# Programowanie
## *przed 2 wykładem*

Andrzej Giniewicz

08.03.2024

Podczas partii materiału na ten tydzień nauczymy się szukać wartości na liście oraz szukać miejsc zerowych i maksimów funkcji. Uwaga: Wyprowadzenia wzorów ogólnych na czas działania programów nie obowiązują na kartkówce, podobnie jak iteratory, które są tu jedynie wspomniane.

## 1 Sprawdzanie, czy element znajduje się na liście

Najprostszą formą wyszukiwania jest sprawdzenie, czy element znajduje się na liście. Możemy to zrobić na kilka sposobów, na przykład używając wbudowanego operatora `in`

```
element in lista
```

Elementów listy jest potencjalnie wiele, więc nie obejdziemy się bez pętli. Ponieważ musimy sprawdzić, czy element na liście jest równy poszukiwanemu, nie obejdzie się również bez instrukcji warunkowej. Moglibyśmy takie wyszukiwanie zapisać na przykład tak.

```
def szukaj(element, lista):
    for x in lista:
        if x == element:
            return True
    return False
```

Przeanalizujmy ten program. W funkcji znajduje się pętla, która wykonuje fragment kodu dla każdego elementu listy, nazywając go `x` w każdym przejściu pętli. Jeśli element `x` jest równy poszukiwanemu, zwracana jest wartość `True`. Pamiętajmy, że `return` powoduje przerwanie działanie funkcji, więc program przeszukuje listę tak długo, aż znajdzie szukany element (i tylko tak długo). Jeśli przejrzymy całą listę i nie znajdziemy elementu, czyli nie wejdziemy do linii `return True`, pętla się zakończy i w następnej linii wejdziemy do `return False`. Szczególny przypadek brzegowy pustej listy również działa, ponieważ jeśli lista jest pusta, pętla nie wykona się ani razu i zwrócone zostanie `False`, co jest zgodne z oczekiwaniami, ponieważ w pustej liście nie może być żadnego elementu.

## 1.1 Analiza czasu wykonania algorytmu

Choć do wykładu o złożoności obliczeniowej zostało nam trochę czasu, przy większości algorytmów będziemy starali się wyliczać ile czasu komputerowi zajmie wykonanie jakiegoś kodu. Przeanalizujmy teraz kod funkcji `szukaj`.

Oczywiście czas działania programu zależy od wielu czynników. Po pierwsze od tego, czy element rzeczywiście jest na liście czy nie, a po drugie, to jeśli element jest na liście, należy zadać pytanie, na którym miejscu listy się on znajduje. Nie bez znaczenia jest też długość listy. Spróbujmy zidentyfikować trzy przypadki:

1. przypadek optymistyczny,
2. przypadek pesymistyczny,
3. przypadek średni.

Przypadek optymistyczny to taki, w którym program działa najkrócej jak to tylko możliwe. Przypadek pesymistyczny to taki, w którym program działa najdłużej jak to tylko możliwe. Przypadek średni to taki, który przedstawia przeciętny czas działania programu. Zwykle będziemy chcieli poznać przypadek pesymistyczny, aby „przygotować się na najgorsze" oraz przypadek średni, aby przeciętnie wiedzieć, czego się spodziewać.

O ile czas poświęcony na wybieranie kolejnych elementów listy jest zwykle nieduży i może być pominięty, to porównanie elementów niekiedy wymaga sporo pracy — pamiętajmy, że w liście mogą być napisy, przykładowo reprezentujące całe powieści, więc porównywanie ich znak po znaku jest operacją długotrwałą. Aby więc policzyć, jak długo będzie działać funkcja, będziemy liczyć, ile porównań wykona w zależności od długości listy.

Istnieje jedna sytuacja, w której wykonujemy zero porównań i jest to sytuacja, w której nie wchodzimy ani razu do pętli — aby to było możliwe, lista musi być długości zero. Wtedy wykonujemy zero porównań i od razu zwracamy wartość `False`, niezależnie od elementu. Jeśli rozważyć przypadek listy niepustej, to przypadek optymistyczny to jedno porównanie — w sytuacji, gdy szukany element jest na początku listy. Liczbę porównań w obu tych sytuacjach możemy zapisać jednym wzorem $\min\{1, n\}$.

