Algorytmy i Struktury Danych

(1) Poprawność Algorytmów

(c) Marcin Sydow

Kontakt

dr hab. Marcin Sydow,

Katedra SIAM, PJATK

Polecane Podręczniki:

Ogólne:

- **T.Cormen, C.Leiserson, R.Rivest et al. "Wprowadzenie do Algorytmów", PWN 2018** (lub wydanie anglojęzyczne)
- G.Mirkowska et al. "Algorytmy i Struktury Danych - Zadania", wydawnictwo PJWSTK, 2005 (zbiór zadań i ćwiczeń, częściowo z rozwiązaniami)
- L.Banachowski, K.Diks, W.Rytter "Algorytmy i Struktury Danych", PWN 2018, (ok. 300 stron), zwięzła książeczka, trudniejsza dla początkujących
- "Algorithms and Datastructures. The Basic Toolbox" (MS), K.Mehlhorn P.Sanders, Springer 2008

Przykłady innych podręczników

- N.Wirth "Algorytmy + Struktury Danych = Programy"
- A.Aho, J.Hopcroft, J.Ullman "Algorytmy i Struktury Danych"
- W.Lipski "Kombinatoryka dla Programistów", WNT 2004

Do pogłębionych studiów:

- D.Knuth "The Art of Computer Programming" (3 tomy)
- Ch.Papadimitriou "Computational Complexity"

Algorytm

Co oznacza wyraz “algorytm”?

Algorytm

Co oznacza wyraz "algorytm"?

Dokładny opis, przepis, np. w postaci listy kolejnych kroków, jak coś wykonać, etc. (nie tylko zadania obliczeniowe, np. przepis wykonania pewnej potrawy albo dokonania pewnej procedury prawnej, etc.)

Wg historyków, słowo to wywodzone jest od arabskiej wersji nazwiska wybitnego matematyka perskiego *al-Khwarizmi* (A.D. 780-850)

Algorytmika stanowi serce informatyki.
Rola algorytmiki nawet rośnie w czasach ogromnego przyrostu danych ("big data").

Jeden poziom abstrakcji ponad programowaniem

Do zapisu algorytmów używany jest tzw. "pseudokod", niebędący żadnym konkretnym językiem programowania, ale podobny do współcześnie używanych, popularnych języków programowania (np. Java, C/C++, Pascal, Python, etc.)

Pseudokod

- abstrakcyjna notacja algorytmów
- wygląda podobnie do popularnych języków (Java, C/C++, Pascal, Python)
- rola bardziej informacyjna niż formalna (możliwie rozluźniony formalizm, o ile nie prowadzi do niejednoznaczności)
  - literały (liczby, znaki, łańcuchy, NULL)
  - zmienne (bez konieczności uprzedniej deklaracji)
  - tablice i operator indeksowania "[ ]" (zakładamy indeksowanie tablic od 0)
  - operatory (przypisanie '=', relacyjne "==,<,>", logiczne "&, ||,!", matematyczne "+,++,+=", etc.
  - funkcje (w tym rekursja), instrukcja `return` (zakładamy przesyłanie argumentu dla typów złożonych przez referencję)
  - konstrukcje sterujące: wyrażenia warunkowe (IF, THEN, ELSE), pętle (WHILE, FOR).

