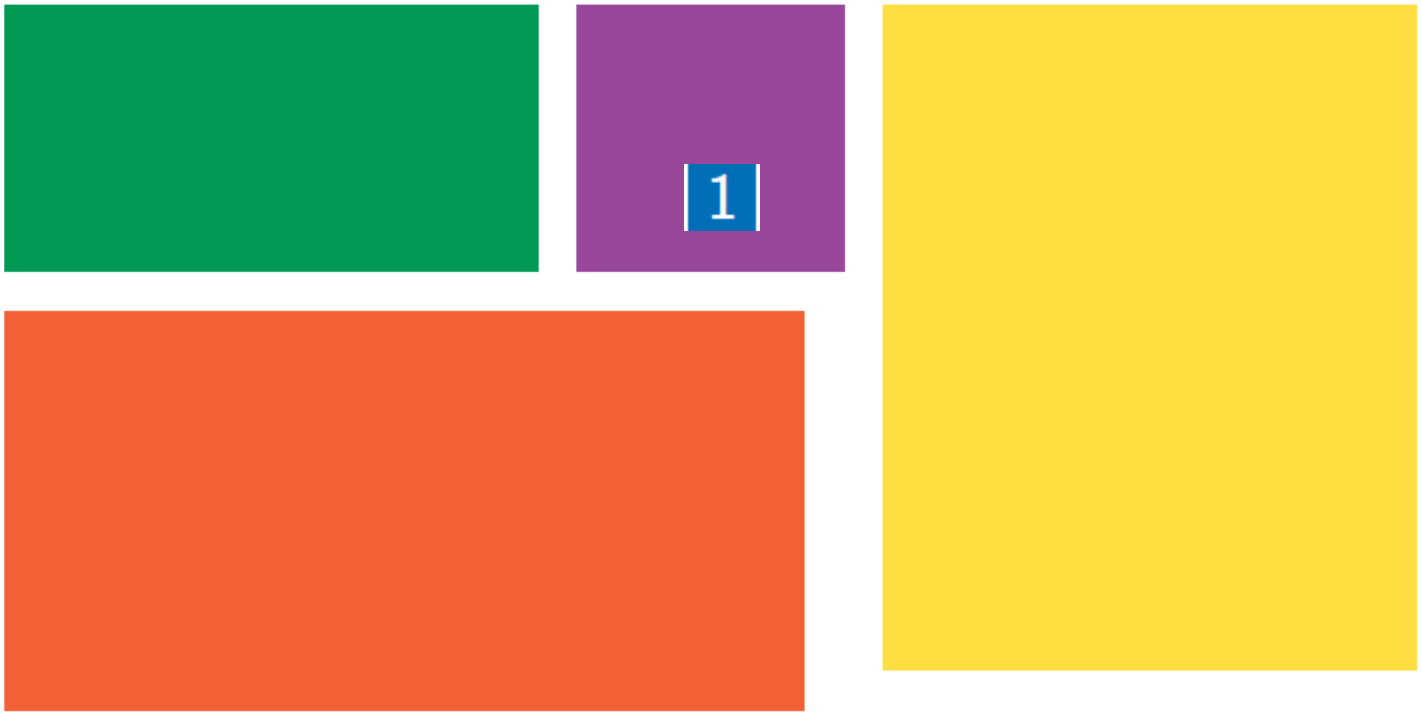


Nie są rozłączne



Metoda: „dziel i zwyciężaj”

1 Dziel na mniejsze problemy:



Metoda: „dziel i zwyciężaj”

2 Zwyciężaj: rozwiąż mniejsze problemy:

Metoda: „dziel i zwyciężaj”

3 Połącz z powrotem:

Przykład: liniowe przeszukiwanie tablicy

- Tłumaczenie słów: w danym wierszu słowa mają to samo znaczenie.

english	french	italian	german	spanish
house	maison	casa	Haus	casa
car	voiture	auto	Auto	auto
table	table	tavola	Tabelle	mesa

Przykład: liniowe przeszukiwanie tablicy

- Rozwiązanie rekurencyjne

LinearSearch(A, low, $high$, key)

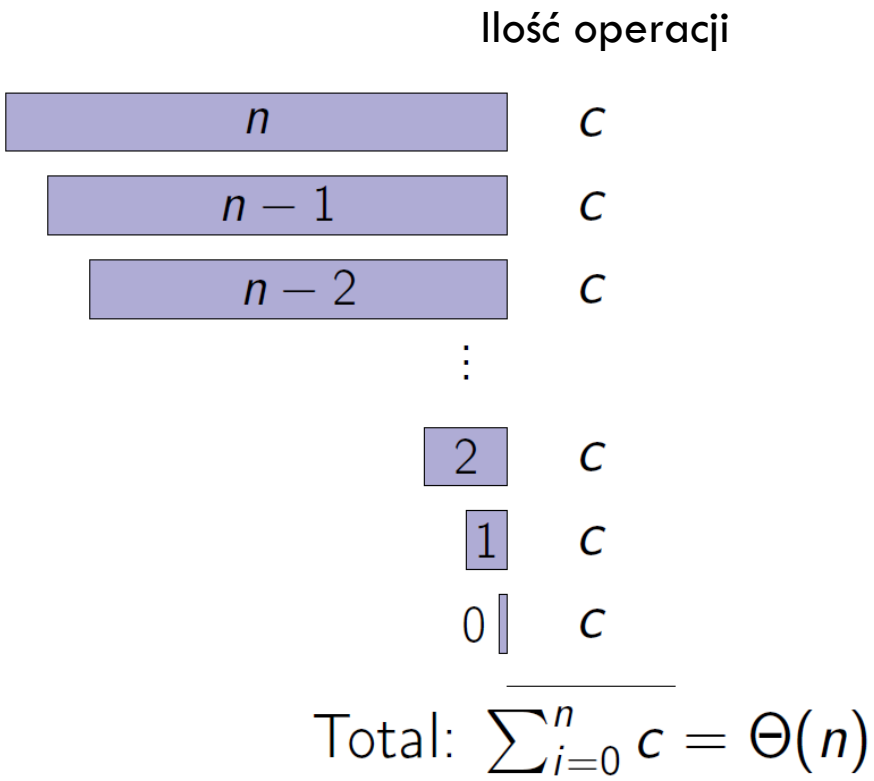
```
if high < low:
    return NOT_FOUND
if A[low] = key:
    return low
return LinearSearch(A, low + 1, high, key)
```

$$T(n) = T(n-1) + c$$

$$T(0) = c$$

Przykład: liniowe przeszukiwanie tablicy

- Rozwiązanie rekurencyjne



Przykład: liniowe przeszukiwanie tablicy

- Rozwiązanie iteracyjne

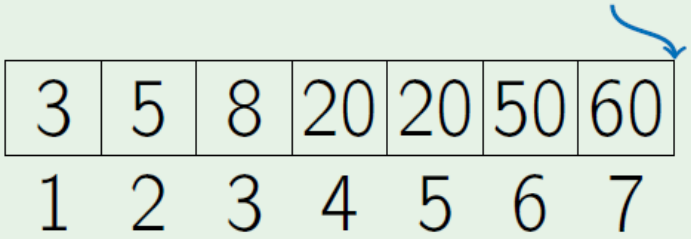
```
LinearSearchIt(A, low, high, key)

for i from low to high:
   if A[i] = key:
      return i
return NOT_FOUND
```