Przypadek pesymistyczny to taki, w którym program działa najdłużej jak to tylko możliwe. Sytuację taką można dość łatwo zidentyfikować — aby program wykonał najwięcej porównań jak się da, elementu nie może być na liście na pierwszych $n - 1$ miejscach. Wtedy, program porówna każdy element listy z szukanym. W takiej sytuacji przy liście długości $n$ wykonamy dokładnie $n$ porównań. Podsumujmy to, co dowiedzieliśmy się do tej pory dla list długości $n$.

| przypadek | liczba porównań |
|---|---|
| optymistyczny | $\min\{1, n\}$ |
| pesymistyczny | $n$ |

Analiza przypadku średniego jest w tym przypadku trudniejsza i wymaga znajomości klasycznego modelu prawdopodobieństwa, takiego jaki poznajemy w szkołach średnich. Aby wyprowadzić wzór, musimy znać prawdopodobieństwo tego, że dana wartość z listy to szukana wartość. Nazwijmy to prawdopodobieństwo $p$ i załóżmy na razie, że $p \in (0, 1)$.

Przypadek $p = 0$ odpowiada już przeanalizowanemu przypadkowi pesymistycznemu (elementu nie ma na liście, więc wykonamy $n$ porównań), natomiast $p = 1$ odpowiada przypadkowi optymistycznemu (lista składa się tylko z szukanego elementu, zatem już po pierwszym porównaniu znajdziemy szukany element). Po wyznaczeniu wzoru dla $p \in (0, 1)$ sprawdzimy, czy da się go uogólnić na przypadek $p \in \{0, 1\}$.

Szansa na to, że wykonamy jedno porównanie to $p$. Szansa na to, że wykonamy dwa porównania, to $(1 - p) \cdot p$, ponieważ raz musi „wypaść" coś innego niż szukana i potem szukana wartość. Szansa na to, że wykonamy trzy porównania, to $(1 - p)^2 \cdot p$. Szansa na to, że wykonamy $n - 1$ porównań, to $(1 - p)^{n-2} \cdot p$. Wzór na $n$ porównań jest inny, ponieważ w nim niezależnie od tego, czy na ostatnim miejscu jest szukana wartość, czy nie, wykonamy $n$ porównań — prawdopodobieństwo tego zdarzenia to zatem $(1 - p)^{n-1}$, czyli $n - 1$ braków zdarzenia. Wypiszmy liczby porównań i prawdopodobieństwa.

| liczba porównań | prawdopodobieństwo |
|---|---|
| 1 | $(1 - p)^0 \cdot p$ |
| 2 | $(1 - p)^1 \cdot p$ |
| … | … |
| $k < n$ | $(1 - p)^{k-1} \cdot p$ |
| … | … |
| $n - 1$ | $(1 - p)^{n-2} \cdot p$ |
| $n$ | $(1 - p)^{n-1}$ |

Policzmy teraz średnią ważoną $l_n$ liczby porównań. Wartość ta będzie przeciętną liczbą porównań wykonaną przez algorytm szukania elementu na liście[1]. Mamy

$$l_n = n(1 - p)^{n-1} + \sum_{k=1}^{n-1} k(1 - p)^{k-1} p = n(1 - p)^{n-1}(1 - p + p) + \sum_{k=1}^{n-1} k(1 - p)^{k-1} p =$$
$$= n(1 - p)^n + \sum_{k=1}^{n} k(1 - p)^{k-1} p.$$

Wyciągając $p$ przed znak sumy mamy

$$l_n = n(1 - p)^n + p \sum_{k=1}^{n} k(1 - p)^{k-1}.$$

Zajmijmy się sumą po prawej stronie

$$\sum_{k=1}^{n} k(1 - p)^{k-1} = \sum_{k=1}^{n} (k - 1 + 1)(1 - p)^{k-1} = \sum_{k=1}^{n} (k - 1)(1 - p)^{k-1} + \sum_{k=1}^{n} (1 - p)^{k-1}.$$

W obu sumach zmienimy indeks tak, aby zaczynał się od $k = 0$

$$\sum_{k=1}^{n} k(1 - p)^{k-1} = \sum_{k=0}^{n-1} k(1 - p)^{k} + \sum_{k=0}^{n-1} (1 - p)^{k}.$$

[1]Po kursie z prawdopodobieństwa moglibyśmy powołać się na wartość oczekiwaną rozkładu geometrycznego, nie jesteśmy jednak jeszcze w tym miejscu — zachęcam, żeby wrócić do analizy niektórych algorytmów po realizacji kursu z prawdopodobieństwa, ponieważ może to rzucić nowe światło zarazem na analizowane algorytmy, jak i na zastosowania poznanych na nim rozkładów.

