Wstęp do informatyki

Wykład 7

Uniwersytet Wrocławski
Instytut Informatyki

Temat wykładu

Poprawność programów
- co oznacza?
- jak ją uzasadnić? sprawdzić?
- przykłady: potęgowanie, selekcja, bąbelki,…

Nowe narzędzia do sortowania i ich analiza:
- scalanie
- podział

Jak sprawdzić poprawność programu?

Co oznacza „poprawność”?

- zgodność ze specyfikacją… (?)

Sposoby weryfikacji:

- testowanie? zawsze czegoś nie da się przewidzieć…
- formalna analiza - dowód?

  żmudne, często trudniejsze od napisania programu
- automatyczne narzędzia dowodzenia?

  możliwe tylko w niektórych przypadkach (uniwersalne narzędzia nie istnieją i nie jest możliwe ich stworzenie!)

Zgodność ze specyfikacją…

Stan obliczeń = wartości zmiennych.

Specyfikacja formalna:

- warunek początkowy A
- program P
- warunek końcowy B

gdzie A, B to formuły logiczne, a zmienne programu to zmienne wolne w A, B

Zgodność ze specyfikacją…

UWAGA!

Dla uproszczenia:

- ignorujemy instrukcje wejścia/wyjścia
- analizujemy tylko

  stan obliczeń = wartości zmiennych
- ignorujemy kwestie typów zmiennych, zakresów, deklaracji, …

Zgodność ze specyfikacją…

Program P jest częściowo poprawny ze względu na specyfikację

{A} P {B}

gdy spełniona jest zależność:

jeśli przed wykonaniem P stan obliczeń spełnia A, to po wykonaniu P stan obliczeń spełnia B.

Mówimy wtedy skrótowo, że zachodzi:

{A} P {B}

({A} P {B} to trójka Hoare'a)

Częściowa poprawność

Zachodzi

$$\{x>0\}\ y \leftarrow x+1\ \{y>1\}$$

gdyż:

jeśli przed wykonaniem P= $y \leftarrow x+1$ stan obliczeń spełnia $\{x>0\}$, to po wykonaniu P stan obliczeń spełnia $\{y>1\}$.

UZASADNIENIE

Po wykonaniu obliczeń mamy: $y=x+1>0+1=1$

Częściowa poprawność

Zachodzi

{true} if (a>b) c $\leftarrow$ a; else c $\leftarrow$ b {c$\geq$a oraz c$\geq$b}

gdyż po wykonaniu P= if (a>b) c$\leftarrow$a; else c$\leftarrow$b stan obliczeń spełnia {c$\geq$a oraz c$\geq$b} niezależnie od stanu obliczeń przed P.

UZASADNIENIE

Jeśli a>b, to c jest równe a po wykonaniu P, więc c=a>b.

Analogicznie gdy a$\leq$b.

Częściowa poprawność

Zachodzi

$$\{x>0\}\ y \leftarrow x+1;\ z \leftarrow y+1\ \{z>2\}$$

gdyż:

jeśli przed P= $y \leftarrow x+1;\ z \leftarrow y+1$ stan obliczeń spełnia $\{x>0\}$, to po wykonaniu P stan obliczeń spełnia $\{z>2\}$.

UZASADNIENIE

- Po wykonaniu $y \leftarrow x+1$ zachodzi $y=x+1>0+1=1$
- Po wykonaniu $z \leftarrow y+1$ zachodzi $z=y+1>1+1=2$

Częściowa poprawność

Zachodzi

$$\{x \neq 0\} \text{ while } (x!=0)\ x \leftarrow x-1\ \{x=0\}$$

gdyż:

jeśli przed P= while (x!=0) $x \leftarrow x-1$ stan obliczeń spełnia $\{x \neq 0\}$, to po wykonaniu P stan obliczeń spełnia $\{x=0\}$.

UZASADNIENIE

- Po wyjściu z pętli while nie jest spełniony warunek $x \neq 0$, zatem jest spełniony warunek $x=0$

Częściowa poprawność

Zachodzi

$$\{x \neq 0\}\ \text{while } (x!=0)\ x \leftarrow x-1\ \{x=0\}$$

lecz powyższe oznacza jedynie częściową poprawność:

- jeśli P=while (x!=0) $x \leftarrow x-1$ zakończy działanie, to $\{x=0\}$
- **nie** pokazaliśmy: P **zawsze** kończy działanie!