Przykład: posortowana tablica

- Przeszukiwanie posortowanej tablicy: ilość operacji

$search(2) \rightarrow 0$ $search(20) \rightarrow 4$
$search(3) \rightarrow 1$ $search(20) \rightarrow 5$
$search(4) \rightarrow 1$ $search(60) \rightarrow 7$
$search(90) \rightarrow 7$



Przykład: przeszukiwanie binarne

- Przeszukiwanie binarne posortowanej tablicy:

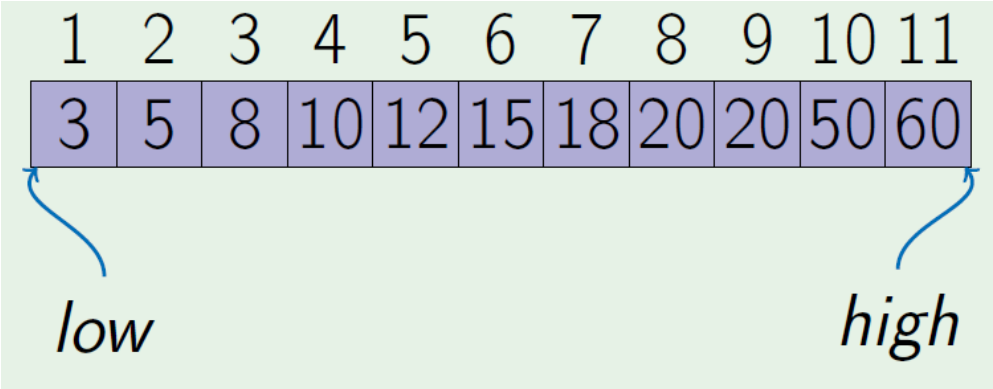
BinarySearch(A, low, $high$, key)

```
if high < low:
    return low − 1
mid ← ⌊low + (high−low)/2⌋
if key = A[mid]:
    return mid
else if key < A[mid]:
    return BinarySearch(A, low, mid − 1, key)
else:
    return BinarySearch(A, mid + 1, high, key)
```

Przykład: przeszukiwanie binarne

- Przeszukiwanie binarne: szukamy liczby „50”

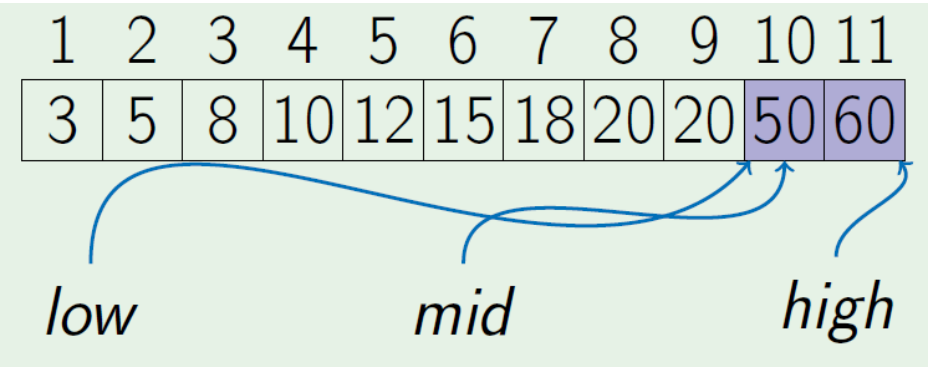
$$\texttt{BinarySearch}(A, 1, 11, 50)$$



Przykład: przeszukiwanie binarne

- Przeszukiwanie binarne: szukamy liczby „50”

$$\texttt{BinarySearch}(A, 1, 11, 50)$$
$$\texttt{BinarySearch}(A, 7, 11, 50)$$
$$\texttt{BinarySearch}(A, 10, 11, 50)$$



Przykład: przeszukiwanie binarne

- Przeszukiwanie binarne: szukamy liczby „50”

$$\texttt{BinarySearch}(A, 1, 11, 50)$$
$$\texttt{BinarySearch}(A, 7, 11, 50)$$
$$\texttt{BinarySearch}(A, 10, 11, 50) \rightarrow 10$$

1	2	3	4	5	6	7	8	9	10	11
3	5	8	10	12	15	18	20	20	50	60

Metoda: „dziel i zwyciężaj”

- Dzielimy rekurencyjnie na rozłączne (dla danego etapu podziału) mniejsze problemy tego samego typu.
- Rozwiązujemy mniejsze problemy
- Scalamy rozwiązania

Metoda: „dziel i zwyciężaj”

- Wersja rekurencyjna: przeszukiwanie binarne

BinarySearch$(A, low, high, key)$

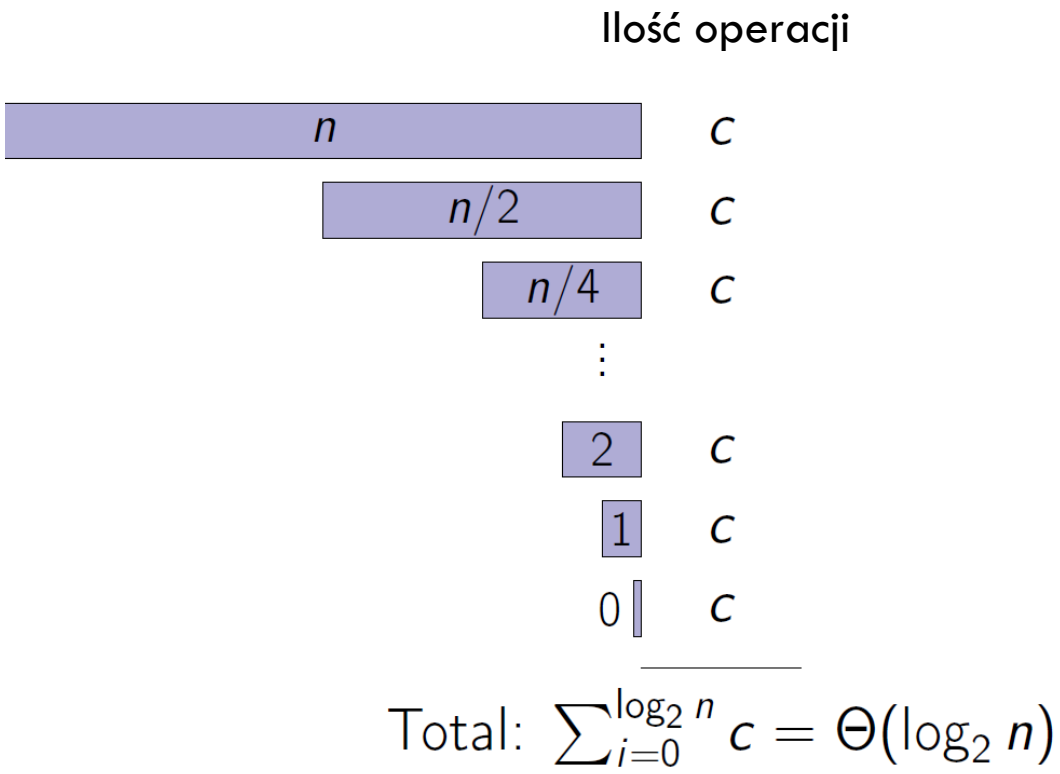
```
if high < low:
    return low − 1
mid ← ⌊low + (high−low)/2⌋
if key = A[mid]:
    return mid
else if key < A[mid]:
    return BinarySearch(A, low, mid − 1, key)
else:
    return BinarySearch(A, mid + 1, high, key)
```