Zauważmy, że w pierwszej z dwóch sum, pierwszy wyraz to $0(1-p)^0 = 0$, więc możemy sumować od jedynki. Druga suma to suma wyrazów ciągu geometrycznego, możemy zatem zastosować znany wzór na jego sumę

$$\sum_{k=1}^{n} k(1-p)^{k-1} = \sum_{k=1}^{n-1} k(1-p)^{k} + \frac{1-(1-p)^n}{p}.$$

Wyciągnijmy z pozostałej sumy $(1-p)$ przed znak sumowania

$$\sum_{k=1}^{n} k(1-p)^{k-1} = (1-p)\sum_{k=1}^{n-1} k(1-p)^{k-1} + \frac{1-(1-p)^n}{p}$$

i dodajmy i odejmijmy brakujący składnik, tak aby upodobnić sumowania po obu stronach równości

$$\begin{aligned}\sum_{k=1}^{n} k(1-p)^{k-1} &= (1-p)\left[\left(\sum_{k=1}^{n-1} k(1-p)^{k-1}\right) + n(1-p)^{n-1} - n(1-p)^{n-1}\right] + \frac{1-(1-p)^n}{p} = \\ &= (1-p)\left[\left(\sum_{k=1}^{n} k(1-p)^{k-1}\right) - n(1-p)^{n-1}\right] + \frac{1-(1-p)^n}{p}.\end{aligned}$$

Zastąpmy teraz oba wystąpienia sumy przez $x$. Uzyskamy równanie

$$x = (1-p)\left(x - n(1-p)^{n-1}\right) + \frac{1-(1-p)^n}{p}.$$

Rozwiążmy to równanie ze względu na $x$. Mamy

$$x = (1-p)x - n(1-p)^n + \frac{1-(1-p)^n}{p},$$

$$px = \frac{1-(1-p)^n}{p} - n(1-p)^n,$$

$$x = \frac{1-(1-p)^n}{p^2} - \frac{n(1-p)^n}{p},$$

czyli

$$\sum_{k=1}^{n} k(1-p)^{k-1} = \frac{1-(1-p)^n}{p^2} - \frac{n(1-p)^n}{p}.$$

Podstawmy sumę do oryginalnego równania

$$\begin{aligned}l_n &= n(1-p)^n + p\sum_{k=1}^{n} k(1-p)^{k-1} = n(1-p)^n + p\left(\frac{1-(1-p)^n}{p^2} - \frac{n(1-p)^n}{p}\right) = \\ &= \frac{1-(1-p)^n}{p}.\end{aligned}$$

Czy wzór ten może być potraktowany jako wzór ogólny? Musimy sprawdzić przypadki brzegowe, które odpowiadają przypadkowi optymistycznemu (element jest pierwszy, więc $p = 1$) oraz pesymistycznemu (elementu nie ma, więc $p = 0$).

Podstawmy do wzoru $p = 1$, mamy

$$l_n = 1 - 0^n.$$

Stosując umowę $0^0 = 1$, która jest często stosowaną umową w kombinatoryce i również domyślną wartością wyrażenia `0**0` w Pythonie, mamy dwa możliwe rezultaty: $l_n = 0$, gdy $n = 0$ oraz $l_n = 1$, gdy $n > 1$. Pokrywa się to z wcześniejszymi wyliczeniami dla przypadku optymistycznego.

Przypadek pesymistyczny jest nieco trudniejszy. Ponieważ $p$ jest w mianowniku, nie możemy wprost podstawić $p = 0$. Niemniej jednak możemy obliczyć granicę wyrażenia przy $p \to 0$

$$\lim_{p\to 0} \frac{1-(1-p)^n}{p} \overset{H}{=} \lim_{p\to 0} \frac{n(1-p)^{n-1}}{1} = n.$$

Wartość ta ponownie pokrywa się z wyliczoną wcześniej dla przypadku pesymistycznego. Oznacza to, że udało nam się znaleźć wzór na średnią liczbę porównań, jaką musi wykonać algorytm wyszukiwania, zależną od prawdopodobieństwa natrafienia na element $p$ oraz długości listy $n$, jest to

$$l_n = \frac{1-(1-p)^n}{p}.$$

Na przykład, jeśli w liście znajdują się wyniki rzutu kostką i szukamy „szóstki", to $p = \frac{1}{6}$. Jeśli lista jest długości $n = 10$, to średnio wykonamy

$$\frac{1-(1-p)^n}{p} = \frac{1-\left(1-\frac{1}{6}\right)^{10}}{\frac{1}{6}} \approx 5.0309665$$

porównań. Gdybyśmy w tej samej liście szukali wyniku parzystego, to $p = \frac{1}{2}$ i średnio wykonamy

$$\frac{1-(1-p)^n}{p} = \frac{1-\left(1-\frac{1}{2}\right)^{10}}{\frac{1}{2}} \approx 1.998046875$$

porównań.

