Akademia Górniczo-Hutnicza
Wydział Elektrotechniki, Automatyki,
Informatyki i Inżynierii Biomedycznej

Adrian Horzyk

WSTĘP DO INFORMATYKI

WPROWADZENIE DO ALGORYTMIKI

Programista vs. Informatyk

Czym różni się informatyk od programisty?

- **Programista** zajmuje się wyszukiwaniem odpowiednich algorytmów oraz ich zestawianiem i translacją na postać programów komputerowych w konkretnym języku programowania, starając się ich działanie możliwie zoptymalizować na poziomie języka programowania poprzez dobór odpowiednich metod i ew. również struktur danych.
- **Informatyk** powinien być programistą, który potrafi na bazie zdobytej wiedzy umieć rozwiązywać różne problemy poprzez tworzenie nowych algorytmów, które mogą być podobne do istniejących lub wykorzystywać istniejące algorytmy i struktury danych, lecz w nowy innowacyjny sposób.
  Ponadto informatyk powinien umieć dostosować i zoptymalizować istniejące algorytmy do rozwiązywanego zadania poprzez uproszczenie, rozbudowanie lub modyfikację zwykle stosowanych struktur danych oraz uproszczenie lub modyfikacje pewnych części algorytmu dopasowując go do wymogów rozwiązywanego zadania.

ALGORYTMIKA

Algorytmika – to dział informatyki zajmujący się teorią związaną z tworzeniem i badaniem algorytmów, poprawnością, złożonością oraz algorytmizowaniem procesów i zjawisk zachodzących w otaczającym nas świecie.

Tworzenie algorytmu to proces modelowania pewnego zjawiska lub procesu dla pewnego rodzaju danych wejściowych, które mają zostać przetworzone w celu osiągnięcia pewnego wyniku w skończonej ilości kroków.

Algorytmizacja jest więc procesem budowy algorytmu dla konkretnego zjawiska lub procesu, który chcemy zamodelować i zautomatyzować, celem jego implementacji i wykonania na maszynie obliczeniowej.

Jednym z podstawowych celów algorytmiki jest analiza algorytmów pod względem efektywności, dokładności i poprawności ich działania oraz poszukiwanie algorytmów o większej efektywności i dokładności.

Algorytmika bada również kwestie złożoności czasowej i pamięciowej algorytmów w celu weryfikacji ich możliwości wykonania na dostępnych zasobach i w pożądanym czasie.

Część problemów wymaga rozwiązania w czasie rzeczywistym (*real time*) .

SCHEMAT BLOKOWY

Schemat blokowy jest jednym ze sposobów zapisu algorytmu prezentujący kolejne kroki (instrukcje), jakie trzeba wykonać w celu osiągnięcia postawionego celu.

Schemat blokowy wykorzystuje pewien zbiór figur geometrycznych reprezentujących pewne kategorie operacji na danych oraz połączenia, które wskazują kierunek ich przetwarzania i możliwie alternatywne przejścia:

- **Blok graniczny** wskazujący początek, koniec, przerwanie lub wstrzymanie wykonywania algorytmu.
- **Blok wejścia-wyjścia** związany jest z wprowadzanie danych do przetworzenia oraz wyprowadzaniem wyników.
- **Blok operacyjny** służy do wykonywania operacji na danych przechowywanych w postaci zmiennych i stałych różnego typu.
- **Blok decyzyjny (warunkowy)** umożliwia dokonanie wyboru na podstawie wyniku operacji logicznej zapisanej w postaci pewnego warunku.
- **Blok podprogramu** umożliwia przejście do wydzielonego fragmentu algorytmu i wywołania go dla pewnych parametrów, ew. zwrot obliczonych wartości.
- **Blok fragmentu** przedstawiający wyodrębniony fragment kodu.
- **Blok komentarza** pozwala wstawić objaśnienia ułatwiające zrozumienie algorytmu.
- **Łącznik wewnętrzny** służy do łączenia odrębnych części schematu na tej samej stronie opatrzony etykietą przeniesienia.
- **Łącznik zewnętrzny** łączy schematy na różnych stronach, więc powinien zawierać numer strony i etykietę przeniesienia.