Przykład użycia pseudokodu

Zadanie: oblicz sumę liczb w tablicy o długości `len`:

```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

Obserwacja: nie jest to żaden konkretny język programowania, ale każdy współczesny programista dokładnie rozumie co się w tym kodzie dzieje.

O czym jest kurs "Algorytmy i Struktury Danych"?

Zasadniczo składa się z 3 komponentów, które częściowo się nakładają:

1. Analiza algorytmów (dany jest kod, należy zrozumieć co robi i jak efektywnie to robi)
2. Projektowanie algorytmów (dana jest specyfikacja zadania obliczeniowego, należy dokładnie zaprojektować poprawny i efektywny algorytm je rozwiązujący)
3. Struktury Danych (dotyczy efektywnej organizacji danych i operacji wykonywanych na nich)

Projektowanie algorytmów

Dane jest dobrze wyspecyfikowane zadanie obliczeniowe do wykonania na komputerze (np. obliczenie sumy liczb w tablicy)

Zanim rozpoczęta zostanie **implementacja**, należy **zaprojektować algorytm**, czyli "sposób" rozwiązania problemu. Zaprojektowany algorytm zapisać można za pomocą pseudokodu lub tzw. schematu blokowego, zanim zostanie zaimplementowany w konkretnym języku programowania.

Po zaprojektowaniu algorytmu i dokonaniu jego **analizy** sprawdzającej poprawność i efektywność można przystąpić do implementacji.

Projekt i analiza algorytmu to niezbędne kroki przed przystąpieniem do implementacji.

Specyfikacja algorytmu

Jest to kluczowe pojęcie, które oznacza w sposób nieco bardziej formalny: "co dokładnie algorytm ma zrobić".

Definicja: Specyfikacja algorytmu wyraża kontrakt algorytmu i składa się z następujących elementów:

- (opcjonalnie) **nazwa** algorytmu i następująca po niej **lista argumentów** w nawiasach
- (wejście) tzw. **warunek początkowy**, który dokładnie specyfikuje typy i dopuszczalne wartości **poprawnych danych wejściowych**
- (wyjście) tzw. **warunek końcowy**, który dokładnie specyfikuje prawidłowy **wynik** (typ i wartość(i)) jaki ma być zwrócony przez algorytm jako funkcja danych wejściowych

Warunki te mogą być sformułowane w języku naturalnym, o ile jest to sformułowanie ścisłe.

Przykład specyfikacji

Mamy dane potocznie sformułowane zadanie: "zwróć sumę całkowitą liczb całkowitych w podanej tablicy o podanej długości"

Stwórzmy na jego podstawie bardziej precyzyjną **specyfikację**:

- **nazwa i argumenty**: `sum(sequence, len)`
- **warunek początkowy** (wejście): `sequence` - tablica liczb całkowitych, `len` - liczba naturalna będąca deklarowaną długością tablicy
- **warunek końcowy** (wyjście): algorytm ma zwrócić liczbę całkowitą będącą sumą pierwszych `len` elementów tej tablicy lub zero jeśli jest pusta.

Obserwacje: ponieważ zdecydowaliśmy, że długość jest liczbą naturalną, więc tablica może mieć poprawnie długość 0 (pusta). Należy więc zdecydować, jaką wartość zwrócić w takim specjalnym przypadku.

Zauważmy, że gdyby długość tablicy musiała być dodatnia, nie trzeba by zajmować się w specyfikacji przypadkiem tablicy pustej (ale tak algorytm jest ogólniejszy)

Całkowita poprawność algorytmu

poprawne dane wejściowe to takie dane, które spełniają **warunek początkowy** specyfikacji

poprawny wynik algorytmu to taki, który spełnia **warunek końcowy** specyfikacji

Definition

Przy danej specyfikacji, algorytm jest całkowicie poprawny wtedy i tylko wtedy, gdy dla każdych poprawnych danych wejściowych zachodzą oba poniższe warunki:

1. algorytm zatrzymuje się po skończonej liczbie kroków (tzw. **własność stopu**)
2. algorytm przy zatrzymaniu zwraca **poprawny wynik** (tzw. **częściowa poprawność algorytmu**)

Zauważmy celowy podział na dwie, niezależne części powyżej.

Częściowa poprawność algorytmu

Wg poprzedniej definicji, sprawdzanie czy algorytm jest całkowicie poprawny w praktyce dzielone jest na 2 części:

1. sprawdzanie czy algorytm ma własność stopu
2. sprawdzanie (przy założeniu, że się zatrzymuje) czy zwraca poprawny wynik

Druga własność powyżej, nazywa się częściową poprawnością algorytmu

Definition

Algorytm jest **częściowo poprawny** jeśli spełnia poniższy warunek (w formie implikacji):
Jeżeli algorytm **zatrzyma się** (i otrzymał poprawne dane wejściowe) **to** zwracany jest **poprawny wynik**

Obserwacja: częściowa poprawność nie gwarantuje zatrzymania się algorytmu

Przykład częściowo poprawnego algorytmu

(zadanie obliczania sumy liczb w tablicy)
(array - tablica liczb całkowitych, len liczba naturalna (może być 0))
(uwaga: pseudokod może zawierać celową usterkę, aby ćwiczenie miało sens)

```
sum(array, len){
  sum = 0
  i = 0
  while(i < len)
    sum += array[i]
  return sum
}
```

Czy powyższy algorytm:

- ma własność stopu?
- kiedykolwiek zatrzymuje się dla jakichś poprawnych danych?
- jeśli się zatrzymuje, to czy zwraca poprawny wynik?
- a więc jest częściowo poprawny?
- czy jest całkowicie poprawny?

Jest to więc (nieco sztuczny) przykład ilustrujący fakt, że algorytm może być częściowo poprawny, ale nie całkowicie poprawny

Sprawdzanie własności stopu: przykład