Zwróćmy uwagę, że oczywiście im mniejsze $p$, tym większy czas wyszukiwania. Dodatkowo im większe $n$, tym większy czas wyszukiwania. Jeśli $n \to \infty$, czas wyszukiwania jest równy $\frac{1}{p}$. Oznacza to, że jeśli prawdopodobieństwo wystąpienia szukanego elementu jest spore, na przykład $\frac{1}{6}$, średnio znajdziemy szukaną wartość w 6 porównaniach, nawet jeśli lista jest ogromna.

## 2 Szukanie elementu największego

Zajmijmy się teraz szukaniem największego elementu listy. Odróżnia się to tym od problemu sprawdzania, czy element jest na liście, że nie wiemy, ile wynosi szukana wartość. Oznacza to, że nigdy nie jesteśmy pewni, czy kolejna wartość w liście nie jest większa. Musimy zatem przejrzeć wszystkie elementy. Zauważmy, że nasz problem jest niezdefiniowany dla pustych list — w takiej sytuacji zwracać będziemy `None`.

```
def maksimum(lista):
    if lista:
        kandydat = lista[0]
        for element in lista:
            if element > kandydat:
                kandydat = element
        return kandydat
```

Przeanalizujmy ten program. Po pierwsze, całość definicji funkcji stanowi jedna instrukcja warunkowa `if lista`. Python wejdzie do zagnieżdżonego kodu, jeśli lista będzie niepusta. Jeśli lista będzie pusta, program nie wejdzie do zagnieżdżonego kodu i tym samym zakończy się definicja funkcji. Przypomnijmy, że jeśli funkcja się skończy bez napotkania na instrukcję `return`, Python sam zwróci za nas wartość `None`, czyli część specyfikacji problemu dotyczącą pustych list mamy z głowy. Załóżmy zatem, że lista jest niepusta i analizujmy dalej.

Skoro lista jest niepusta, to istnieje element `lista[0]` i ustawiamy go jako kandydata do bycia maksimum. Jeśli nie znajdziemy nic lepszego, to właśnie on będzie największym elementem. Po ustaleniu początkowego kandydata przeglądamy całą listę w poszukiwaniu elementu większego od kandydata. Jeśli taki element znajdziemy, staje się naszym nowym kandydatem do bycia największym. Ponieważ relacja większości jest przechodnia, po zmianie kandydata nie musimy go porównywać z wcześniejszymi elementami.

Złożoność zarazem pesymistyczna, jak i optymistyczna, to w tym przypadku $n$ porównań. Nie sposób nie sprawdzić wszystkich elementów, ponieważ zawsze tuż za rogiem może czekać na nas jeszcze większy element, niż się do tej pory spodziewaliśmy.

W programie tym, choć jest on zapisany dość intuicyjnie, jest jeden niewielki problem — zbędne porównanie elementu zerowego z samym sobą. Moglibyśmy to rozwiązać na kilka sposobów, ale mają swoje wady i zalety. To, co przychodzi na myśl jako pierwsze, czyli wykorzystanie wycinka,

```
def maksimum(lista):
    if lista:
        kandydat = lista[0]
        for element in lista[1:]:
            if element > kandydat:
                kandydat = element
        return kandydat
```

jest rozwiązaniem bardzo złym, ponieważ podczas brania wycinka `lista[1:]` tworzona jest kopia listy. To trwa więcej niż jedno porównanie. Inne rozwiązanie to przejście po elementach za pomocą pętli `while`.

```
def maksimum2(lista):
    n = len(lista)
    if n > 0:
        kandydat = lista[0]
        i = 1
```

```
    while i<n:
        if lista[i] > kandydat:
            kandydat = lista[i]
        i += 1
    return kandydat
```

Jest to mniej więcej ten sam kod i w dodatku zaoszczędza nam jednego porównania. Niestety, ku naszemu zaskoczeniu, jeśli zmierzymy czas wykonania dla listy stu losowych wartości całkowitych, okaże się, że wersja z `while` jest do trzech razy wolniejsza. A to dlatego, że pętla `for` korzysta z lepszych optymalizacji — nie musi pamiętać indeksu `i`, zwiększać go o jeden co przejście pętli, dostawać się do $i$-tego elementu, sprawdzać, czy `i` nie jest za duże. Na tym wszystkim wersja z pętlą `for` oszczędza czas.

