Algorytmy i struktury danych

Wykład 5 - Algorytmy i ich analiza (ciąg dalszy)

Janusz Szwabiński

Plan wykładu:

- Studium przypadku - analiza anagramów
- Model matematyczny
- Klasyfikacja algorytmów

Studium przypadku - analiza anagramów

Anagram oznacza wyraz, wyrażenie lub całe zdanie powstałe przez przestawienie liter bądź sylab innego wyrazu lub zdania, wykorzystujące wszystkie litery (głoski bądź sylaby) materiału wyjściowego:

- kebab $\leftrightarrow$ babek
- Gregory House $\leftrightarrow$ Huge ego, sorry
- "Quid est veritas?" (*Co to jest prawda?*, Piłat) $\leftrightarrow$ "Vir est qui adest" (*Człowiek, który stoi przed tobą*, Jezus)

Testowanie, czy łańcuchy znaków są anagramami, to przykład zagadnienia, które można rozwiązać algorytmami o różnym tempie asymptotycznego wzrostu, a jednocześnie na tyle prostego, aby można było o tym opowiedzieć w ramach kursu ze wstępu do programowania.

Rozwiązanie 1: "odhaczanie" liter

Jednym z możliwych rozwiązań naszego problemu jest sprawdzenie, czy litera z jednego łańcucha znajduje się w drugim. Jeżeli tak, "odhaczamy" ją i powtarzamy czynność dla pozostałych liter. Jeżeli po zakończeniu tej operacji wszystkie litery w drugim łańcuchu zostaną "odhaczone", łańcuchy muszą być anagramami. Odhaczanie możemy zrealizować poprzez zastąpienie odnalezionej litery wartością specjalną `None`.

In [1]:

```
def anagramSolution1(s1,s2):
    alist = list(s2) #zamień drugi łańcuch na listę

    pos1 = 0
    stillOK = True

    while pos1 < len(s1) and stillOK:
        pos2 = 0
        found = False
        while pos2 < len(alist) and not found:
            if s1[pos1] == alist[pos2]:    #sprawdzamy literę s1[pos1]
                found = True
            else:
                pos2 = pos2 + 1

        if found:
            alist[pos2] = None
        else:
            stillOK = False #przerwij, jeśli litery nie ma

        pos1 = pos1 + 1 #pozycja następnej litery w łańcuchu s1

    return stillOK

print(anagramSolution1('abcd','dcba'))
```

```
True
```

Dla każdego znaku z łańcucha $s1$ musimy wykonać iterację po maksymalnie n elementach listy $s2$. Każda pozycja w liście $s2$ musi zostać odwiedzona raz w celu odhaczenia. Dlatego całkowita liczba wizyt elementów listy $s2$ jest sumą liczb naturalnych od 1 do n:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Dla dużych n wyraz $\frac{1}{2}n^2$ będzie dominował nad $\frac{1}{2}n$. Dlatego algorytm jest klasy $O(n^2)$.

Rozwiązanie 2: sortowanie i porównywanie

Zauważmy, że jeżeli $s1$ i $s2$ są anagramami, muszą składać się z tych samych liter, występujących taką samą liczbę razy. Jeżeli więc posortujemy każdy z łańcuchów alfabetycznie od 'a' do 'z', powinniśmy otrzymać dwa takie same łańcuchy.

Do posortowania wykorzystamy metodę `sort`:

In [2]:

```
def anagramSolution2(s1,s2):
    alist1 = list(s1) #konieczne, żeby skorzystać z sort
    alist2 = list(s2)

    alist1.sort()
    alist2.sort()

    pos = 0
    matches = True

    while pos < len(s1) and matches:
        if alist1[pos]==alist2[pos]:
            pos = pos + 1
        else:
            matches = False

    return matches

print(anagramSolution2('abcde','edcba'))
```

```
True
```

Na pierwszy rzut oka może się wydawać, że algorytm jest klasy $O(n)$, ponieważ wykonujemy tylko jedną iterację po elementach łańcuchów. Jednak wywołanie metody `sort` również "kosztuje", najczęściej $O(n^2)$ lub $O(n \log n)$. Dlatego czas wykonania będzie zdominowany przez operację sortowania.