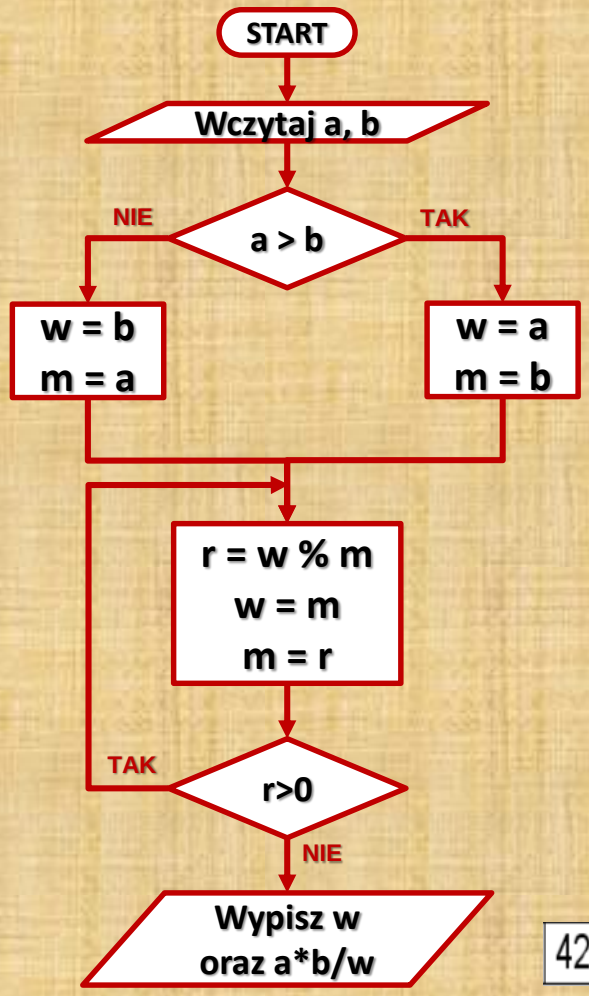
SCHEMAT BLOKOWY

Algorytm możemy opisać słownie, przy pomocy schematu blokowego lub kodu w języku programowania, np. wyznaczający NWD i NWW algorytmem Euklidesa:

Opis słowny (pseudokod)

1. Wczytaj dwie liczby naturalne a i b.
2. Jeśli a > b,
   to przyporządkuj
   w = a oraz m = b,
   a w przeciwnym przypadku
   przyporządkuj
   w = b oraz m = a.
3. Następnie dopóki r jest większe od zera wyznacz resztę z dzielenia w przez m i zapisz ją w zmiennej r oraz dokonaj przyporządkowania w = m, m = r.
4. Wypisz wartość w NWD oraz a*b/w jako NWW

Schemat blokowy



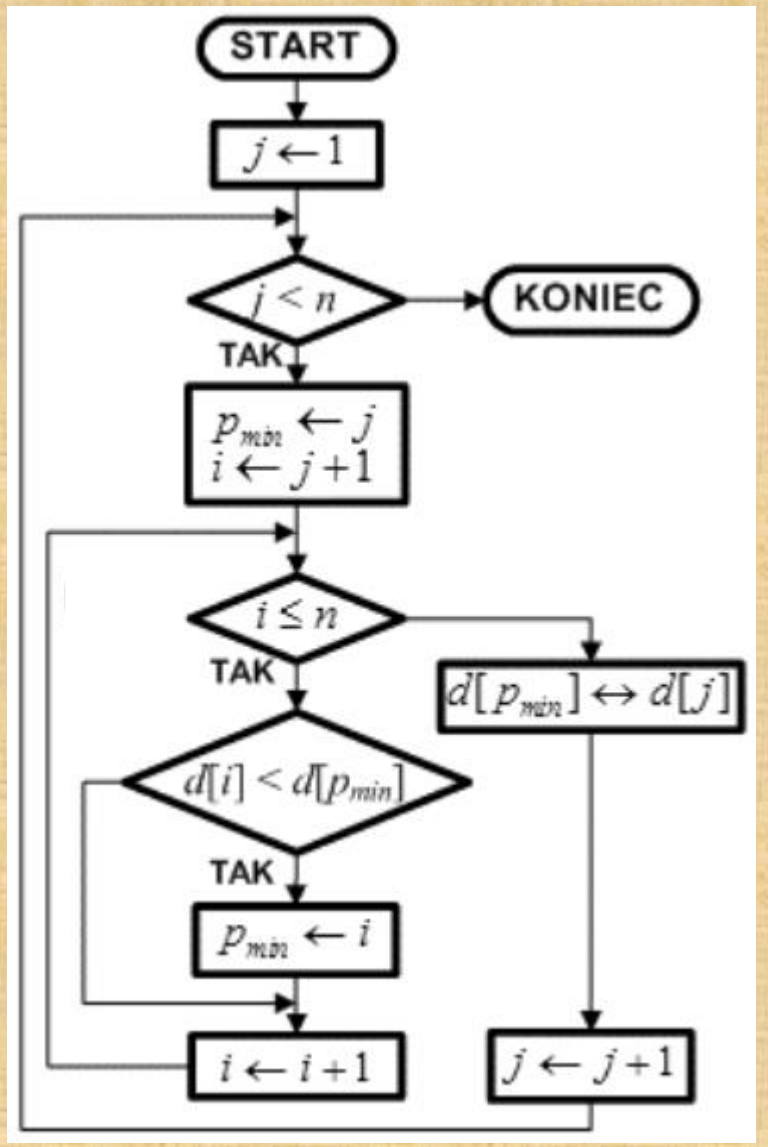
Program w Pythonie

```
print "Podaj dwie liczby naturalne:”
a = input("Pierwsza liczba:")
b = input("Druga liczba:")
if a > b:
    w = a; m = b
else:
    w = b; m = a
while r:
    r = w % m
    w = m
    m = r
print "NWD liczb %i i %i wynosi %i, a ich NWW wynosi %i" % (a, b, w, a*b/w)
```