Na ten moment nie potrafimy jeszcze zapisać optymalnego rozwiązania, ale dla ciekawych świata już teraz wybiegniemy w przyszłość. Jest sposób na pozbycie się zbędnego porównania, wciąż korzystając z pętli `for`. Zachęcam do poczytania o iteratorach i funkcjach `iter` oraz `next`. Na zajęciach zajmiemy się nimi pod koniec semestru. Tymczasem poniżej optymalny kod korzystający z tych właśnie funkcji.

```
def maksimum3(lista):
    if lista:
        lista = iter(lista)
        kandydat = next(lista)
        for element in lista:
            if element > kandydat:
                kandydat = element
        return kandydat
```

W wielkim skrócie — iterator pozwala przejść po liście raz. Każdy kolejny element pobieramy funkcją `next`. Ponieważ pętla `for` rozumie iteratory, kontynuuje pobieranie elementów listy od drugiego, po tym, jak my pobraliśmy pierwszy z nich i podstawiliśmy pod kandydata. Jeśli zmierzymy czas wykonania tego programu, okaże się, że jest około 10% szybszy niż pierwsza wersja.

# 3 Miejsce zerowe funkcji

Zajmijmy się teraz problemem zupełnie innym, choć wciąż polegającym na poszukiwaniu. Tym razem nie poszukujemy jednak elementów listy, tylko miejsc zerowych funkcji.

Zademonstrujemy jeden spośród znanych algorytmów, będzie to algorytm bisekcji. Aby znaleźć oszacowanie miejsca zerowego funkcji $f$ na przedziale $[a, b]$, musi ona spełniać następujące warunki:

1. $f$ musi być funkcją ciągłą na $[a, b]$,
2. $f$ musi mieć przeciwne znaki na krańcach $[a, b]$.

Warunki te odpowiadają założeniom twierdzenia Darboux, jednego z twierdzeń analizy matematycznej, które gwarantuje nam, że funkcja $f$ ma przynajmniej jedno miejsce zerowe we wnętrzu przedziału $(a, b)$. Ponieważ mamy gwarancję matematyczną istnienia choć jednego miejsca zerowego, możemy go poszukać za pomocą Pythona.

Pomysł na algorytm jest następujący — skoro wiemy, że miejsce zerowe jest w przedziale $(a, b)$, przekrójmy ten przedział na mniejsze części:

$$\left(a, \frac{a+b}{2}\right), \left\{\frac{a+b}{2}\right\}, \left(\frac{a+b}{2}, b\right).$$

Miejsce zerowe będzie zatem w jednym z trzech wskazanych zbiorów, w lewym przedziale o połowie długości, na środku lub w prawym przedziale o połowie długości. Jeśli okaże się, że miejsce zerowe jest na środku, mamy rozwiązanie. Jeśli nie, to będzie w jednym z mniejszych podprzedziałów — spróbujemy więc tą samą metodą poszukać go w jednym z nich. Każdy krok algorytmu zawęzi przedział o połowę. Uznamy, że znaleźliśmy miejsce zerowe, jeśli w nie trafimy, lub przedział zawęzi się tak bardzo, że jego środek będzie dostatecznie przybliżał miejsce zerowe. Zobaczmy na następujący kod, który zakłada, że funkcja $f$ spełnia postawione warunki. Odpowiedzialnością użytkownika jest nie używać funkcji z niewłaściwymi parametrami.

```
def bisekcja(f, a, b, tolerancja=1e-6):
    c = (a+b)/2
    pół_długości = (b-a)/2
    if pół_długości <= tolerancja:
        return c
    f_a = f(a)
    while pół_długości > tolerancja:
        f_c = f(c)
        if f_a*f_c < 0:
            b = c
        elif f_a*f_c > 0:
            a = c
            f_a = f_c
        else:
            return c
        pół_długości /= 2
        c = (a+b)/2
    return c
```

Program ten stara się za wszelką cenę jak najrzadziej liczyć funkcję $f$ w punkcie, ponieważ technicznie to może być coś, co trwa bardzo bardzo długo, na przykład może być to wynik całkowania numerycznego lub innej operacji wymagającej pętli i skomplikowanych obliczeń. Nie jest to najprostszy możliwy zapis bisekcji, ale taki, który gwarantuje najmniejszą możliwą liczbę koniecznych wyliczeń funkcji $f$. Przeanalizujmy go krok po kroku.