Rozwiązanie 3: algorytm siłowy (ang. *brute force*)

Metoda siłowa rozwiązania jakiegoś zadania polega na wyczerpaniu wszystkich możliwości. Dla anagramów oznacza to wygenerowanie listy wszystkich możliwych łańcuchów ze znaków łańcucha $s1$ i sprawdzenie, czy $s2$ znajduje się na tej liście. Nie jest to jednak zalecane podejście, przynajmniej w tym przypadku. Zauważmy mianowicie, że dla ciągu znaków $s1$ o długości n mamy n wyborów pierwszego znaku, $(n-1)$ możliwości dla znaku na drugiej pozycji, $(n-2)$ na trzeciej pozycji itd. Musimy zatem wygenerować $n!$ łańcuchów znaków.

Dla ustalenia uwagi przyjmijmy, że $s1$ składa się z 20 znaków. Oznacza to konieczność wygenerowania

In [3]:

```
import math
math.factorial(20)
```

Out[3]:

```
2432902008176640000
```

łańcuchów znaków, a następnie odszukanie wśród nich ciągu $s2$. Widzieliśmy już wcześniej, ile czasu zajmuje algorytm klasy $O(n!)$, dlatego nie jest to polecane podejście do zagadnienia anagramów.

Rozwiązanie 4: zliczaj i porównuj

Jeżeli $s1$ i $s2$ są anagramami, będą miały tę samą liczbę wystąpień litery 'a', tę samą litery 'b' itd. Dlatego możemy zliczyć liczbę wystąpień poszczególnych znaków w łańcuchach i porównać te liczby ze sobą:

In [4]:

```
def anagramSolution4(s1,s2):
    c1 = [0]*26     #dla ułatwienia ograniczamy się do języka angielskiego
    c2 = [0]*26

    for i in range(len(s1)):
        pos = ord(s1[i])-ord('a') #pozycja znaku w alfabecie
        c1[pos] = c1[pos] + 1

    for i in range(len(s2)):
        pos = ord(s2[i])-ord('a')
        c2[pos] = c2[pos] + 1

    j = 0
    stillOK = True
    while j<26 and stillOK:
        if c1[j]==c2[j]:
            j = j + 1
        else:
            stillOK = False

    return stillOK

print(anagramSolution4('apple','pleap'))
```

```
True
```

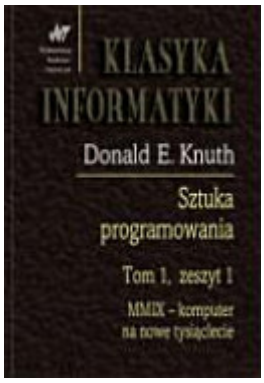
Również w przypadku tej metody mamy do czynienia z iteracjami, jednak teraz żadna z nich nie jest zagnieżdżona. Dodając kroki konieczne do wykonania w tych iteracjach do siebie otrzymamy

$$T(n) = 2n + 26$$

Jest to zatem algorytm klasy $O(n)$, czyli najszybszy ze wszystkich prezentowanych. Zauważmy jednak, że lepszą wydajność uzyskaliśmy kosztem większego obciążenia pamięci (dwie dodatkowe listy $c1$ i $c2$). To sytuacja bardzo często spotykana w praktyce. Dlatego programując, nie raz staniemy przed koniecznością wyboru, który z zasobów (czas procesora czy pamięć) należy poświęcić.

Model matematyczny

- analiza doświadczalna algorytmu pozwala przewidzieć jego wydajność bez zrozumienia jego działania
- model matematyczny czasu pracy algorytmu pomaga zrozumieć to działanie
- koncepcja modelu matematycznego została opracowana i spopularyzowana przez Donalda Knutha pod koniec lat 60 XX wieku
  - *Sztuka programowania* (ang. *The Art of Computer Programming*, w skrócie *TAOCP*)



  - http://cs.stanford.edu/~uno/ (http://cs.stanford.edu/~uno/)