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c$$

$$T(0) = c$$

Metoda: „dziel i zwyciężaj”

- Wersja rekurencyjna: złożoność obliczeniowa



$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c$$

$$T(0) = c$$

Metoda: „dziel i zwyciężaj”

- Wersja iteracyjna: przeszukiwanie binarne

BinarySearchIt($A, low, high, key$)

```
while low ≤ high:
    mid ← ⌊low + (high − low)/2⌋
    if key = A[mid]:
        return mid
    else if key < A[mid]:
        high = mid − 1
    else:
        low = mid + 1
```

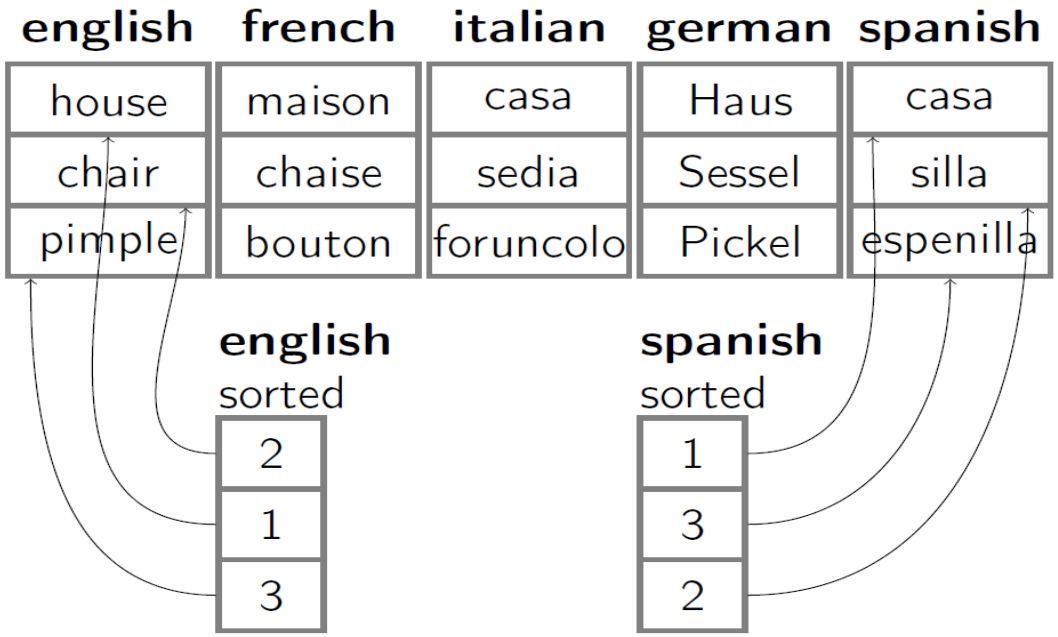
Metoda: „dziel i zwyciężaj”

- **Tłumaczenie słów: w każdej kolumnie posortowane wg. kolejności liter słowa w danym języku. W danym wierszu słowa nie maja tego samego znaczenia.**

english (sorted)	**french** (sorted)	**italian** (sorted)	**german** (sorted)	**spanish** (sorted)
chair	chaise	casa	Haus	casa
house	bouton	foruncolo	Pickel	espenilla
pimple	maison	sedia	Sessel	silla

Metoda: „dziel i zwyciężaj”

- **Tłumaczenie słów: w każdej kolumnie posortowane wg. kolejności liter słowa w danym języku. W danym wierszu słowa nie maja tego samego znaczenia.**
- **Zorganizuj jako przeszukiwanie binarne: pomocnicze tabele.**

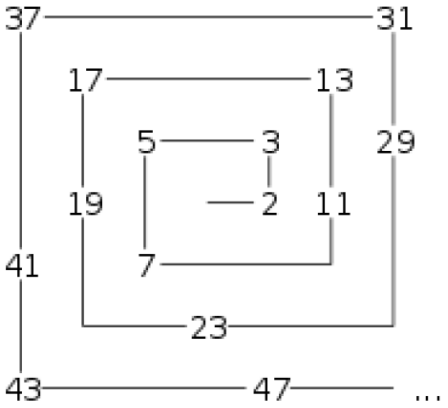


Największy wspólny podzielnik (NWP)

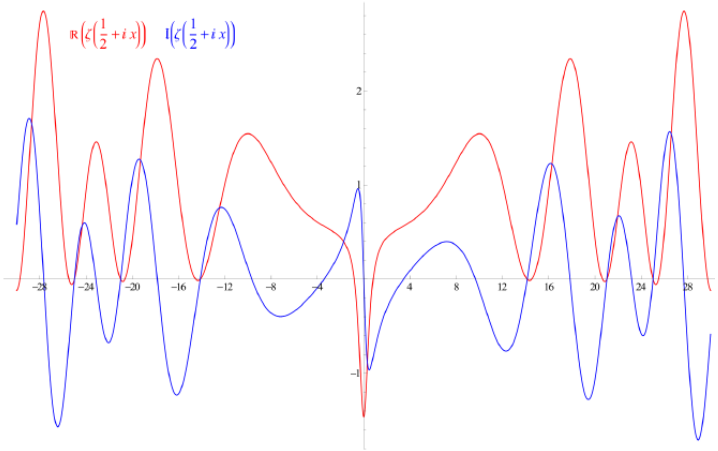
- **Definicja**

Dla pary liczb całkowitych a,b, ich największy wspólny podzielnik d = NWP(a,b) to największa liczba całkowita d taka ze dzieli bez reszty a, b.

Teoria liczb



Funkcje specjalne



Kryptografia

Największy wspólny podzielnik (NWP)

- **Definicja**

Dla pary liczb całkowitych a,b, ich największy wspólny podzielnik d = NWP(a,b) to największa liczba całkowita d taka ze dzieli bez reszty a, b.

Input: Integers $a, b \geq 0$

Output: $\gcd(a, b)$.

gcd – greatest common divisor

$$\gcd(3918848, 1653264)$$

Naiwny algorytm znajdowania NWP


```
Function NaiveGCD(a, b)
best ← 0
for d from 1 to a + b:
    if d|a and d|b:
        best ← d
return best
```

- Ilość operacji w przybliżeniu równa a+b
- Bardzo wolny już dla liczb 20-to cyfrowych

Algorytm Euklidesa

- **Lemat:**

  Jeżeli a' = reszta z dzielenia a/b
  to gcd(a,b)=gcd(a',b) = gcd(b,a')

- **Dowod (szkic)**
  - **a = a' + b*q**
  - **d dzieli a i b wtedy i tylko wtedy jeżeli d dzieli a' i b.**

Algorytm Euklidesa


```
Function EuclidGCD(a, b)
if b = 0:
   return a
a' ← the remainder when a is
    divided by b
return EuclidGCD(b, a')
```

Funkcja rekurencyjna, wprost zastosowanie lematu.

Algorytm Euklidesa


```
Function EuclidGCD(a, b)
if b = 0:
  return a
a' ← the remainder when a is
   divided by b
return EuclidGCD(b, a')
```

Ilość operacji: każdy krok redukuje liczby o czynnik 2. Ilość kroków to log(a*b). Jeżeli liczba 100-cyfrowa, potrzebne około 600 kroków. Każdy krok to pojedyncze dzielenie.