Na początku program wylicza środek przedziału $(a, b)$ i nazywa go $c$. Jeśli długość przedziału $(a, b)$ równa $b - a$ jest mniejsza niż $2\varepsilon$, gdzie $\varepsilon$ to zadana tolerancja, mamy $(c - \varepsilon, c + \varepsilon) \supseteq (a, b)$ i ponieważ miejsce zerowe $x_0 \in (a, b)$ to również $x_0 \in (c - \varepsilon, c + \varepsilon)$ czyli $|x_0 - c| < \varepsilon$. Oznacza to, że odległość od $c$ do miejsca zerowego jest mniejsza od zadanej tolerancji. Tym samym, $c$ jest oszacowaniem miejsca zerowego i możemy zakończyć proces poszukiwania. Aby ułatwić obliczenia, definiujemy zmienną pomocniczą zawierającą połowę długości przedziału, którą porównujemy z tolerancją. Do tej pory ani razu nie musieliśmy policzyć wartości funkcji!

W kolejnym kroku liczymy wartość funkcji $f(a)$ i zapamiętujemy wynik na potem, aby nie liczyć go dwukrotnie. Zaczynamy pętlę `while`, która wykona się przynajmniej raz (ponieważ przypadek zbyt wąskiego przedziału obsłużyliśmy osobno przed policzeniem wartości $f(a)$). Pętla `while` zakończy się, gdy przedział będzie na tyle wąski, aby środek przedziału dobrze przybliżał miejsce zerowe, podobnie jak przeanalizowaliśmy to na początku w instrukcji warunkowej. Wewnątrz pętli obliczamy wartość funkcji $f(c)$ i podobnie jak wcześniej wartość funkcji $f(a)$, zapamiętujemy wynik. Interesuje nas znak iloczynu na krańcach przedziału $[a, c]$. Są trzy możliwości:

1. $f(a) \cdot f(c) < 0$, co oznacza, że miejsce zerowe jest w przedziale $(a, c)$,
2. $f(a) \cdot f(c) > 0$, co oznacza, że miejsce zerowe jest w przedziale $(c, b)$,
3. $f(a) \cdot f(c) = 0$, co oznacza, że miejsce zerowe to $c$.

W pierwszym przypadku przesuwamy wartość $b$ na $c$, w drugim przypadku przesuwamy wartość $a$ na $c$. Po tych operacjach przedział jest zaktualizowany, ale gdy „ruszyliśmy" punkt $a$, musimy zaktualizować wartość $f(a)$. Na szczęście mamy wartość $f(c)$, która jest tą liczbą, której szukamy. Dzięki temu nie musimy jej liczyć ponownie! Na końcu pętli wyliczamy nowy środek przedziału, aby na wypadek, gdyby warunek pętli `while` ją przerwał, wartość `c` była aktualna. Zmieniamy też zmienną zawierającą połowę długości, dzieląc ją przez 2. Przeliczenie nowej wartości funkcji $f(c)$ następuje na początku pętli, a nie na jej końcu, ponieważ jeśli przerwiemy pętlę warunkiem o tolerancji, nie ma konieczności liczyć $f(c)$. Zachęcam do prześledzenia kodu na `https://tinyurl.com/metoda-bisekcji-pi`, który wykorzystuje zdefiniowaną tu funkcję `bisect` do obliczenia liczby $\pi$, jako podwojone miejsce zerowe funkcji cos na przedziale $(0, 4)$.

## 3.1 Analiza czasu wykonania algorytmu

Przeanalizujmy czas działania funkcji `bisect`. Zajmiemy się głównie przypadkiem pesymistycznym, ponieważ chcemy mieć gwarancję, że algorytm znajdzie rozwiązanie w skończonym czasie, niezależnie od okoliczności. Ponieważ najbardziej kosztowną operacją jest wyliczanie funkcji, policzymy, ile razy wywołuje się funkcja $f$.

Patrząc na to, ile razy liczy się funkcja $f$, możemy zauważyć, że jest to 1 dodać liczba wywołań pętli `while`.