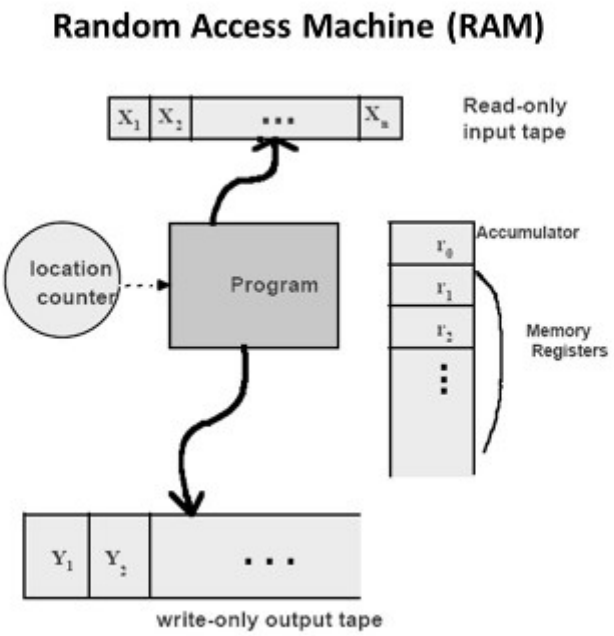
Złożoność obliczeniowa algorytmu

- koszt realizacji algorytmu, czyli ilość zasobów komputera niezbędnych do wykonania algorytmu
- **złożoność czasowa**
  - pomiar rzeczywistego czasu zegarowego jest mało użyteczny ze względu na silną zależność od sposobu realizacji algorytmu, użytego kompilatora oraz maszyny, na której algorytm wykonujemy
  - liczba operacji podstawowych w zależności od rozmiaru wejścia
  - operacje podstawowe: podstawienie, porównanie lub prosta operacja arytmetyczna
- **złożoność pamięciowa**
  - miara ilości wykorzystanej pamięci
  - możliwe jest obliczanie rozmiaru potrzebnej pamięci fizycznej wyrażonej w bitach lub bajtach
  - jako ilość często przyjmuje się użytą pamięć maszyny abstrakcyjnej

Dygresja - maszyny abstrakcyjne

- istniejące komputery różnią się między sobą istotnymi parametrami (np. liczba i rozmiar rejestrów, udostępniane operacje matematyczne)
- komputery podlegają ciągłym ulepszeniom
- algorytmy często analizuje się, wykorzystując abstrakcyjne modele obliczeń, np.:
  - maszyna RAM
  - maszyna Turinga

Maszyna RAM



- części składowe:
  - jednostka kontrolna (program)
  - jednostka arytmetyczna (procesor)
  - pamięć
  - jednostka wejścia
  - jednostka wyjścia
- **jednostka kontrolna** zawiera program oraz jego rejestr (rejestr wskazuje na instrukcję do wykonania)
- **jednostka arytmetyczna** wykonuje podstawowe operacje arytmetyczne
- **pamięć** składa się z ponumerowanych komórek, z których każda może przechować dowolną liczbę całkowitą
  - liczba komórek jest nieograniczona
  - nie ma ograniczeń na rozmiar liczby
  - indeks komórki to jej adres
  - komórka o adresie 0 to tzw. rejestr roboczy
- **jednostka wejścia** składa się z taśmy i głowicy
  - taśma jest podzielona na komórki
  - taśma jest nieskończona
  - każda komórka przechowuje jedną liczbę całkowitą
  - w danej chwili czasu głowica odczytuje tylko jedną komórkę
  - po odczytaniu głowica przesuwa się o jedną komórkę w prawo
- **jednostka wyjścia** składa się z taśmy i głowicy
  - taśma jest podzielona na komórki
  - taśma jest nieskończona
  - do każdej komórki może być zapisana jedna liczba całkowita
  - po zapisie głowica przesuwa się o jedną komórkę w prawo
- konfiguracja maszyny to odwzorowanie, które przypisuje liczbę naturalną do każdej komórki wejściowej, wyjściowej i pamięci oraz do rejestru programu
- obliczenia to sekwencja konfiguracji, z których pierwsza jest konfiguracją początkową, a każda następna została wygenerowana zgodnie z programem
- dopuszalne instrukcje:
  - instrukcje przesunięcia: `LOAD` i `STORE` (kopiowanie do i z rejestru roboczego)
  - instrukcje arytmetyczne: `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`