W naszym przykładzie:

dla $x<0$ program P nie kończy działania.

Poprawność

Poprawność względem

$\{A\}\ P\ \{B\}$

wymaga spełnienia dwóch warunków:

- częściowa poprawność $\{A\}\ P\ \{B\}$

  („wynik poprawny”, gdy P zakończy działanie)
- własność stopu:

  jeśli przed wykonaniem P zachodzi A, to P zawsze kończy działanie

Własność stopu – przykład

Własność stopu: jeśli przed wykonaniem P zachodzi A, to P zawsze kończy działanie

P= while (x!=0) x $\leftarrow$ x-1

A= x – liczba **naturalna**

zachodzi własność stopu

P= while (x!=0) x $\leftarrow$ x-1

A= x – liczba **całkowita**

nie zachodzi własność stopu

Częściowa poprawność… cd

Warunek N jest niezmiennikiem programu P względem warunku początkowego A, gdy:

$$\{N \wedge A\}\ P\ \{N\}$$

czyli:

jeśli przed rozpoczęciem P jest spełniony warunek początkowy A i niezmiennik N, to po wykonaniu P nadal spełniony jest niezmiennik N.

Niezmiennik - przykład

Warunek $a+b=10$ jest niezmiennikiem programu

$a{+}{+};\ b{-}{-}$

względem warunku początkowego

$b>0$

gdyż:

$$\{a+b=10 \wedge b>0\}\ a{+}{+};\ b{-}{-}\ \{a+b=10\}$$

Niezmiennik pętli

Def. Warunek N jest niezmiennikiem pętli

while (A) P;

gdy N to niezmiennik P względem warunku początkowego A, czyli:

$$\{ N \wedge A \} \; P \; \{ N \}$$

INTUICJA

niezmiennik to warunek, którego prawdziwość nie zmienia się wskutek wykonania pętli.

Niezmiennik programu/pętli

Warunek $a+|b|=10$ jest niezmiennikiem programu a++; b – – względem warunku początkowego $b>0$ gdyż:

$$\{a+|b|=10 \wedge b>0\}$$

a++; b – –

$$\{a+|b|=10\}$$

Ale: $a+|b|=10$ nie jest niezmiennikiem programu a++; b – – bez warunku początkowego. Dlaczego?

Niezmiennik progr./pętli - przykład

Warunek $a+|b|=10$ jest niezmiennikiem pętli `while (b>0) {a++; b – –}` gdyż:

$$\{a+|b|=10 \land b>0\}\ a{+}{+};\ b{-}{-}\ \{a+|b|=10\}$$

Niezmiennik pętli - zastosowanie

Obserwacja.

Jeśli N jest niezmiennikiem pętli while (A) P to

$$\{ N \} \text{ while (A) P } \{ N \wedge \neg A \}$$

INTUICJA (do zapamiętania)

Jeśli przed rozpoczęciem pętli spełniony jest niezmiennik N to po zakończeniu pętli (nadal) spełniony jest N oraz zaprzeczenie warunku kontynuacji pętli, czyli $\neg A$.

Niezmiennik pętli - przykład

Warunek $a+|b|=10$ jest niezmiennikiem pętli `while (b>0) {a++; b – –;}`, czyli :

$$\{a+|b|=10 \wedge b>0\}\ a{+}{+};\ b{-}{-}\ \{a+|b|=10\}$$

Zatem zachodzi:

$$\{a+|b|=10\}$$

`while (b>0) {a++; b – –; }`

$$\{a+|b|=10 \wedge \neg b>0\}$$

Niezmienniki a częściowa poprawność

Po co taka sformalizowana notacja:

- automatyczne dowodzenie poprawności (przedmiot: metody programowania i in.)

Praktyka dowodzenia (prostych) programów:

- „wymyśl” niezmienniki kluczowych pętli
- uzasadnij (mniej lub bardziej formalnie) ich poprawność
- uzasadnij, że niezmiennik jest spełniony przy wejściu do pętli
- skorzystaj z tego, że niezmiennik jest spełniony po wyjściu z pętli

Praktyka dowodzenia poprawności

Bardziej formalnie:

- podaj „asercje” (warunki określające stan zmiennych) zachodzące w kluczowych punktach programu;
- wykaż, że podane asercje zachodzą, wykorzystując metodę niezmienników.

Przykład

Program P:

```
x=a; y=b; rez=1;
while (y!=0) { rez = rez*x; y=y – 1; }
```

Jaki efekt działania programu?