Wiedząc, że w każdym kroku algorytmu długość przedziału zmniejsza się o połowę, po $k$ krokach pętli długość przedziału wynosić będzie

$$\frac{1}{2^k}(b-a)$$

natomiast połowa długości

$$\frac{1}{2^{k+1}}(b-a)$$

Warunek stopu mówi, że połowa długości przedziału ma być mniejsza lub równa tolerancji, czyli

$$\frac{1}{2^{k+1}}(b-a) \leqslant \varepsilon.$$

Przekształcając, uzyskujemy

$$\frac{b-a}{\varepsilon} \leqslant 2^{k+1},$$

co po zlogarytmowaniu przyjmuje postać

$$\log_2\left(\frac{b-a}{\varepsilon}\right) \leqslant k+1,$$

Ponieważ program przerywamy przy pierwszym $k$ naturalnym spełniającym tę nierówność, sprowadza się to do równania

$$\left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil = k+1$$

i tym samym ostatecznej liczby kroków pętli

$$k = \left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil - 1.$$

Ponieważ jak zauważyliśmy liczba wyliczeń funkcji $f$ to $k+1$, mamy ostatecznie

$$\left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil$$

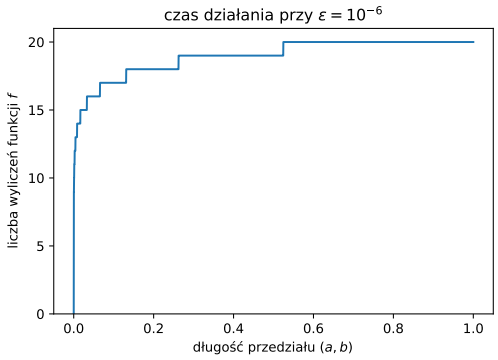
razy wyliczoną wartość funkcji $f$.

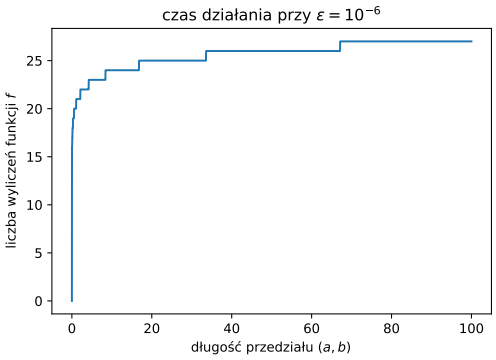
Aby uwzględnić szczególny przypadek wąskiego przedziału początkowego, w którym instrukcja warunkowa szybciej przerywa program przed policzeniem pierwszego wystąpienia funkcji $f$, możemy zapisać liczbę wywołać funkcji $f$ jako

$$\begin{cases} 0, & \text{gdy } \frac{b-a}{\varepsilon} \leqslant 2, \\ \left\lceil \log_2\left(\frac{b-a}{\varepsilon}\right)\right\rceil, & \text{gdy } \frac{b-a}{\varepsilon} > 2. \end{cases}$$

Zauważmy, że czas wykonania algorytmu zależy jedynie od wartości $\frac{b-a}{\varepsilon}$, a w przypadku ustalonej tolerancji, zależy jedynie od szerokości przedziału $(a,b)$.

Funkcja logarytmiczna rośnie bardzo powoli, dlatego liczba kroków algorytmu jest niewielka. Dla precyzji $\varepsilon = 10^{-6}$, co zostało wybrane jako wartość domyślna w implementacji, liczba kroków dla różnych szerokości przedziału została przedstawiona na następujących ilustracjach.




Musimy pamiętać, że choć metoda zatrzyma się po wskazanym czasie i liczbie kroków, zwracając wynik, jeśli założenia o funkcji $f$ nie są spełnione, wynik nie musi być miejscem zerowym, ale może być zupełnie inną wartością. Zawsze pamiętajmy o sprawdzaniu założeń!

# 4 Maksimum funkcji

Przejdźmy do badania maksimum funkcji. Podobnie jak przy bisekcji, będziemy potrzebować dodatkowych założeń:

1. funkcja $f$ jest ciągła na przedziale $[a, b]$,
2. funkcja $f$ jest silnie jednomodalna na przedziale $[a, b]$.

Mówimy, że funkcja jest silnie jednomodalna, jeśli ma na tym przedziale jedno maksimum lokalne, na lewo od niego jest silnie rosnąca a na prawo od niego silnie malejąca.