    - instrukcje we/wy: `READ`, `WRITE`
    - instrukcje skoku: `JUMP`, `JZERO`, `JGTZ`
    - instrukcje stopu: `HALT`, `ACCEPT`, `REJECT`
- operand to albo liczba j albo zawartość j-tej komórki pamięci
- program to dowolna sekwencja powyższych instrukcji wykonywana na operandach

Maszyna Turinga

- https://pl.wikipedia.org/wiki/Maszyna_Turinga (https://pl.wikipedia.org/wiki/Maszyna_Turinga)

Całkowity czas pracy algorytmu

- **całkowity czas pracy algorytmu** (lub programu) to suma wartości

$$koszt\ operacji \times częstotliwość\ występowania$$

  po wszystkich operacjach występujących w algorytmie

Koszt wykonania pojedynczej operacji

- pierwsze komputery dostarczane były z instrukcją zawierającą dokładny czas wykonania każdej operacji
- obecnie można taki koszt oszacować eksperymentalnie - wykonujemy np. bilion dodawań i wyliczamy średnią z czasu wykonania pojedynczej operacji
- w praktyce zakłada się, że podstawowe operacje (dodawanie, odejmowanie, mnożenie, dzielenie, przypisanie, porównanie, deklaracja zmiennej itp) wykonywane na standardowych typach danych zajmują pewien stały czas wykonania

Częstotliwość występowania operacji

Niech:

In [5]:

```
import random

lista = [random.randint(-10,10) for r in range(10)]
```

In [6]:

```
lista
```

Out[6]:

```
[10, 5, 9, -4, -9, 1, 3, 10, 1, 7]
```

Rozważmy program:

In [7]:

```
count = 0
for i in lista:
    if i==0:
        count = count + 1
```

In [8]:

```
count
```

Out[8]:

```
0
```

Częstotliwość wykonywania poszczególnych operacji w tym programie jest następująca:

Operacja	Liczba wystąpień
przypisanie	1
porównanie	n
dostęp do listy	n
inkrementacja `i`	n
inkrementacja `count`	$\leq n$

Uproszczenie liczenia częstości występowania

- liczenie wszystkich operacji mających wpływ na czas pracy algorytmu może być żmudne

> "It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings." (A. Turing, *Rounding-off errors in matrix processes*, 1947)

- **wybór operacji dominującej**
  - np. przypisanie, porównanie, działanie arytmetyczne, dostęp do tablicy/listy
  - najczęściej wybiera się tę, która najwięcej kosztuje i najczęściej występuje

In [9]:

```
lista
```

Out[9]:

```
[10, 5, 9, -4, -9, 1, 3, 10, 1, 7]
```

In [10]:

```
count = 0
for i in range(len(lista)):
    for j in range(i+1,len(lista)):
        if lista[i]+lista[j]==0:
            count = count + 1
```

In [11]:

```
count
```

Out[11]:

1

Operacja dominująca	Liczba wystąpień
dostęp do listy	$n(n-1)$

Nasuwa się pytanie, dlaczego liczba dostępów do listy w tym przykładzie wynosi $n(n-1)$. Aby to wyjaśnić, przyjrzyjmy się liczbom dostępów do listy dla poszczególnych wartości iteratorów i oraz j:

Wartości i	Wartości j	Liczba dostępów do listy
0	$1, 2, 3, 4, \ldots, n-1$	$2(n-1)$
1	$2, 3, 4, \ldots, n-1$	$2(n-2)$
2	$3, 4, \ldots, n-1$	$2(n-3)$
$\ldots$	$\ldots$	$\ldots$
$n-2$	$n-1$	2
$n-1$	None	None

Sumując liczbę operacji, otrzymamy:

$$2(n-1+n-2+n-3+\cdots+1) = 2\left(n*n-\sum_{i=1}^{n} i\right) = 2n^2 - 2\frac{n(n+1)}{2} = n(n-1)$$