Gdzie program umieszcza „wynik” obliczeń?

Jak to pokazać?

Przykład – częściowa poprawność

Program P:

$x \leftarrow a$; $y \leftarrow b$; $rez \leftarrow 1$;

while (y!=0) { $rez \leftarrow rez*x$; $y \leftarrow y - 1$; }

Warunek wstępny A:

b – liczba naturalna

Warunek końcowy B:

$rez = a^b$

czyli program wyznacza a^b

PYTANIE: czy {A} P {B}

Przykład – częściowa poprawność

Program P:

~~$x \leftarrow a$; $y \leftarrow b$; $rez \leftarrow 1$;~~

while (y!=0) { $rez \leftarrow rez*x$; $y \leftarrow y - 1$; }

Niezmiennik pętli (???):

$$rez * x^y = a^b$$

1. Czy to rzeczywiście niezmiennik?
2. Czy pomoże pokazać, że $rez = a^b$ po zakończeniu programu?

Przykład – ad. 2 (czy pomoże?)

Pętla L:

while (y!=0) { rez $\leftarrow$ rez*x; y $\leftarrow$ y – 1; }

Jeśli $N \equiv rez * x^y = a^b$ to niezmiennik L, który jest spełniony przed wejściem do L, wówczas po wyjściu z pętli L mamy:

$$rez * x^y = a^b \text{ ORAZ } \neg y!=0$$

czyli $rez * x^y = a^b$ ORAZ $y=0$

czyli $rez * x^0 = a^b$

czyli $rez = a^b$ …… !

Przykład – ad. 2

Pętla L:

while (y!=0) { rez $\leftarrow$ rez*x; y $\leftarrow$ y – 1; }

- $N \equiv rez * x^y = a^b$ to niezmiennik L (założenie)
- instrukcje przed pętlą: $x \leftarrow a$; $y \leftarrow b$; $rez \leftarrow 1$ powodują, że

$$rez * x^y = 1 * a^b = a^b$$

przed wejściem do pętli, czyli niezmiennik

$$N \equiv rez * x^y = a^b$$

jest spełniony przed wejściem do pętli.

Przykład – asercje

Program P:

$\{$ b – liczba naturalna $\}$

$x \leftarrow a;\ y \leftarrow b;\ rez \leftarrow 1;$

$\{\ rez * x^y = a^b\ \}$

while (y!=0) { rez ← rez*x; y ← y – 1; }

$\{\ rez * x^y = a^b$ oraz $y=0\ \}$

Notacja – formuła z parametrami

Oznaczenie. Niech $F(x_1, \ldots, x_p)$ to formuła logiczna, w której występują (między innymi) zmienne wolne $x_1, \ldots, x_p$.

Wówczas $F(a_1, \ldots, a_p)$ to formuła F, w której wystąpienia zmiennej x_i zastępujemy przez a_i dla $i=1, \ldots, p$.

Przykład:

$N(rez, y) \equiv (y \geq 0) \wedge rez * x^y = a^b$

Wówczas:

$N(z, y+1) \equiv (y+1 \geq 0) \wedge z * x^{y+1} = a^b$

Przykład – ad. 1 (czy niezmiennik?)

Pętla L:

while (y!=0) { rez $\leftarrow$ rez*x; y $\leftarrow$ y – 1; }

TW. Warunek:

$$N(rez,y) \equiv rez * x^y = a^b$$

to niezmiennik pętli L.

DOWÓD.

Musimy pokazać:

$\{ N(rez,y) \wedge (y \neq 0) \}$ rez $\leftarrow$ rez*x; y $\leftarrow$ y – 1; $\{N(rez,y)\}$

Inaczej:

$\{(rez * x^y = a^b) \wedge (y \neq 0)\}$ rez $\leftarrow$ rez*x; y $\leftarrow$ y – 1; $\{rez * x^y = a^b\}$

Przykład – częściowa poprawność

Mamy pokazać:

$\{(y \neq 0) \wedge rez * x^y = a^b\}$ rez←rez*x; y←y – 1; $\{rez * x^y = a^b\}$

Niech:

rez', y' – wartości rez i y po wykonaniu

rez ← rez*x; y=y – 1;

Wówczas zakładając, że $\{rez * x^y = a^b\}$ musimy pokazać, że: $\{rez' * x^{y'} = a^b\}$

MAMY:

- rez' = rez*x, y'=y – 1
- $rez' * x^{y'} = rez * x * x^{y'} = rez * x * x^{y-1} = rez * x^y = a^b$

$\square$

Przykład – przypomnienie

Program P:

$x \leftarrow a$; $y \leftarrow b$; $rez \leftarrow 1$;

while ($y>0$) { $rez \leftarrow rez*x$; $y \leftarrow y - 1$; }

Warunek wstępny A:

b – liczba naturalna

Warunek końcowy B:

$rez = a^b$

Przykład – asercje

Program P:

{ b – liczba naturalna }

x=a; y=b; rez=1;

{ $rez * x^y = a^b$ }

while (y!=0) { rez = rez*x; y=y – 1; }

{ $rez * x^y = a^b$ oraz $y=0$ }

czyli { $rez = a^b$ }

Przykład – podsumowanie

Kroki dowodu częściowej poprawności programu

{A} P {B}:

- „wymyśliliśmy” niezmiennik N pętli L
- udowodniliśmy poprawność niezmiennika N
- pokazaliśmy, że niezmiennik N zachodzi przed wejściem do pętli L (o ile spełniony jest warunek początkowy programu A)
- sprawdziliśmy, że spełnienie N i zaprzeczenia warunku kontynuacji pętli L (czyli warunku y!=0) pociąga za sobą warunek końcowy B

Przykład – refleksja

Podany dowód częściowej poprawności:

- żmudny, sformalizowany ☹;
- najciekawsze – jaki wybrać (wymyślić?) niezmiennik pętli… tak, żeby było możliwe jego uzasadnienie i żeby pomógł w analizie programu

Co dalej (z częściową poprawnością):

- będziemy wybierać niezmienniki…
- … takie, które przekonają **nas**, że nasze programy są poprawne
- … a dowody będą mniej formalne.

Przykład – własność stopu

Pętla:

while (y!=0) { rez ← rez*x; y ← y – 1; }

Pyt.: Czy zawsze kończy działanie dla <u>naturalnego</u> y?

Odp.: TAK, ponieważ:

- jeśli y=0: zakończy natychmiast
- wpp: y będzie zmniejszane w każdym obrocie pętli, aż do osiągnięcia wartości zero – pętla również się zakończy (formalnie – np. indukcja).

Przykład – bubble

Program P:

```
i ← 0;
while (i<j–1)     {
   if (a[i]>a[i+1]) zamień(a[i],a[i+1]);
   i ← i+1;
}
```

Jaki cel tego programu?

Jaki efekt?

Przykład – bubble

Program P:

```
i ← 0;
while (i<j–1)    {
   if (a[i]>a[i+1]) zamień(a[i],a[i+1]);
   i ← i+1;
}
```

Warunek wstępny A:

$j>0$

Warunek końcowy B:

$a[j-1]=\max\{a[0],\ldots,a[j-1]\}$

Przykład – bubble

Pętla L:

```
~~i ← 0;~~
while (i<j–1)     {
   if (a[i]>a[i+1]) zamień(a[i],a[i+1]);
   i ← i+1;
}
```

Niezmiennik:

$$a[i]=\max\{a[0],\ldots,a[i]\} \wedge 0 \le i \le j-1$$

Przykład – bubble

Pętla L:

while (i<j–1) {

if (a[i]>a[i+1]) zamień(a[i],a[i+1]); i $\leftarrow$ i+1;}

Niezmien.: N(a,i) $\equiv$ a[i]=max{a[0],…,a[i]} $\wedge$ $0 \leq i \leq j-1$

Dowód poprawności niezmiennika (szkic):

Założenie: (N(a,i) oraz i<j – 1) przed wykonaniem:

if (a[i]>a[i+1]) zamień(a[i],a[i+1]); i $\leftarrow$ i+1;.

Cel: pokazać, że N(a',i') po wykonaniu

if (a[i]>a[i+1]) zamień(a[i],a[i+1]); i $\leftarrow$ i+1.

gdzie a', i' to wartości a oraz i po wykonaniu powyższej instrukcji

Przykład – bubble

Pętla L:

while $(i<j-1)$ {

if $(a[i]>a[i+1])$ zamień$(a[i],a[i+1])$; $i \leftarrow i+1$;}

Niezmiennik: $N(a,i) \equiv a[i]=\max\{a[0],\ldots,a[i]\} \wedge 0 \le i \le j-1$

Dowód poprawności niezmiennika:

Przypadek 1: $a[i]>a[i+1]$

- $a[i]$ był równy $\max\{a[0],\ldots,a[i]\}$, został zamieniony z elementem na pozycji $i+1$ i jest większy od elementu, z którym został zamieniony, więc mamy:

$$a'[i']=\max\{a'[0],\ldots,a'[i']\}$$

gdzie a' to tablica a po zamienie $a[i]$ z $a[i+1]$, $i'=i+1$.