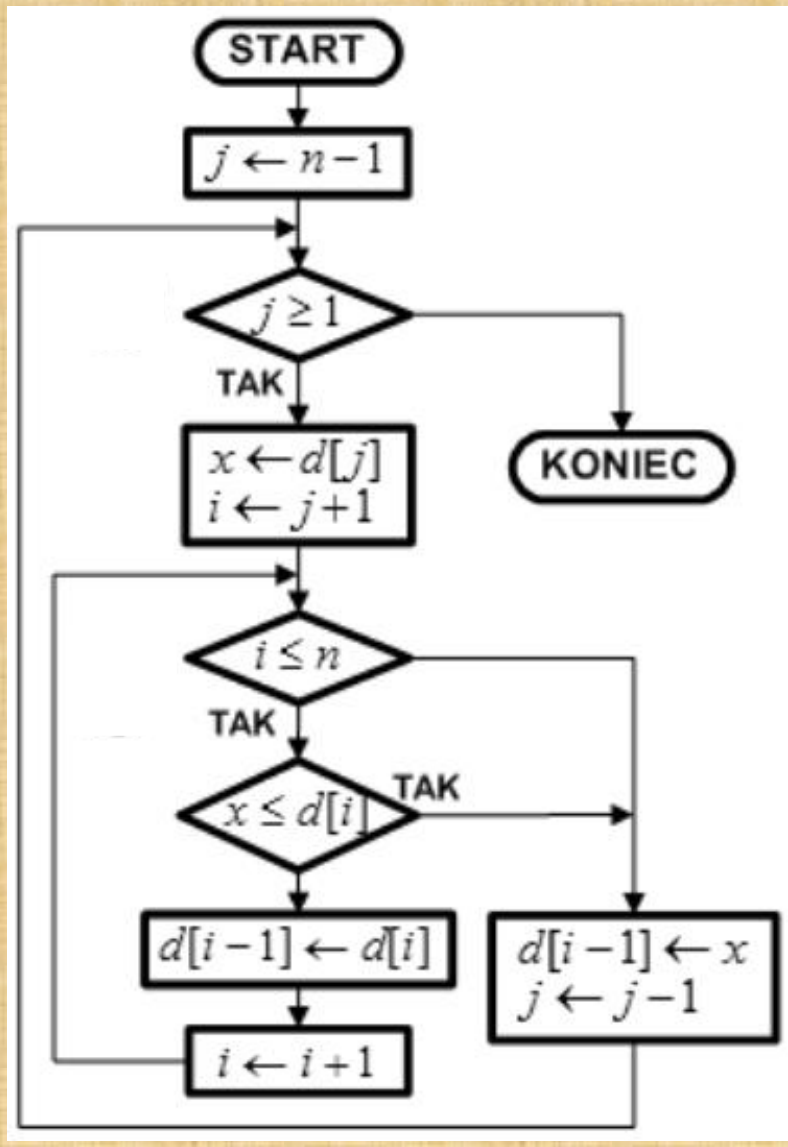
4200	3324	876	696	180	156	24	12	0

SCHEMAT BLOKOWY – PRZYKŁADY

SORTOWANIE PRZEZ WYBIERANIE



SORTOWANIE PRZEZ WSTAWIANIE



STRATEGIA: DZIEL I ZWYCIĘŻAJ

Strategia „dziel i zwyciężaj” (*a divide and conquer strategy*) polega na rekurencyjnym dzieleniu zadania na mniejsze i stosowaniu tej strategii do zadań składowych tak długo, dopóki nie uzyska się rozwiązania całego zadania.

Następnie (w zależności od postawionego zadania) rozwiązanie końcowe składa się z rozwiązań cząstkowych uzyskanych dla podzielonych podzadań, jedynie że rozwiązanie cząstkowe jest zarazem rozwiązaniem końcowym.

Dzięki takiej strategii, ilość porównań (lub innych operacji) znacznie maleje (zwykle logarytmicznie).

Strategię tą stosujemy w wielu algorytmach, np. w:

- **wyszukiwanie połówkowe (*binary search*),**
- **wyszukiwanie interpolowane (*interpolation search*),**
- **sortowaniu szybkim (*quick sort*),**
- **sortowaniu przez scalanie (*merge sort*)**

w celu osiągnięcia wysokiej efektywności ich działania.

WYSZUKIWANIE

Wyszukiwanie – to najczęstsze operacje wykonywane w informatyce!