- **notacja przybliżona**
  - notacja dużego O
  - notacja dużego Ω (https://pl.wikipedia.org/wiki/Asymptotyczne_tempo_wzrostu (https://pl.wikipedia.org/wiki/Asymptotyczne_tempo_wzrostu))
  - notacja Θ

Klasyfikacja algorytmów

Rodzaje złożoności obliczeniowej

- **złożoność pesymistyczna** (ang. *worst-case*)
  - maksymalna ilość zasobu potrzebna do wykonania algorytmu dla dowolnego wejścia
- **złożoność oczekiwana** (ang. *average-case*)
  - oczekiwana ilość zasobu potrzebna do wykonania algorytmu
  - zależy istotnie od założeń na temat rozważanej przestrzeni probabilistycznej danych wejściowych
  - może wymagać trudnych analiz matematycznych

Porównywanie algorytmów

W przykładzie z anagramami pierwszy algorytm rozwiązywał zadany problem w czasie

$$T_1(n) = \frac{1}{2}n^2 + \frac{1}{2}n,$$

natomiast czwarty algorytm w czasie

$$T_4(n) = 2n + 26$$

Chcemy odpowiedzieć na pytanie, który z nich jest lepszy.

In [14]:

```
%matplotlib inline
```

In [15]:

```
import matplotlib.pyplot as plt
import numpy as np
```

In [16]:

```
n = np.arange(1,20)
```

In [17]:

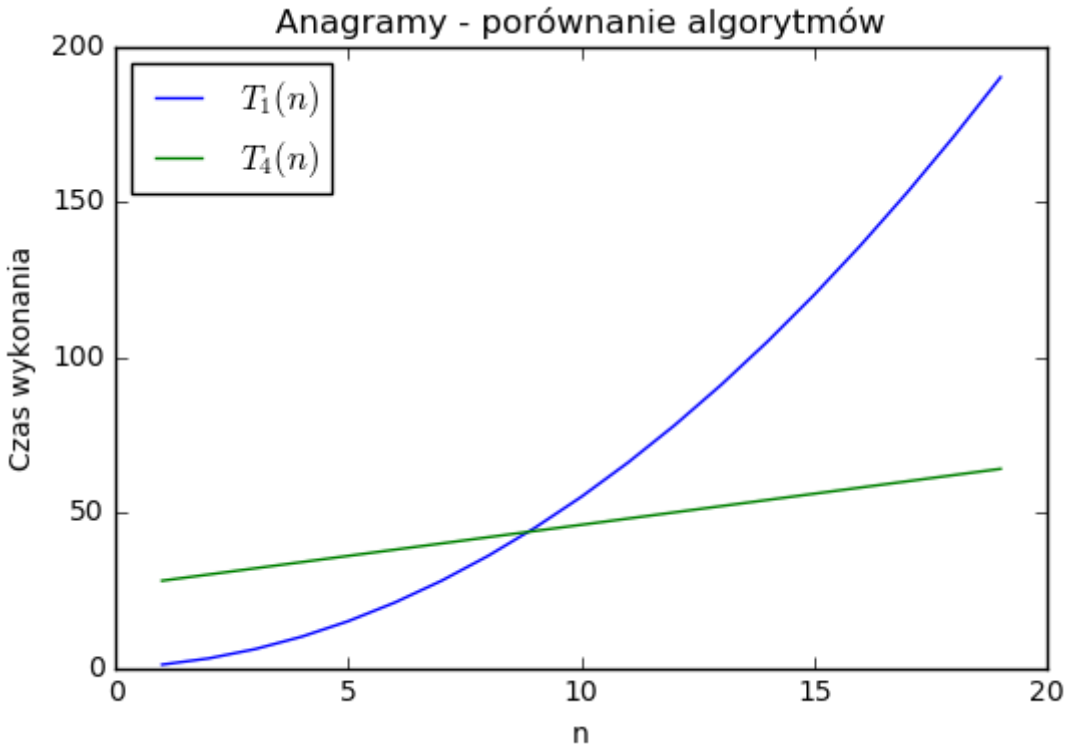
```
def T1(x):
    return 0.5*x*(1+x)

def T4(x):
    return 2*x+26
```