Przykład – bubble

Pętla L:

while (i<j–1)   {

  if (a[i]>a[i+1]) zamień(a[i],a[i+1]); i $\leftarrow$ i+1;}

Niezmiennik: N(a,i) $\equiv$ a[i]=max{a[0],…,a[i]} $\wedge$ 0 $\leq$i$\leq$ j – 1

Dowód poprawności niezmiennika:

Przypadek 2: $\neg$ (a[i]>a[i+1])

a[i] był równy max{a[0],…,a[i]}, a[i+1] jest od niego większy (lub równy), tablica a się nie zmienia więc:

a'[i']=max{a'[0],…,a'[i']}

gdzie a' jest taka sama jak a, i'=i+1.

Przykład – bubble

Pętla L:

```
while (i<j–1)     {
   if (a[i]>a[i+1]) zamień(a[i],a[i+1]); i=i+1;}
```

Pokazaliśmy:

$$N(a,i) \equiv a[i]=\max\{a[0],\ldots,a[i]\} \wedge 0 \leq i \leq j-1$$

to niezmiennik pętli L.

Przykład – bubble

Pętla L:

```
while (i<j–1)        {
    if (a[i]>a[i+1])
          zamień(a[i],a[i+1]);
    i ← i+1;}
```

Niezmiennik:

$N(a,i) \equiv$

$a[i] = \max\{a[0],\ldots,a[i]\} \wedge$

$0 \leq i \leq j-1.$

Sortowanie bąbelkowe:

```
j=n;
while (j>0) {
    i ← 0;
    while (i<j–1)        {
          if (a[i]>a[i+1])
            zamień(a[i],a[i+1]);
          i ← i+1;} j--;
}
```

CZĘŚCIOWA POPRAWNOŚĆ:

po każdym obrocie głównej pętli max{a[0],…,[j-1]} trafia na pozycję j-1;

Przykłady – dlaczego tylko while?

Inne pętle można „zamienić” na while

(ćw.)

Niezmienniki - podsumowanie

- Niezmienniki pętli możemy określać bez formułowania formalnej specyfikacji całych programów.
- Będziemy używać niezmienników w mniej formalnych „dowodach” poprawności programów.
- Samo sformułowanie i nieformalne uzasadnienie właściwego niezmiennika jest często ważnym krokiem do dowodu poprawności.

Jeszcze jeden przykład – selection

Program P:

```
k ← 0; i ← 1;
while (i<n)        {
   if (a[i]<a[k]) k ← i;
   i++;
}
zamień(a[k],a[0])
```

Jaki cel tego programu?

Jaki efekt?

Przykład – selection

Program P:

$k \leftarrow 0$; $i \leftarrow 1$;
while ($i<n$) {
  if ($a[i]<a[k]$) $k \leftarrow i$;
  i++;
}
zamień($a[k]$,$a[0]$)

Można pokazać:

- niezmiennik pętli:

$N(k,i) \equiv$

$a[k]=\min\{a[0],\ldots,a[i-1]\}$

$\wedge\ 0 \le i \le n$

A dla całego programu P:

- $\{n>0,\ n \text{ naturalne}\}$

$$P$$

$\{\ a[0]=\min\{a[0],\ldots,a[n-1]\}\ \}$

Przykład – selection - pętla

Pętla L:

```
while (i<n)        {
   if (a[i]<a[k]) k ← i;
   i++;
}
```

Niezmiennik:

$N(k,i) \equiv a[k]=\min\{a[0],\ldots,a[i-1]\} \wedge 0 \leq i \leq n$

Przykład – selection - pętla

TW. $N(k,i) \equiv a[k]=\min\{a[0],\ldots,a[i-1]\} \wedge 0 \leq i \leq n$ to niezmiennik pętli L.