Przykład

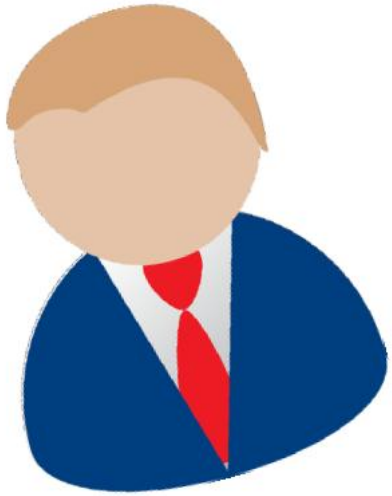
$$\begin{aligned}
&\gcd(3918848, 1653264)\\
=&\gcd(1653264, 612320)\\
=&\gcd(612320, 428624)\\
=&\gcd(428624, 183696)\\
=&\gcd(183696, 61232)\\
=&\gcd(61232, 0)\\
=&61232.
\end{aligned}$$

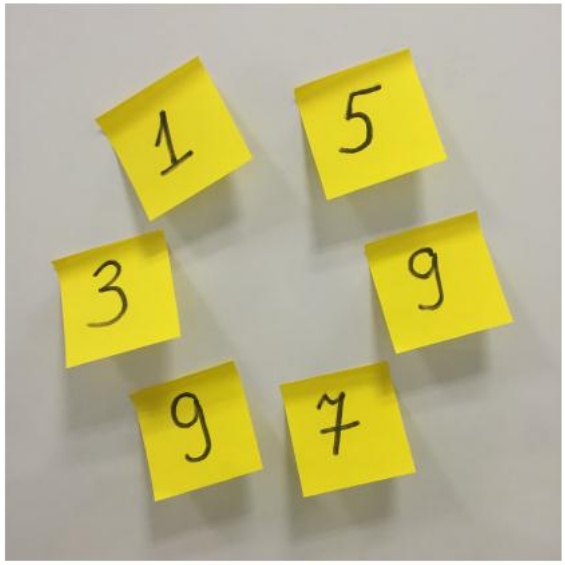
Efektywny algorytm

- Naiwny algorytm jest zbyt wolny.
- Algorytm Euklidesa dużo efektywniejszy.
- Wymyślenie efektywnego algorytmu wymagało wiedzy na temat problemu, w tym przypadku z dziedziny teorii liczb..

Algorytm zachłanny

- Jaka jest największa liczba która możesz zbudować maja do dyspozycji podane cyfry.

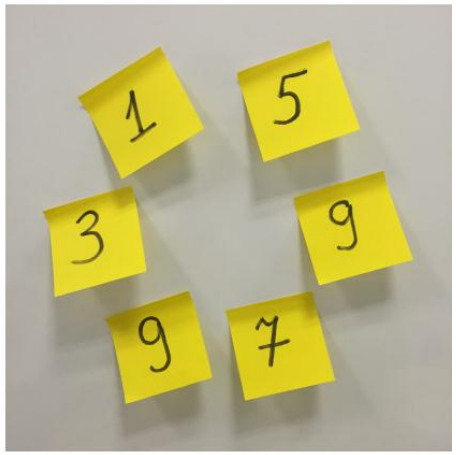




????

Algorytm zachłanny

- Jaka jest największa liczba która możesz zbudować używając wszystkie podane cyfry.

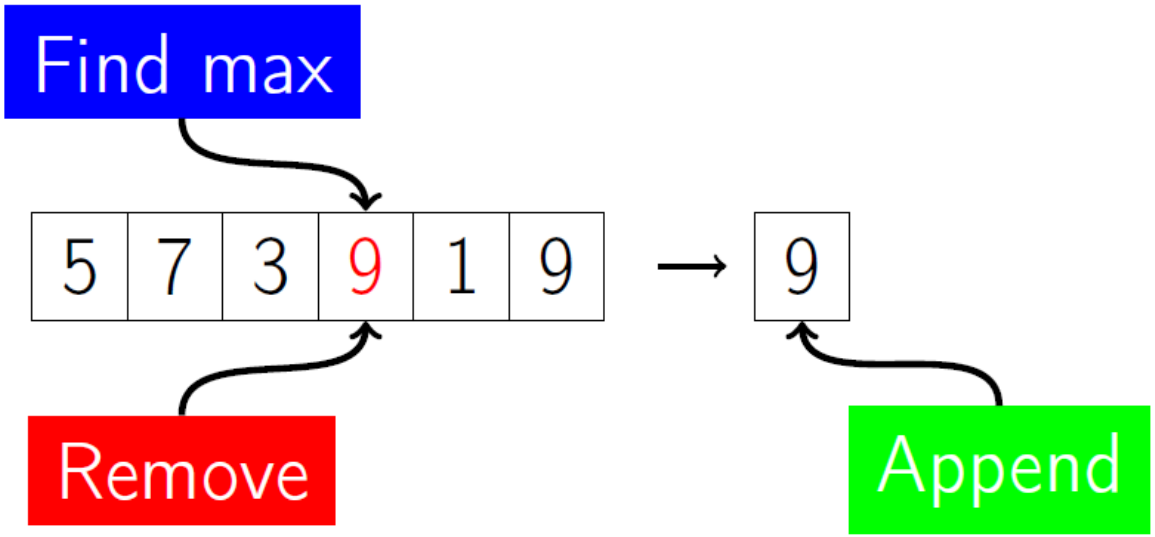


359179, 537991, 913579, . . .

Poprawna odpowiedz

997531

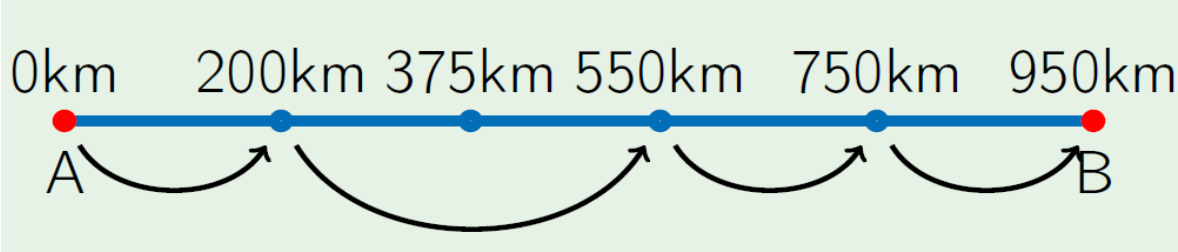
Strategia zachłanna: największa liczba

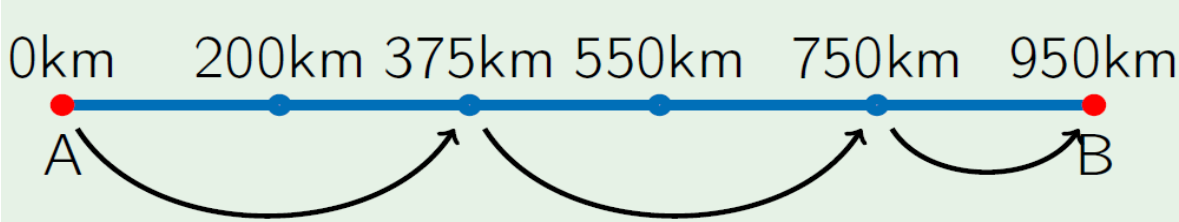


- Znajdź największą cyfrę
- Dołącz do liczby
- Usuń cyfrę z listy cyfr
- Powtarzaj dopóki lista cyfr nie jest pusta

Strategia zachłanna: tankowanie benzyny

Pełny bak benzyny wystarcz na przejechanie 400km. Na których stancjach tankować paliwo aby ilość tankowan potrzebna do przejechania 950 km była minimalna.