Jego efektywna implementacja decyduje zwykle o szybkości całej aplikacji.

Nieuporządkowane struktury sekwencyjne musimy przeszukiwać sekwencyjnie element po elemencie, czyli tzw. wyszukiwanie liniowe (*sequential search*), co w pesymistycznym przypadku zmusza nas do przeglądnięcia wszystkich N elementów, więc kosztuje N porównań!

PRZYKŁAD: N = 1.000.000.000 → Max ilość porównań: 1.000.000.000

W przypadku przeszukiwania uporządkowanej sekwencyjnej struktury danych, możemy zastosować algorytm wyszukiwania połówkowego (*binary search*), który wymaga maksymalnie $\log_2 N$ operacji porównywania.

PRZYKŁAD: N = 1.000.000.000 → Max ilość porównań: 30

W przypadku równomiernego rozkładu liczb w uporządkowanej sekwencji możemy zastosować algorytm wyszukiwania interpolowanego (*interpolation search*), który próbuje „odgadnąć" pozycję (obliczyć indeks) poszukiwanej wartości, co wiąże się z ilością operacji porównywania równą $\log_2 \log_2 N$

PRZYKŁAD: N = 1.000.000.000 → Przewidywana ilość porównań ok. 5

Istnieje jeszcze możliwość wykorzystania tablicy haszującej do bardzo szybkiego wyszukiwania elementów w czasie stałym, o ile jesteśmy w stanie określić funkcję haszującą dla przeszukiwanej sekwencji danych, od których zwykle również wymaga się pewnego równomiernego rozkładu. Funkcja haszująca określa miejsce w tablicy wskaźnika krótkiej listy elementów, w której znajduje się poszukiwany element, stąd tak ważny jest odpowiedni rozkład danych oraz możliwość szybkiego O(1) przejrzenia takiej listy. Zwykle i tak wykonujemy pewną stałą ilość porównań, więc ich ilość jest zwykle porównywalna z wyszukiwaniem interpolowanym.

WYSZUKIWANIE SEKWENCYJNE (LINIOWE)

Przeszukiwanie sekwencyjne – stosowane jest do nieuporządkowanych liniowych struktur danych, przeglądając poszczególne elementy jeden po drugim aż do napotkania poszukiwanego elementu:

Pierwszy algorytm przeszukuje listę od tyłu w celu eliminacji obliczania wartości indeksu w przypadku nieodnalezienia elementu.

```
5  @author: Adrian Horzyk
6  """
7
8  def SequentialSearch(unsortedlist, searchitem):
9      found = len(unsortedlist)-1
10     while found>=0 and unsortedlist[found]!=searchitem:
11         found -= 1
12     return found
13
14 unsortedlist = [1, 3, 7, 4, 28, 9, 11, 10, 8]
15 print(SequentialSearch(unsortedlist, 3))
16 print(SequentialSearch(unsortedlist, 13))
```

Drugi algorytm stosuje wartownika dodanego na końcu listy w celu uproszczenia warunku sprawdzania zakończenia pętli, co zwiększa szybkość jego działania w stosunku do poprzedniego.

```
5  @author: Adrian Horzyk
6  """
7
8  def SequentialGuardSearch(unsortedlist, searchitem):
9      unsortedlist.append(searchitem)
10     found = 0
11     while unsortedlist[found] != searchitem:
12         found += 1
13     if found == len(unsortedlist) - 1:
14         found = -1
15     return found
16
17 unsortedlist = [1, 3, 7, 4, 28, 9, 11, 10, 8]
18 print(SequentialGuardSearch(unsortedlist, 4))
19 print(SequentialGuardSearch(unsortedlist, 13))
```

WYSZUKIWANIE POŁÓWKOWE

Przeszukiwanie połówkowe – dzieli przeszukiwaną uporządkowaną strukturę liniową (np. listę, tablicę) na 2 części, wyznaczając indeks środkowego elementu i sprawdzając, czy jest on równy poszukiwanemu.
Jeśli to nie jest, wtedy powtarza tą samą procedurę rekurencyjnie na tej części struktury, która może zawierać poszukiwany element:

```
 5 @author: Adrian Horzyk
 6 """
 7
 8 def BinarySearch(sortedlist, searchitem):
 9     first = 0
10     last = len(sortedlist)-1
11     found = -1
12     while first<=last and found<0:
13         mid = (first + last)//2
14         if sortedlist[mid] == searchitem:
15             found = mid
16         elif sortedlist[mid] > searchitem:
17             last = mid-1
18         else:
19             first = mid+1
20     return found
21
22 sortedlist = [1, 3, 4, 7, 8, 9, 10, 11, 28]
23 print(BinarySearch(sortedlist, 4))
24 print(BinarySearch(sortedlist, 13))
```

Algorytm wykonuje znacznie mniejszą ilość porównań niż te poprzednie!