```
sum(array, len){
  sum = 0
  i = 0
  while(i < len){
    sum += array[i]
    i++
  }
  return sum
}
```

Jak udowodnić, że powyższy algorytm ma własność stopu?
Wystarczy zaobserwować, że:

1. algorytm zatrzyma się, kiedykolwiek zajdzie `i >= len`
2. `len` jest **stałą i skończoną** liczbą naturalną
3. wartość zmiennej `i` rośnie o 1 w każdej iteracji

A więc, po skończonej liczbie iteracji algorytm zatrzyma się

Zwróćmy uwagę na istotność wszystkich detali: np. nie wystarczy sam wzrost zmiennej, ale wystarczy wzrost o stałą wartość, albo nie wystarczy, że len jest stałe, ale także, że jest skończone (liczba naturalna), etc.

Dowodzenie częściowej poprawności - niezmienniki

Dowodzenie własności stopu (pierwszy krok całkowitej poprawności) jest na ogół stosunkowo proste.
Natomiast dowodzenie częściowej poprawności (drugi krok całkowitej poprawności) jest na ogół trudniejsze, często wymaga pewnej inwencji i zastosowania specjalnego narzędzia: **niezmiennika pętli**.

Obserwacja: większość nietrywialnej aktywności algorytmów odbywa się wewnątrz pętli. **niezmiennik pętli** jest narzędziem pozwalającym udowodnić poprawność częściową algorytmu zawierającego pętlę.

Definition

Niezmiennik pętli to logiczny predykat spełniający następujący warunek:
jeśli predykat jest spełniony **przed** wejściem w pewną (dowolną) iterację pętli **to** jest także spełniony **po wyjściu z tej iteracji** pętli.

Niezmienniki a Indukcja Matematyczna

Można zauważyć analogię definicji niezmiennika pętli do tzw **kroku indukcyjnego** używanego w dowodach przez *indukcję matematyczną*.
Idea jest następująca: Tworzymy niezmiennik tak, aby w momencie zakończenia działania algorytmu był równoważny z warunkiem końcowym specyfikacji

- jeśli predykat jest spełniony tuż przed pierwszą iteracją pętli (analogia do kroku bazowego w indukcji matematycznej)
- oraz jeśli dodatkowo jest niezmiennikiem, czyli po wejściu w pętlę, zostanie zachowany przez dowolną skończoną liczbę pojedynczych iteracji

to wtedy jest też spełniony na końcu algorytmu (po wyjściu z pętli), niezależnie od tego ile było iteracji algorytmu (co może być zmienne i zależeć od konkretnych danych wejściowych)
Jest to analogia do dowodu przez indukcję matematyczną

Przykład użycia niezmiennika

Zadanie: mając dany pseudokod algorytmu, zgadnąć co oblicza i **udowodnić jego całkowitą poprawność**

wejście: Arr - tablica liczb całkowitych, len > 0 - jej długość

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Co zwraca ten algorytm?

Przykład użycia niezmiennika

Zadanie: mając dany pseudokod algorytmu, zgadnąć co oblicza i **udowodnić jego całkowitą poprawność**

wejście: Arr - tablica liczb całkowitych, len $>$ 0 - jej długość

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Co zwraca ten algorytm?
odpowiedź: maksimum z `len` pierwszych liczb zawartych w tablicy Arr

Przykład użycia niezmiennika

Zadanie: mając dany pseudokod algorytmu, zgadnąć co oblicza i **udowodnić jego całkowitą poprawność**

wejście: Arr - tablica liczb całkowitych, len > 0 - jej długość

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Co zwraca ten algorytm?
odpowiedź: maksimum z `len` pierwszych liczb zawartych w tablicy Arr

Należy teraz **udowodnić** całkowitą poprawność algorytmu.

Przykład, c.d.

Aby udowodnić całkowitą poprawność, wykonujemy standardowe 2 kroki (jakie?):

Przykład, c.d.

Aby udowodnić całkowitą poprawność, wykonujemy standardowe 2 kroki (jakie?):

1. dowód własności stopu
2. dowód częściowej poprawności (przy użyciu niezmiennika pętli)

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

(dowód własności stopu jest podobny jak w poprzednim przykładzie)

Przykład, c.d.

Aby udowodnić całkowitą poprawność, wykonujemy standardowe 2 kroki (jakie?):

1. dowód własności stopu
2. dowód częściowej poprawności (przy użyciu niezmiennika pętli)

```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

(dowód własności stopu jest podobny jak w poprzednim przykładzie)

Natomiast dowód częściowej poprawności wymaga więcej inwencji i użycia niezmiennika pętli.