Tylko dwa tankowania

Wybór strategii

- **Jakie są możliwości**
  - **Zatankuj na każdej stacji po drodze**
  - **Zatankuj na najdalszej stacji do której możesz dojechać**
  - **Jedz aż do pustego baku.**
- **Strategia zachłanna**
  1) **Wystartuj w punkcie A**
  2) **Dojedz do najdalszej możliwie stacji benzynowej**
  3) **Zatankuj i potraktuj ten punkt jako nowy punkt A**

  Powtarzaj (1-3) dopóki nie zajdzie A=B

Podproblem

- Podproblem jest taki sam jak oryginalny problem.

  - $\texttt{LargestNumber}(3, 9, 5, 9, 7, 1) =$ ‘‘9’’ $+ \texttt{LargestNumber}(3, 5, 9, 7, 1)$
  - Min number of refills from A to $B =$ first refill at G + min number of refills from G to B

Algorytm zachłanny: liniowa zależność od ilości stacji benzynowych.

$$A = x_0 \leq x_1 \leq x_2 \leq \cdots \leq x_n \leq x_{n+1} = B$$

MinRefills(x, n, L)

```
numRefills ← 0, currentRefill ← 0
while currentRefill ≤ n:
    lastRefill ← currentRefill
    while (currentRefill ≤ n and
           x[currentRefill + 1] − x[lastRefill] ≤ L):
        currentRefill ← currentRefill + 1
    if currentRefill == lastRefill:
        return IMPOSSIBLE
    if currentRefill ≤ n:
        numRefills ← numRefills + 1
return numRefills
```

Jak podzielić na grupy

- Chcemy podzielić na grupy gdzie wiek dziecka nie różni się więcej niż o 1 rok i tak aby było jak najmniej grup.

Jak efektywnie podzielić na grupy

- Naiwny algorytm

```
MinGroups(C)

m ← len(C)
for each partition into groups
C = G_1 ∪ G_2 ∪ ··· ∪ G_k:
  good ← true
  for i from 1 to k:
    if max(G_i) − min(G_i) > 1:
      good ← false
  if good:
    m ← min(m, k)
return m
```

Ilość operacji:
co najmniej 2^n
dla n elementów które należy pogrupować.

Jak efektywnie podzielić na grupy

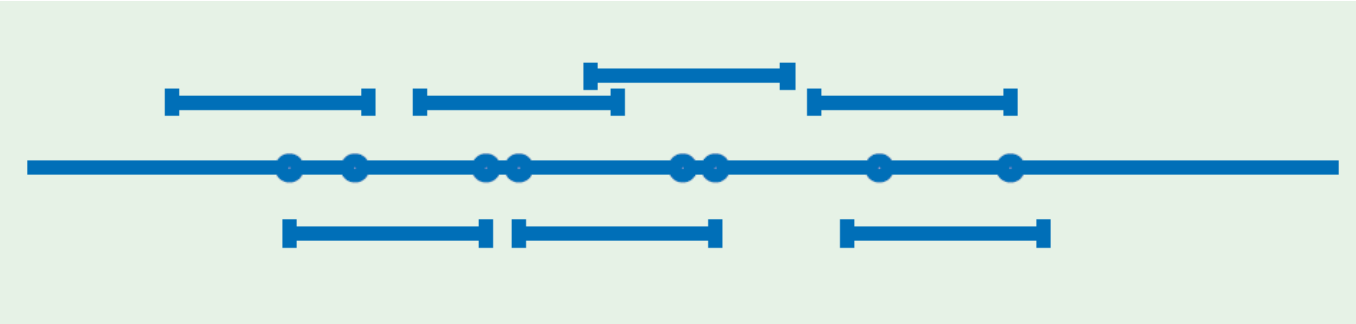
- **Naiwny algorytm**
  - **Rozważmy podział na 2 grupy: C = G1 + G2**
  - **Każdy element i może być zaakceptowany do grupy G1 albo odrzucony i wtedy jest w grupie G2.**
  - **W ten sposób można utworzyć 2^n różnych grup G1**
  - **Ilość operacji co najmniej 2^n**

$$n = 50$$

$$2^{50} = 1125899906842624$$

Jak efektywnie podzielić na grupy

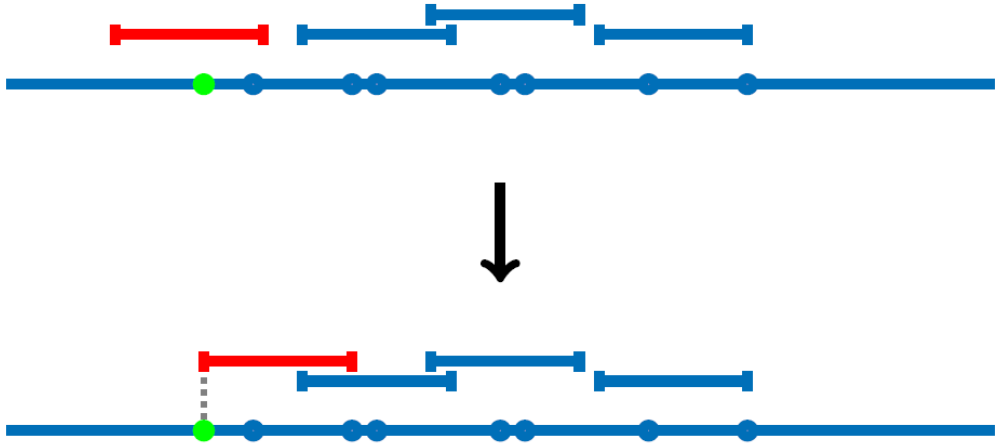
- **Posortujmy najpierw elementy (punkty): n log(n)**
- **Zdefiniujmy problem jako poszukiwanie najmniejszej ilości odcinków które pokryją wszystkie punkty. Długość odcinka nie większa niż 1 jednostka.**



- **Jak to zrobić: zacznij od najbardziej lewego punktu i wyznacz odcinek o długości co najwyżej 1. Punkty leżące na odcinku tworzą jedna grupę.**

Jak efektywnie podzielić na grupy

- Posortujmy najpierw elementy (punkty): n log(n)
- Zdefiniujmy problem jako poszukiwanie najmniejszej ilości odcinków które pokryją wszystkie punkty. Długość odcinka nie większa niż 1 jednostka.