WYSZUKIWANIE INTERPOLOWANE

Przeszukiwanie interpolowane – umożliwia bardzo szybkie wyszukiwanie elementów w posortowanej liniowej strukturze danych o mniej więcej równomiernym rozkładzie wartości w przeszukiwanym przedziale:

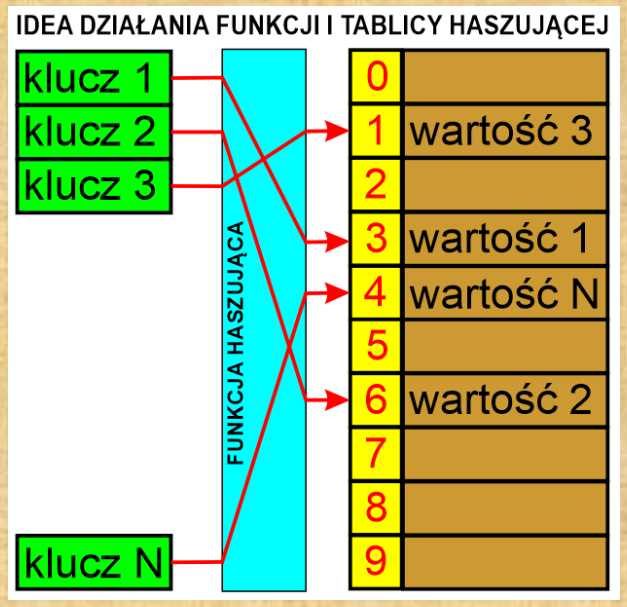
```
@author: Adrian Horzyk
"""

def InterpolationSearch(asclist, searchitem):
    first = 0
    last = len(asclist)-1
    found = -1
    while first<last and found<0:
        interpol = first + ((last - first) * (searchitem - asclist[first])) / (asclist[last] - asclist[first])
        if asclist[interpol] < searchitem:
            first = interpol+1
        elif asclist[interpol] > searchitem:
            last = interpol-1
        else:
            found = interpol
    return found

ascendantSortedList = [1, 3, 4, 7, 8, 9, 10, 11, 28]
print(InterpolationSearch(ascendantSortedList, 4))
print(InterpolationSearch(ascendantSortedList, 13))
```

Przy tych założeniach algorytm wykonuje jeszcze mniejszą ilość porównań niż te poprzednie, gdyż próbuje obliczyć (zgadnąć) indeks poszukiwanego elementu na podstawie jego wartości oraz wartości pierwszego i ostatniego elementu w przeszukiwanym przedziale. Jeśli mu się to nie uda, wtedy zawęża obszar poszukiwać na przedziału, który zawiera poszukiwany element.

TABLICE I FUNKCJE HASHUJĄCE

Wyszukiwanie z wykorzystaniem tablicy haszującej (hash table) umożliwia teoretycznie najszybsze wyszukiwanie (w czasie stałym) pod warunkiem określenia takiej funkcji haszującej, iż możliwe jest dokładne obliczenie w tablicy indeksu początku odpowiedniej krótkiej listy, w której znajduje się poszukiwany element. Podobnie więc jak w wyszukiwaniu interpolowanym równomierny rozkład wartości danych jest zaletą.

Ponadto stworzenie **funkcji haszującej** wymaga zwykle znajomości wartości minimalnej i maksymalnej, co kosztuje czas liniowy O(n) i wymaga dodatkowego miejsca w pamięci zależnego liniowo od n, a więc również O(n).



Funkcja haszująca h(x) przekształca klucz w indeks w tablicy haszującej, której wielkość może być mniejsza niż ilość kluczy, lecz wtedy pod tym samym indeksem mieści się kilka kluczy, które zwykle organizowane są w postaci listy (nieposortowanej lub posortowanej przez proste wstawianie).

Funkcja haszująca modularna h(k) uniemożliwia przekroczenie przewidywanego zakresu kluczy: h(k) = k % m, gdzie m to rozmiar **tablicy mieszającej (haszującej)**.