Dowód:

Zakł. że $N(k,i) \wedge (i<n)$ przed wykonaniem

if (a[i]<a[k]) k $\leftarrow$ i; i++;

Chcemy pokazać, że N(k',i'), gdzie

k', i' to wartości k, i po if (a[i]<a[k]) k $\leftarrow$ i; i++;

Przypadek 1: a[i]<a[k]

k' ustawiamy na „nowe minimum"

Przypadek 2: $\neg$a[i]<a[k]

k' równe k, minimum się nie zmienia

Nowe problemy:
scalanie
podział

Scalanie

Specyfikacja:

Wejście:

- n, m – liczby naturalne
- $a[0] \le a[1] \le \ldots \le a[n-1]$
- $b[0] \le b[1] \le \ldots \le b[m-1]$

Wyjście:

- $c[0] \le c[1] \le \ldots \le c[n+m-1]$ takie, że

  $\{c[0], c[1], \ldots, c[n+m-1]\}$ to suma multizbiorów $\{a[0], \ldots, a[n-1]\}$ i $\{b[0], \ldots, b[m-1]\}$

Scalanie – przykład

Przykład:

Wejście:

7 5 3 5 5 7 8 8 9 3 4 6 12 14

n m *a[0]*,…,*a[n-1]* *b*[0],…,*b*[m-1]

Wyjście:

3 3 4 5 5 6 7 8 8 9 12 14

Scalanie

Algorytm:

1. $i \leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$ //i, j – wskazują dokąd skopiowane
2. dopóki (i<n oraz j<m)
   a) jeżeli(a[i]<b[j]) // wybór najmn. jeszcze nie w c
      - $c[k] \leftarrow a[i]$, $i \leftarrow i+1$

   b) w przeciwnym przypaku:
      - $c[k] \leftarrow b[j]$, $j \leftarrow j+1$

   c) $k \leftarrow k+1$
3. dopóki (i<n) // kopiowanie „ogona” z a
   a) $c[k] \leftarrow a[i]$, $i \leftarrow i+1$, $k \leftarrow k+1$
4. dopóki (j<m) // kopiowanie „ogona” z b
   a) $c[k] \leftarrow b[j]$, $j \leftarrow j+1$, $k \leftarrow k+1$

Scalanie – implementacja

```
merge(int a[],int b[],int c[],int n,int m)
{ int i,j,k;
  i=j=k=0;
  while (i<n && j<m)
    if (a[i]<b[j]) { c[k++]=a[i++]; }
    else           { c[k++]=b[j++] }
  while (i<n) { c[k++]=a[i++]; }
  while (j<m) { c[k++]=b[j++];}
}
```

Scalanie – implementacja

```
def merge(a, b, c, n, m):
    i=j=k=0
    while i<n and j<m:
        if a[i]<b[j]:
            c[k]=a[i]
            k=k+1
            i=i+1
        else:
            c[k]=b[j]
            k=k+1
            j=j+1
    while i<n:
        c[k]=a[i]
        k=k+1
        i=i+1
    while j<m:
        c[k]=b[j]
        k=k+1
        j=j+1
```

Scalanie

Oznaczenia:

- $x[p..k]$ – multizbiór $\{x[p],x[p+1],\ldots,x[k]\}$ (zbiór z powtórzeniami) lub zbiór pusty gdy $p>k$
- $A \cup B$ – suma multizbiorów A i B!

„z powtórzeniami" oznacza, że dopuszczamy wiele wystąpień tego samego elementu.

Przykład:

$\{5,3,3,2\} \cup \{3,4,7\}=\{5,3,3,3,2,4,7\}$

Scalanie - poprawność

Poprawność formalnie (przypomnienie):

- Częściowa poprawność – jeśli program się zakończy, wynik będzie poprawny.
- Własność stopu – program zawsze zakończy działanie.

Nasz cel:

- Ustalmy niezmienniki pętli pomocne w dowodzeniu częściowej poprawności.
- (Nieformalnie) uzasadnijmy częściową poprawność programu, w oparciu o niezmienniki.