Jak efektywnie podzielić na grupy

$x_1 \leq x_2 \leq \ldots \leq x_n$

PointsCoverSorted$(x_1, \ldots, x_n)$

$R \leftarrow \{\}, \; i \leftarrow 1$
while $i \leq n$:
  $[\ell, r] \leftarrow [x_i, x_i + 1]$
  $R \leftarrow R \bigcup \{[\ell, r]\}$
  $i \leftarrow i + 1$
  while $i \leq n$ and $x_i \leq r$:
    $i \leftarrow i + 1$
return R

Sortowanie: Ilość operacji proporcjonalna do n log(n)

Ilość operacji proporcjonalna do n

Jak efektywnie podzielić na grupy

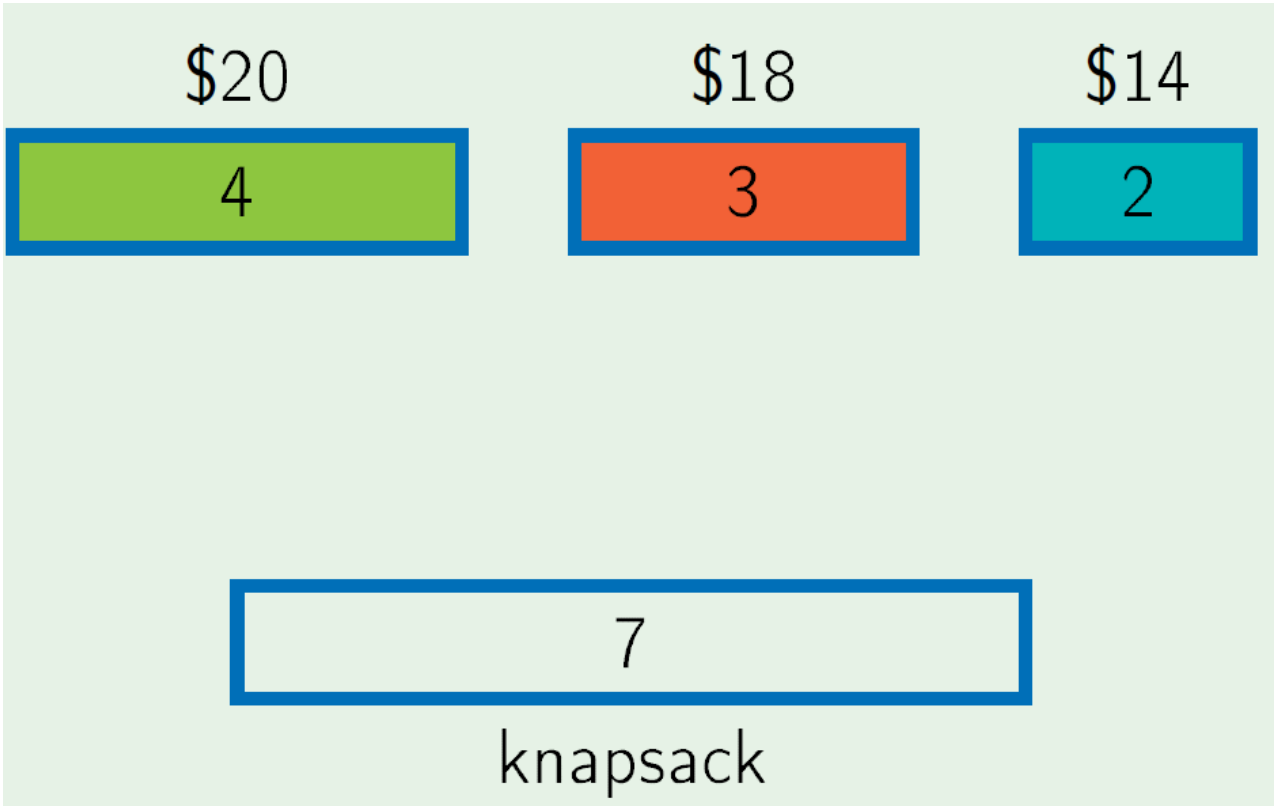
- **Naiwny algorytm: ilość operacji ~ 2^n**
  - **Bardzo wolny już dla n=50**
- **Sortowanie + algorytm zachłanny**
  - **Sortowanie ~ n log()**
  - **Algorytm zachłanny ~ n**
  - **Całość efektywna nawet dla n=10 000 000**