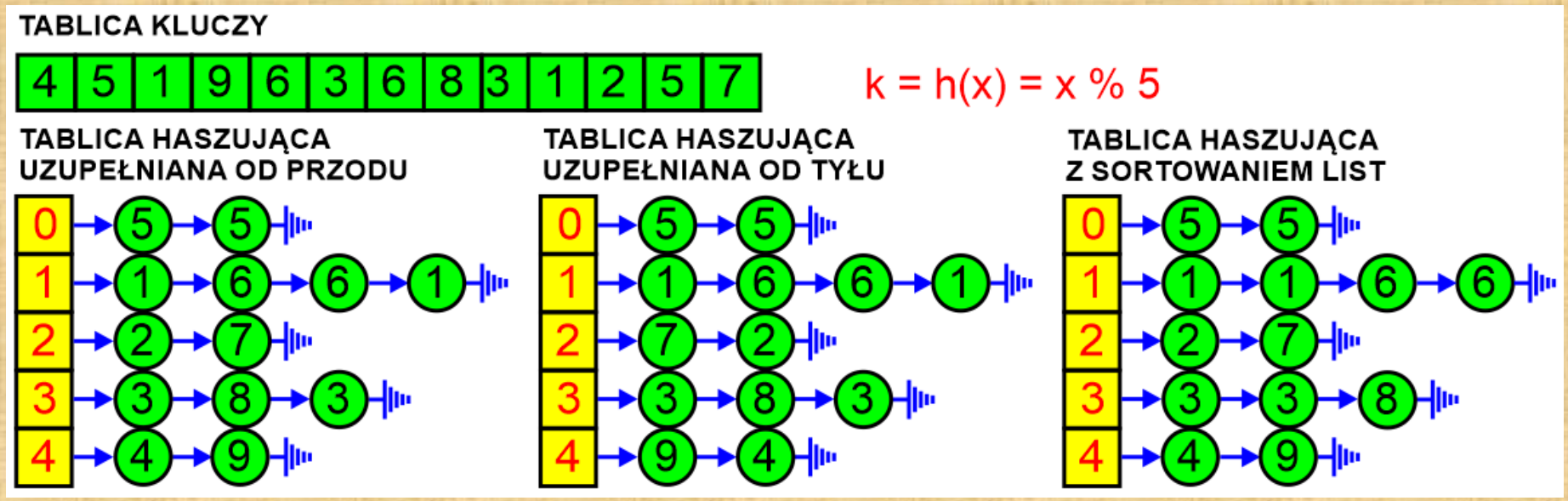
Tablice haszujące nazywane są też **tablicami mieszającymi** lub **tablicami z haszowaniem**.

Niektóre **tablice haszujące** można również wykorzystać do sortowania danych. Najpierw tworzymy taką tablicę krótkich list, a następnie łączymy je ze sobą tworząc listę wynikową lub przepisujemy kolejno wyniki do tablicy wynikowej, co jednak kosztuje miejsce w pamięci zależne od ilości danych (**not in place**).

Słowniki w Pythonie są wewnętrznie implementowane jako tablice haszujące.

PRZYKŁAD TABLICY HASZUJĄCEJ

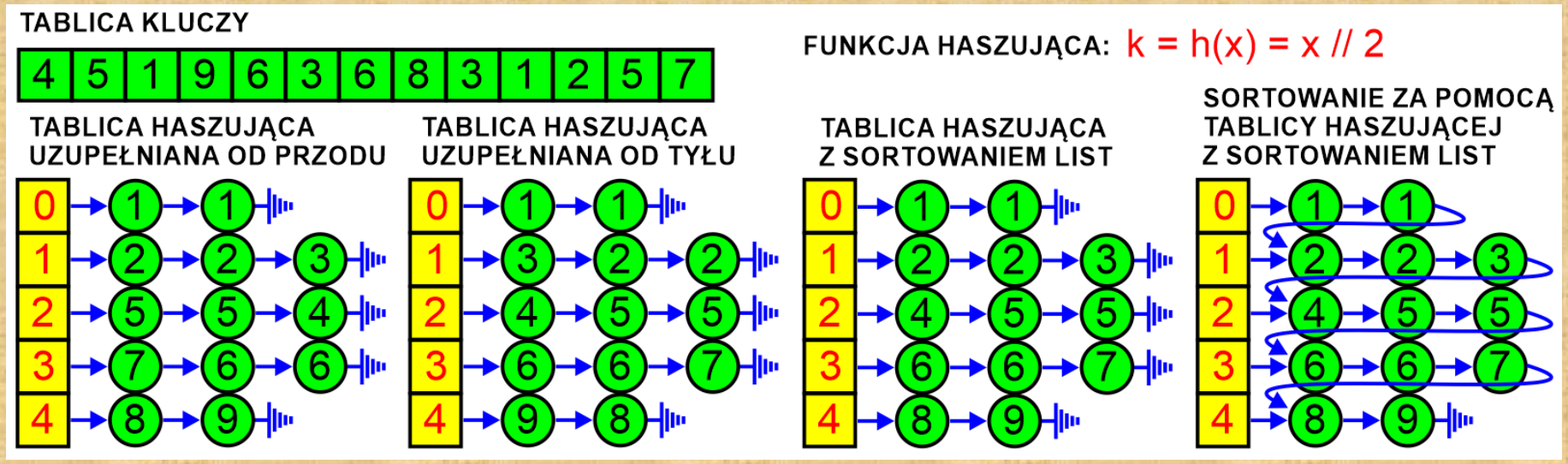
Typowa tablica haszująca to tablica krótkich posortowanych list, w których indeks początku listy w tablicy wyznaczamy przy pomocy funkcji haszującej, którą dobiera się do konkretnego rodzaju i rozkładu kluczy w przestrzeni.
Idealnie jest, gdy uda się znaleźć przyporządkowanie zwarte, które N kluczom przydziela unikalne N wartości z zakresu od 0 do N-1.