In [18]:

```
plt.plot(n,T1(n),label="$T_1(n)$")
plt.plot(n,T4(n),label="$T_4(n)$")
plt.title("Anagramy - porównanie algorytmów")
plt.xlabel("n")
plt.ylabel("Czas wykonania")
plt.legend(loc=2)
```

Out[18]:

```
<matplotlib.legend.Legend at 0x7fc1b1bc5278>
```



- dla bardzo krótkich słów algorytm T_1 okazuje się szybszy!
- funkcja T_4 zachowuje się jednak dużo lepiej, gdy n rośnie
- ponieważ interesuje nas głównie czas asymptotyczny, wybieramy T_4:

$$T_4(n) = O(n)$$
$$T_1(n) = O(n^2)$$

Rzędy złożoności obliczeniowej

W zależności od asymptotycznego tempa wzrostu, algorytmy dzieli się na klasy złożoności obliczeniowej. Najczęściej wyróżnia się:

$O(n)$	Funkcja
1	stała (nie zależy od rozmiaru wejścia)
$\log n$	logarytmiczna
n	liniowa
$n \log n$	liniowo-logarytmiczna (quasi-liniowa)
n^2	kwadratowa
n^c	wielomianowa
c^n	wykładnicza
$n!$	silnie wykładnicza

Złożoność logarytmiczna

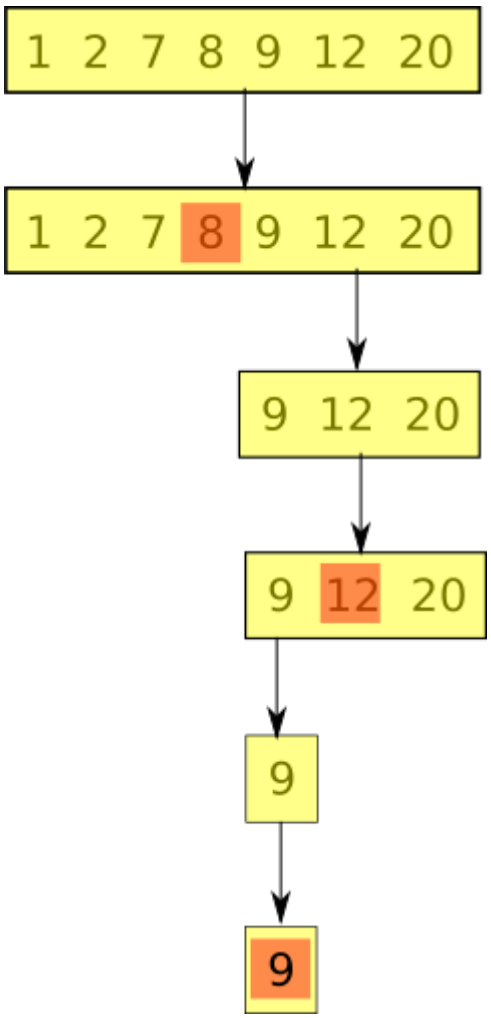
- typowa dla algorytmów, w których problem przedstawiony dla danych o rozmiarze n da się sprowadzić do problemu z danymi o rozmiarze $n/2$
- algorytmy o złożoności logarytmicznej należą do **klasy P** problemów, tzn. problemów **łatwych**, które potrafimy rozwiązać w czasie co najwyżej wielomianowym

Przykład: wyszukiwanie binarne

> Dany jest posortowany zbiór danych (np. w liście) A oraz pewien element `item`. Chcemy odpowiedzieć na pytanie, czy `item` znajduje się w A.