Algorytm zachłanny: pakowanie plecaka

- Jak najlepiej zapakować plecak

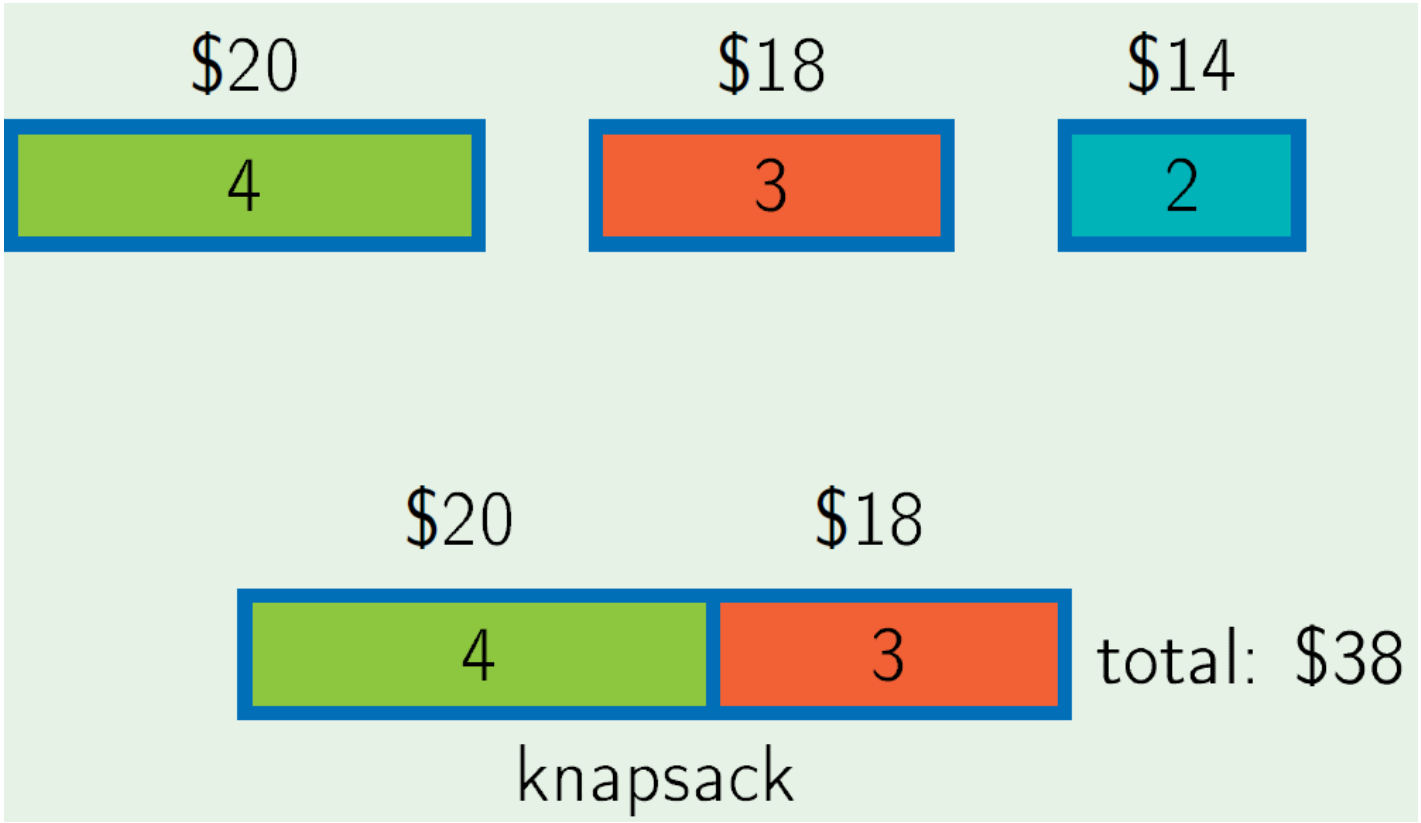
Algorytm zachłanny: pakowanie plecaka

- Jak najlepiej zapakować plecak



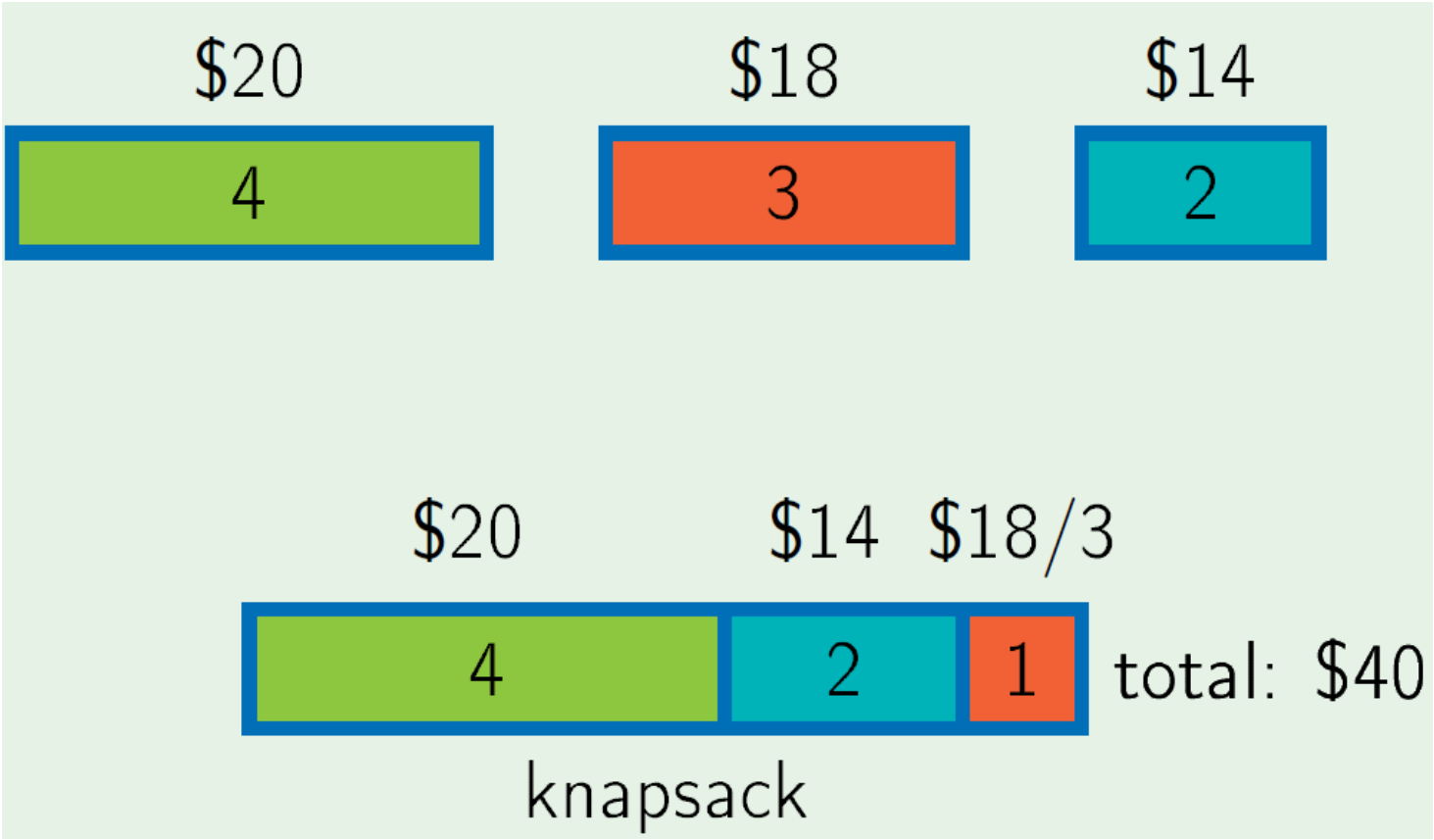
Algorytm zachłanny: pakowanie plecaka

- Jak najlepiej zapakować plecak



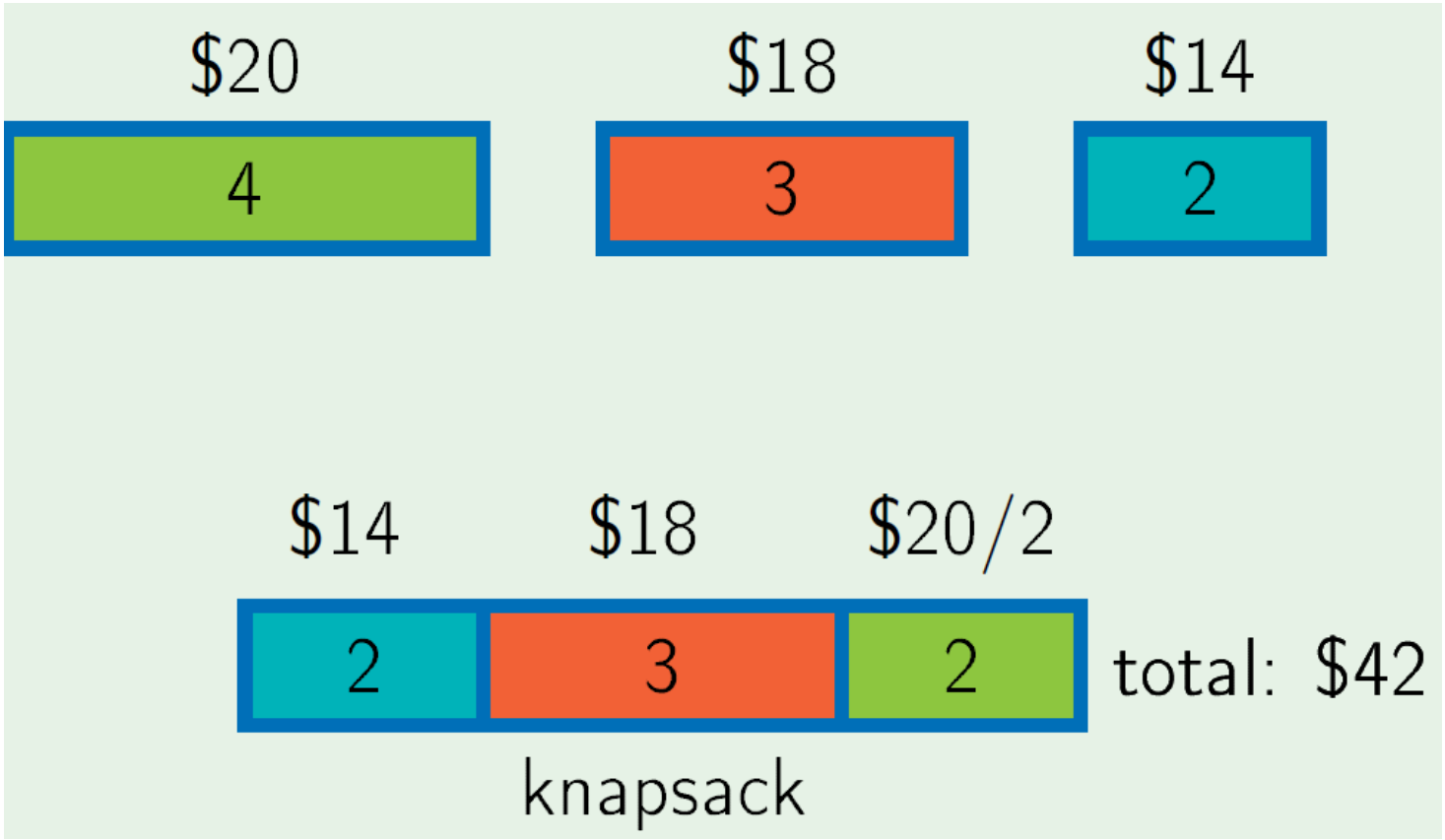
Algorytm zachłanny: pakowanie plecaka

- Jak najlepiej zapakować plecak



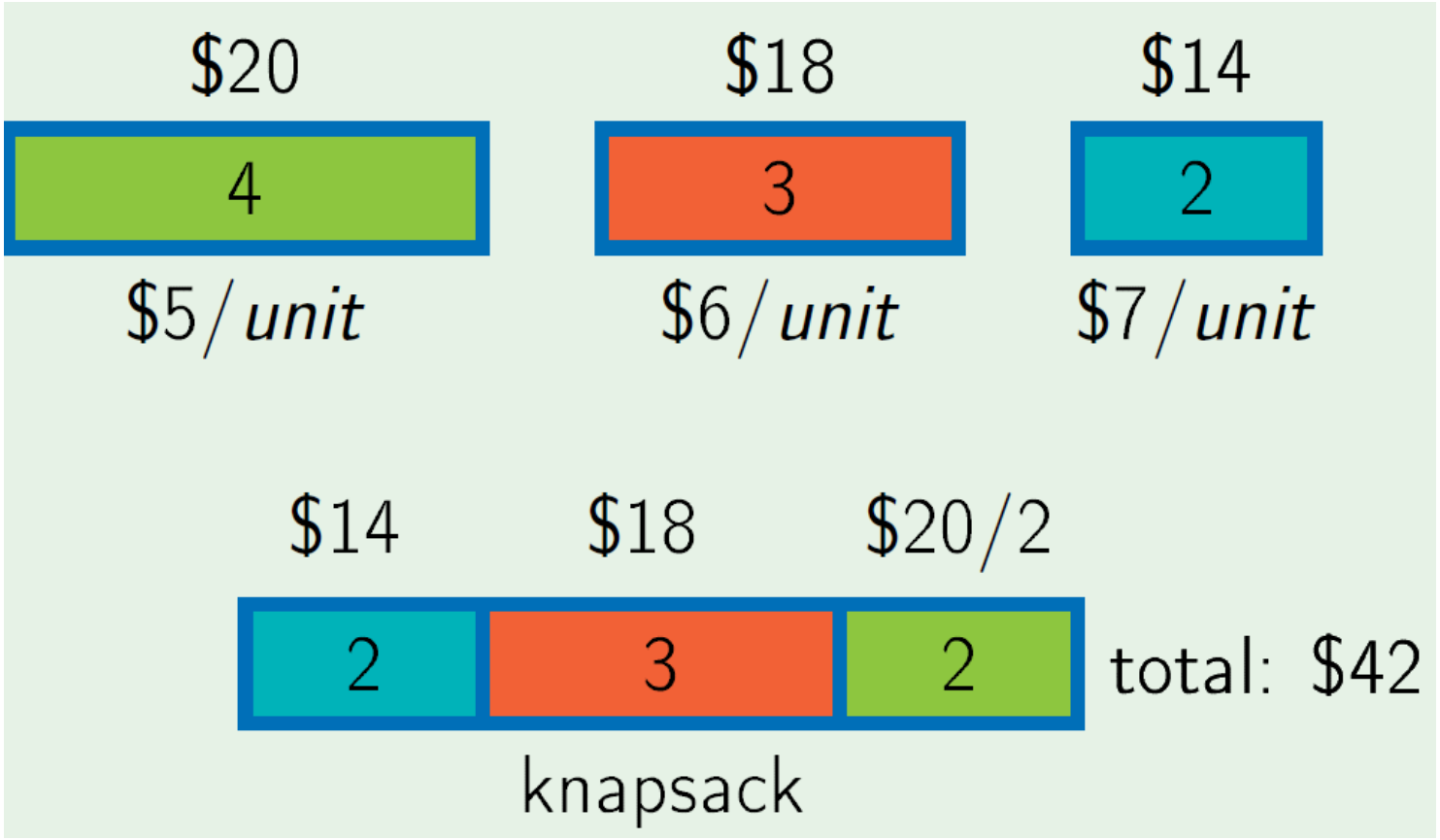
Algorytm zachłanny: pakowanie plecaka

- Jak najlepiej zapakować plecak



Algorytm zachłanny: pakowanie plecaka

- Jak najlepiej zapakować plecak



Algorytm zachłanny: pakowanie plecaka

- **Jak najlepiej zapakować plecak**
  1) **Dopóki plecak nie jest pełny, wybierz grupę i o największym vi/wi.**
  2) **Jeżeli przedmioty mieszczą się w plecaku umieść wszystkie. Jeżeli nie ta tyle aby zapełnić plecak.**
  3) **Jeżeli jeszcze jest miejsce w plecaku, wybierz nowe i na kolejne o największym vi/wi.**

Powtarzaj (1)-(3) do momentu zapełnienia plecaka lub aż nie będzie więcej przedmiotów które się w nim mieszczą.

Algorytm zachłanny: pakowanie plecaka

Knapsack$(W, w_1, v_1, \ldots, w_n, v_n)$