Listy najprościej i najszybciej O(1) uzupełnia się od przodu w miejscu głowy listy, ew. od tyłu, dodając nowy obiekt na końcu listy (append), uzyskując nieposortowane listy obiektów, które potem są przeglądane liniowo w celu odnalezienie klucza. Biorąc pod uwagę to, iż elementów na liście jest niewiele, przyjmuje się, iż zabiera to stałą ilość czasu.

SORTOWANIE KUBEŁKOWE

Odpowiednio dobrana funkcja haszująca wespół z odpowiednim rozkładem danych w przestrzeni pozwala wykorzystać funkcję haszującą do szybkiego O(n) sortowania danych:



Do krótkich list można przez proste wstawianie dodawać nowe elementy w kolejności rosnącej lub malejącej uzyskują tablicę posortowanych list.
Takie sortowanie jest również wykonywane w czasie stałym przy założeniu niewielkich list.

W przypadku konieczności wyszukania kluczy elementów na poszczególnych podlistach możliwe jest wykorzystanie szybszych algorytmów, np. wyszukiwanie połówkowe.

Ponadto można takie posortowane listy wykorzystać do budowy posortowanej listy wynikowej wszystkich obiektów, co odbywa się teoretycznie średnio w czasie liniowym O(n) i nazywa się sortowaniem kubełkowym (bucket sort).

SORTOWANIE SZYBKIE

Sortowanie szybkie (*quick sort*) – jest jednym z najbardziej efektywnych algorytmów sortowania ogólnego przeznaczenia wykorzystującym strategię „dziel i zwyciężaj”, lecz niestety nie jest sortowaniem stabilnym, co wyklucza go z pewnych zastosowań, gdzie stabilność jest wymagana!

Sortowanie stabilne to takie, które zachowuje względną kolejność kluczy o tych samych wartościach w ciągu posortowanym.

Przedstawiony algorytm jest algorytmem rekurencyjnym, gdyż wywołuje sam siebie i posiada warunek stopu, tj. dla first >= j oraz i >= last nie dojdzie do ponownego wywołania rekurencyjnego.

```
@author: Adrian Horzyk
"""
def QuickSort(arr, first, last):
    pivot = arr[(last+first)/2]
    i = first
    j = last
    while i<=j:
        while arr[i]<pivot:
            i+=1
        while arr[j]>pivot:
            j-=1
        if i<=j:
            temp = arr[i]
            arr[i] = arr[j]
            arr[j] = temp
            i+=1
            j-=1
            print(arr)
    if first<j:
        QuickSort(arr, first, j)
    if i<last:
        QuickSort(arr, i, last)

arr = [28, 9, 3, 7, 4, 8, 10, 1, 11]
print(arr)
QuickSort(arr, 0, len(arr)-1)

```

```
[28, 9, 3, 7, 4, 8, 10, 1, 11]
[1, 9, 3, 7, 4, 8, 10, 28, 11]
[1, 4, 3, 7, 9, 8, 10, 28, 11]
[1, 3, 4, 7, 9, 8, 10, 28, 11]
[1, 3, 4, 7, 9, 8, 10, 28, 11]
[1, 3, 4, 7, 8, 9, 10, 28, 11]
[1, 3, 4, 7, 8, 9, 10, 28, 11]
[1, 3, 4, 7, 8, 9, 10, 28, 11]
[1, 3, 4, 7, 8, 9, 10, 11, 28]
```

To typowa dla Pythona forma skrótowa, ale niezbyt efektywna.

```
def quicksort(L):
    if L == []: return []
    return quicksort([x for x in L[1:] if x< L[0]]) + L[0:1] + quicksort([x for x in L[1:] if x>=L[0]])
```

REKURENCJA

Algorytm rekurencyjny to taki, który **wywołuje sam siebie** i posiada **warunek stopu**, który określa moment, w którym do dalszych wywołań nie dojdzie.

Warunek stopu umożliwia zatrzymanie kolejnych wywołań i wyjście z procedury rekurencyjnej.

Parametry wywołania procedury / funkcji rekurencyjnej mogą się zmieniać.

Należy pamiętać o tym, iż każde wywołanie procedury / funkcji pociąga za sobą odłożenia pewnych danych **na stosie systemowym**, co dodatkowo wpływa na zmniejszenie efektywności wykonania programu!

Algorytm sortowania szybkiego można przedstawić w formie rekurencyjnej.

Algorytmy rekurencyjne można zawsze przekształcić na **algorytmy iteracyjne**.