Algorytm:

1. Sprawdź środkowy element tablicy. Jeśli jest równy `item`, zakończ przeszukiwanie.
2. Jeśli środkowy element jest większy niż `item`, kontynuuj wyszukiwanie w lewej części listy.
3. W przeciwnym razie, szukaj elementu `item` w prawej części listy



In [19]:

```
def binary_search(a, x, lo=0, hi=None):
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        midval = a[mid]
        if midval == x:
            return mid
        elif midval < x:
            lo = mid + 1
        else:
            hi = mid
    return -1
```

In [20]:

```
A = [1,2,7,8,9,12,20]
```

In [21]:

```
binary_search(A,9)
```

Out[21]:

4

Dla danej listy wejściowej algorytm wymaga jednego sprawdzenia elementu środkowego, a następnie przeszukuje binarnie jedną z jej połówek. Zatem czas wykonania da się opisać równaniem

$$T(n) \leq T\left(\frac{n}{2}\right) + 1$$
$$T(1) = 1$$

Załóżmy, że $n = 2^m$ (czyli $m = \log_2 n$). Wówczas:

$$\begin{aligned} T(2^m) &\leq T\left(\tfrac{2^m}{2}\right) + 1 \\ &\leq T\left(\tfrac{2^m}{4}\right) + 1 + 1 \\ &\leq T\left(\tfrac{2^m}{8}\right) + 1 + 1 + 1 \\ &\leq T\left(\tfrac{2^m}{16}\right) + 1 + 1 + 1 + 1 \\ &\dots \\ &\leq T\left(\tfrac{2^m}{2^m}\right) + m = T(1) + m \end{aligned}$$

Zatem

$$T(n) \leq 1 + \log_2 n,$$

czyli wyszukiwanie binarne jest przykładem algorytmu o złożoności logarytmicznej.

Złożoność liniowa

- typowa dla algorytmów, w których dla każdego elementu danych wejściowych wymagana jest pewna stała liczba operacji
- algorytmy tego typu również należą do **klasy P** problemów łatwych.

In [22]:

```
def linear_search(a,x):
    for i in a:
        if i == x:
            return a.index(i)
    return -1
```

In [23]:

```
A
```

Out[23]:

```
[1, 2, 7, 8, 9, 12, 20]
```

In [24]:

```
linear_search(A,9)
```

Out[24]:

4

Mamy tutaj maksymalnie n iteracji i n porównań. Jest to zatem algorytm klasy $O(n)$.

Złożoność liniowo-logarytmiczna

- typowa dla algorytmów, w których problem postawiony dla danych o rozmiarze n daje się sprowadzić w liniowej liczbie operacji do dwóch problemów o rozmiarach $n/2$
- również należy dla klasy P problemów łatwych
- przykłady: sortowanie przez scalanie, ogólnie - algorytmy typu **dziel i rządź**

Złożoność kwadratowa

- typowa dla algorytmów, w których dla każdej pary elementów wejściowych trzeba wykonać stałą liczbę operacji podstawowych
- klasa P
- przykłady:
  - anagramy metodą odhaczania
  - 2SUM metodą brute-force

Złożoność wielomianowa

- typowa dla algorytmów, w których dla każdej krotki elementów wejściowych wykonywana jest stała liczba operacji podstawowych
- klasa P
- przykłady:
  - 3SUM metodą brute-force

Złożoność wykładnicza

- typowa dla algorytmów, w których dla każdego podzbioru danych wejściowych należy wykonać stałą liczbę działań
- **klasa NP**
- przykłady
  - wieża z Hanoi (o tym na następnym wykładzie)

Złożoność silnie wykładnicza

- stała liczba działań wykonywana dla każdej permutacji danych wejściowych
- klasa NP
- przykład:
  - problem komiwojażera rozwiązywany metodą siłową

n	10	20	30	40	50	60
$O(n)$	0,00001s	0,00002s	0,00003s	0,00004s	0,00005s	0,00006s
$O(n^2)$	0,0001s	0,0004s	0,0009s	0,0016s	0,0025s	0,0036
$O(n^3)$	0,001s	0,008s	0,027s	0,064s	0,125s	0,216s
$O(2^n)$	0,001s	1,048s	17,9min	12,7dni	35,7lat	366w
$O(3^n)$	0,059s	58min	6,5lat	3855w	227*10^6 w	1,3*10^{13} w
$O(n!)$	3,6s	771w	8,4*10^{16} w	2,6*10^{32} w	9,6*10^{47} w	2,6*10^{66} w

(założenie: dla $n = 1$ każdy algorytm wykonuje się 10^{-6} s)