Rozważmy przecięcie przedziału $[a, b]$ na trzy przedziały

$$[a, b] = [a, x] \cup [x, y] \cup [y, b],$$

dla pewnych $a < x < y < b$. Rozważmy trzy sytuacje

1. maksimum jest w $[a, x]$, wtedy $f(x) > f(y) > f(b)$,
2. maksimum jest w $[x, y]$, wtedy $f(a) < f(x)$ i $f(y) > f(b)$,
3. maksimum jest w $[y, b]$, wtedy $f(a) < f(x) < f(y)$.

Odwracając te zależności otrzymujemy:

1. Jeśli $f(x) < f(y)$, to maksimum leży w przedziale $[x, b]$,
2. Jeśli $f(x) = f(y)$, to maksimum leży w przedziale $[x, y]$,
3. Jeśli $f(x) > f(y)$, to maksimum leży w przedziale $[a, y]$.

Często dla uproszczenia rozważa się prostszą regułę

1. Jeśli $f(x) < f(y)$, to maksimum leży w przedziale $[x, b]$,

2. Jeśli $f(x) \geqslant f(y)$, to maksimum leży w przedziale $[a, y]$.

Wyznaczmy teraz takie punkty $x$ i $y$ wewnątrz $[a, b]$, aby po dokonaniu podziału i poszukiwaniu maksimum w podprzedziale $[x, b]$ lub $[a, y]$, można było wykorzystać jedną z już obliczonych wartości:

1. jeśli maksimum jest w przedziale $[a, y]$, mamy jedną wartość $x \in [a, y]$ którą możemy ponownie wykorzystać,
2. jeśli maksimum jest w przedziale $[x, b]$, mamy jedną wartość $y \in [x, b]$ którą możemy ponownie wykorzystać.

Rozważmy pierwszą sytuację, czyli $x \in [a, y]$. Niestety punkt ten nie może pozostać lewym krańcem nowego podziału, ale jeśli położymy $y' = x$, będziemy mogli znaleźć nowy podział $a < x' < y' < y$ o takich samych własnościach, co $a < x < y < b$. Zauważmy z proporcji, że

$$c = \frac{y-a}{b-a} = \frac{y'-a}{y-a},$$

czyli

$$c = \frac{y-a}{b-a} = \frac{x-a}{y-a},$$

Z drugiej strony z symetrii

$$c = \frac{b-x}{b-a},$$

stąd

$$1 - c = \frac{b-a}{b-a} - \frac{b-x}{b-a} = \frac{x-a}{b-a} = \frac{x-a}{y-a} \cdot \frac{y-a}{b-a} = c \cdot c = c^2.$$

Otrzymujemy równanie

$$c^2 + c - 1 = 0,$$

które rozwiążemy, sprowadzając do postaci kanonicznej. Otrzymujemy

$$\left(c + \frac{1}{2}\right)^2 = \frac{5}{4},$$

czyli

$$c + \frac{1}{2} = \pm\frac{\sqrt{5}}{2}$$

i ostatecznie

$$c = \frac{\pm\sqrt{5} - 1}{2}.$$

Jedna z tych wartości jest ujemna, więc nie może być rozwiązaniem równania, w którym $c$ było ilorazem długości. Wykluczamy to rozwiązanie i pozostaje

$$c = \frac{\sqrt{5}-1}{2} = \frac{1}{\frac{2}{\sqrt{5}-1}} = \frac{1}{\frac{\sqrt{5}+1}{2}}.$$

Stosując oznaczenie

$$\varphi = \frac{\sqrt{5}+1}{2},$$

otrzymujemy $c = \frac{1}{\varphi}$. Dlaczego wyrażamy $c$ za pomocą $\varphi$? $\varphi$ jest znaną liczbą, tak zwaną złotą liczbą — możemy ją znaleźć w wielu miejscach w przyrodzie i architekturze, a nawet w proporcjach marginesów w średniowiecznych manuskryptach.

Wobec tego, dla przedziału $[a, b]$, wyliczamy punkt $y$ na podstawie równania

$$\frac{1}{\varphi} = \frac{y - a}{b - a},$$

czyli

$$y = a + \frac{b - a}{\varphi}.$$

Z symetrii wyprowadzić można również punkt $x$ jako

$$x = b - \frac{b - a}{\varphi}.$$

Tak wybrane punkty wewnętrzne gwarantują nam to, że będziemy mogli wykorzystać ponownie jeden z punktów wewnętrznych podczas szukania maksimum. Ponieważ w metodzie tej pojawia się złota liczba, metodę tą nazywamy metodą złotego podziału.