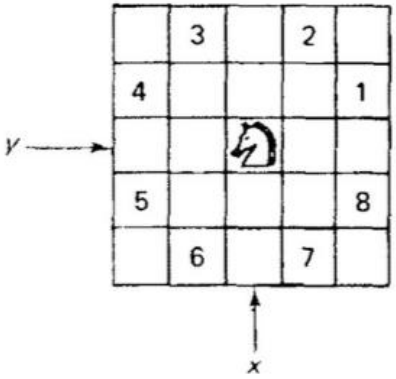
```
5  @author: Adrian Horzyk
6  """
7  def QuickSort(arr, first, last):
8      pivot = arr[(last+first)/2]
9      i = first
10     j = last
11     while i<=j:
12         while arr[i]<pivot:
13             i+=1
14         while arr[j]>pivot:
15             j-=1
16         if i<=j:
17             temp = arr[i]
18             arr[i] = arr[j]
19             arr[j] = temp
20             i+=1
21             j-=1
22             print(arr)
23     if first<j:
24         QuickSort(arr, first, j)
25     if i<last:
26         QuickSort(arr, i, last)
27
28 arr = [28, 9, 3, 7, 4, 8, 10, 1, 11]
29 print(arr)
30 QuickSort(arr, 0, len(arr)-1)
31
32
```

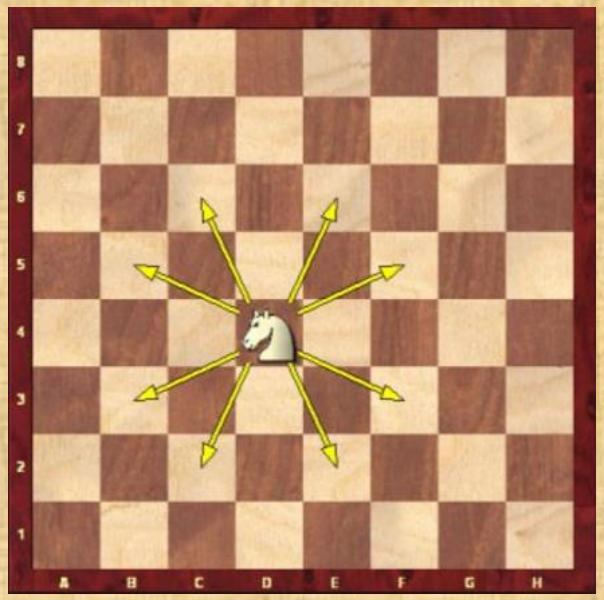
PRZYKŁAD REKURENCJI

Skoczek szachowy ma za zadanie obskoczyć wszystkie pola szachownicy NxN dokładnie jeden raz. Można do tego wykorzystać algorytm rekurencyjny, który próbuje wykonać jeden z dostępnych ruchów, a jeśli dana sekwencja skoków nie prowadzi do sukcesu, wtedy się wycofać próbować ponownie inną:

algorytm w pseudokodzie

```
procedure próbuj następny ruch;
begin zapoczątkuj wybieranie ruchów;
   repeat wybierz następnego kandydata z listy następnych ruchów;
      if dopuszczalny then
      begin zapisz ruch;
         if na szachownicy są wolne pola then
         begin próbuj następny ruch;
            if nieudany then
               wykreśl ostatni zapis ruchu
         end
      end
   until (ruch był udany) ∨ (nie ma następnego kandydata)
end
```

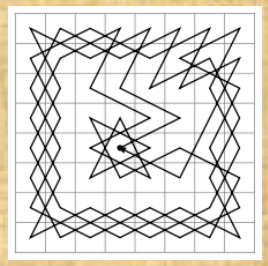


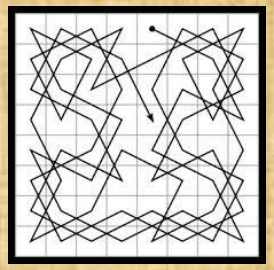


1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

23	10	15	4	25
15	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29





BIBLIOGRAFIA I LITERATURA UZUPEŁNIAJĄCA

1. L. Banachowski, K. Diks, W. Rytter: „Algorytmy i struktury danych”, WNT, Warszawa, 2001.
2. Z. Fortuna, B. Macukow, J. Wąsowski: „Metody numeryczne”, WNT, Warszawa, 1993.
3. J. i M. Jankowscy: „Przegląd metod i algorytmów numerycznych”, WNT, Warszawa, 1988.
4. A. Kiełbasiński, H. Schwetlick: „Numeryczna algebra liniowa”, WNT, Warszawa 1992.
5. M. Sysło: „Elementy Informatyki”.
6. A. Szepietowski: „Podstawy Informatyki”.
7. R. Tadeusiewicz, P. Moszner, A. Szydełko: „Teoretyczne podstawy informatyki”.
8. W. M. Turski: „Propedeutyka informatyki”.
9. N. Wirth: „Wstęp do programowania systematycznego”.
10. N. Wirth: „ALGORYTMY + STRUKTURY DANYCH = PROGRAMY”.