Przykład, c.d. - dowód częściowej poprawności

Zilustrowane zostanie użycie niezmiennika pętli do dowodu częściowej poprawności algorytmu.
Znalezienie użytecznego niezmiennika pętli wymaga pewnej inwencji.
Pomocna może być następująca technika:

1. zapisać predykat wyrażający warunek końcowy
2. przekształcić warunek końcowy tak, aby:
   - zawierał wszystkie istotne w algorytmie zmienne
   - wyrażał bieżącą (w bieżącej iteracji) wartość zmiennej zwracanej przez algorytm
   - spełniał definicję niezmiennika pętli.
3. sprawdzić czy zaproponowany niezmiennik jest także spełniony tuż przed wejściem w pierwszą iterację pętli

Przykład dowodu częściowej poprawności, c.d.


```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Zadanie: pokazać, że zwracana zmienna x reprezentuje maksimum w tablicy Arr

Przykład dowodu częściowej poprawności, c.d.


```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Zadanie: pokazać, że zwracana zmienna x reprezentuje maksimum w tablicy Arr
Warunek końcowy: x jest niemniejsze niż dowolna liczba w tablicy Arr i x jest zawarte w Arr.
W notacji matematycznej byłoby to zapisane następująco:

Przykład dowodu częściowej poprawności, c.d.


```
algor1(Arr, len){
  i = 1
  x = Arr[0]
  while(i < len)
    if(Arr[i] > x){
      x = Arr[i]
    }
    i++
  return x
}
```

Zadanie: pokazać, że zwracana zmienna x reprezentuje maksimum w tablicy Arr
Warunek końcowy: x jest niemniejsze niż dowolna liczba w tablicy Arr i x jest zawarte w Arr.
W notacji matematycznej byłoby to zapisane następująco:
$(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

Przykład, c.d.

Mamy więc warunek końcowy:
$(\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

Teraz spróbujmy przekształcić powyższy warunek końcowy w niezmiennik. Brakuje zmiennej i (licznik iteracji).
Niezmiennik powinien wyrażać: "w i-tej iteracji x stanowi maksimum pośród i pierwszych wartości w tablicy Arr"
W zapisie matematycznym powyższy predykat wyglądałby tak:
$\forall_{0 \leq j < i}\, x \geq Arr[j] \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$
Zauważmy, że powyższy predykat **jest niezmiennikiem** (jeśli był prawdziwy w iteracji i to będzie prawdziwy po iteracji i, ze względu na warunkową aktualizację zmiennej x w instrukcji "if")
Dodatkowo, powyższy niezmiennik jest też prawdziwy tuż przed pierwszą iteracją pętli (bo $i == 1$ oraz $x = Arr[0]$) A więc będzie też prawdziwy po dowolnej liczbie iteracji. W szczególności, gdy algorytm zatrzyma się (dla i==len): $((\forall_{0 \leq j < i}\, x \geq Arr[j]) \wedge (i == len))$
przyjmie on szczególną postać, z której wynika warunek końcowy.
$\Rightarrow (\forall_{0 \leq j < len}\, x \geq Arr[j]) \wedge (\exists_{0 \leq j < len}(x == Arr[j]))$

Co na pewno należy umieć/wiedzieć po tym wykładzie:

1. Umieć podać z pamięci dokładne definicje:
   - specyfikacji algorytmu
   - poprawnych danych wejściowych i wyjściowych
   - całkowitej poprawności algorytmu
   - częściowej poprawności algorytmu
   - niezmiennika pętli
2. mając dane zadanie obliczeniowe stwórz ścisłą specyfikację
3. podać przykład algorytmu częściowo poprawnego, ale bez własności stopu i odwrotnie
4. umieć udowodnić własność stopu podanego algorytmu
5. umieć znaleźć niezmiennik dla danej prostej pętli
6. umieć udowodnić, że predykat jest niezmiennikiem
7. przy użyciu niezmiennika umieć udowodnić częściową poprawność algorytmu

Dziękuję za uwagę