$A \leftarrow [0, 0, \ldots, 0], V \leftarrow 0$
repeat n times:
  if $W = 0$:
    return (V, A)
  select i with $w_i > 0$ and max $\frac{v_i}{w_i}$
  $a \leftarrow \min(w_i, W)$
  $V \leftarrow V + a\frac{v_i}{w_i}$
  $w_i \leftarrow w_i - a, A[i] \leftarrow A[i] + a, W \leftarrow W - a$
return (V, A)

Algorytm zachłanny: pakowanie plecaka

- Czy można efektywniej?
  - **Ilość operacji które były wykonywane: n^2**
    - **Wybieranie kolejnego i – maksymalnie n-razy**
    - **Wkładanie przedmiotów – maksymalnie n-razy**
  - **Gdybyśmy najpierw posortowali wg. malejącego vi/wi to ilość operacji bylaby: n log(n) + n + n**
    - **Sortowanie: n log(n) operacji**
    - **Wybranie kolejnego i – 1operacja**
    - **Wkładanie do plecaka – n operacji**

Algorytm zachłanny: pakowanie plecaka

- Posortowaliśmy: $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$

Knapsack$(W, w_1, v_1, \ldots, w_n, v_n)$

$A \leftarrow [0, 0, \ldots, 0], V \leftarrow 0$
for i from 1 to n:
  if $W = 0$:
    return (V, A)
  $a \leftarrow \min(w_i, W)$
  $V \leftarrow V + a\frac{v_i}{w_i}$
  $w_i \leftarrow w_i - a, A[i] \leftarrow A[i] + a, W \leftarrow W - a$
return (V, A)