Scalanie – częściowa poprawność

Niezmiennik pierwszej pętli (`while i<n and j<m`):

1) $c[0..k-1] = a[0..i-1] \cup b[0..j-1]$

   skopiowaliśmy odpowiednie fragmenty a i b do c

2) $c[0] \leq \ldots \leq c[k-1]$

   c jest uporządkowana

3) $a[i] \geq c[k-1]$ lub $i=n$

4) $b[j] \geq c[k-1]$ lub $i=m$

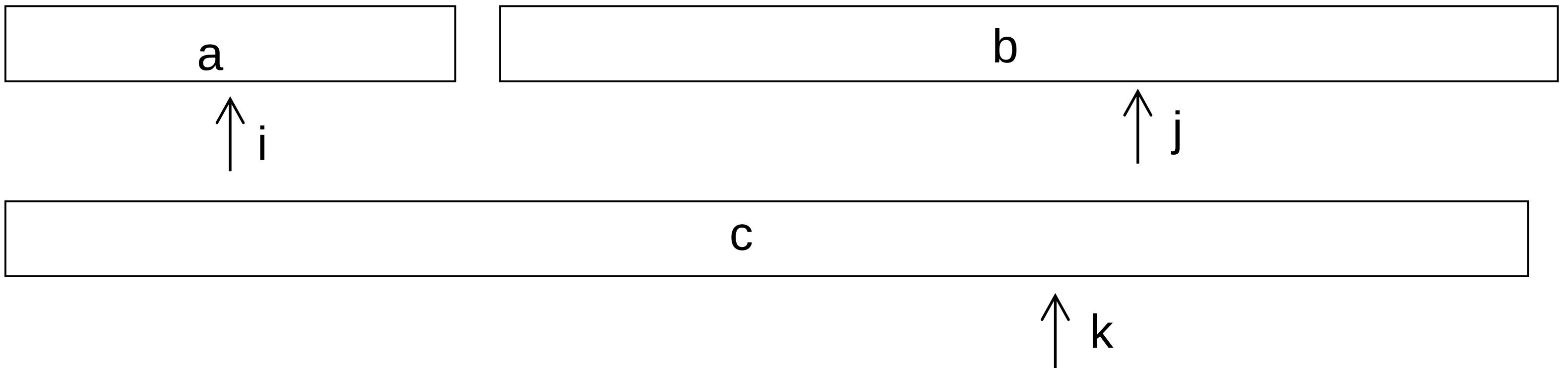
   nieskopiowane elementy a i b są większe (lub równe) od tych już w c

Scalanie – częściowa poprawność

Wniosek 1.

Po zakończeniu pierwszej (czerwonej) pętli:

- do c skopiowaliśmy cały ciąg a[0..n – 1] lub cały ciąg b[0..m – 1] – wynika z (1) i warunku zakończenia pętli
- nieskopiowane do c elementy ciągów a i b są niemniejsze od elementów już w c – z (4) i (5)
- elementy c są uporządkowane – z (2)
- indeksy i, j wskazują na początek „nieskopiowanych” do c części ciągów a i b.



Scalanie – częściowa poprawność

Wniosek 2.

- druga pętla kopiuje do c pozostałe elementy a – większe od dotychczas umieszczonych w c;
- trzecia pętla kopiuje do c pozostałe elementy b – większe od dotychczas umieszczonych w c;

Scalanie – własność stopu + złożoność

Pętla 1 - „`while i<n and j<m`”:

- każdy obrót pętli zwiększa o 1 sumę i+j
- na początku i+j=0
- Wniosek: po co najwyżej n+m krokach mamy i≥n lub j≥m

Pętla 2 – „`while i<n`” (pętla 3 analogicznie):

- każdy obrót pętli zwiększa i o 1
- i nie jest zmniejszane w pętli, wartość początkowa i≥0
- Wniosek: po co najwyżej n krokach mamy i≥n

Scalanie - złożoność

Rozmiar danych: n + m

Czas : O(n+m)

Pamięć : O(n+m)

Nowe problemy: podział

Podział

Specyfikacja:

Wejście (można zapisać jako formułę opis. stan):

- l, p – liczby naturalne takie, że $l<p$
- a – tablica elementów ze zbioru uporz.

Wyjście (można zapisać jako formułę opis. stan):

- s – liczba naturalna taka, że $l \leq s < p$
- multizbiór {a[l],..,a[p]} bez zmian, ale w takiej kolejności, że
  - a[l]≤x, a[l+1]≤x, …, a[s]≤x
  - a[s+1]≥x, a[s+2]≥x,….a[p]≥x

  dla pewnego $x \in$ {a[l],..,a[p]}.

Podział – przykład

Przykład:

Wejście:

- l=0, p=9
- a[$l..p$] = 5 1 4 9 5 5 8 7 3 5

Wyjście (przykładowe):

- a[$l..p$] = 5 1 4 3 5 **5** 8 7 9 5
- s=5

dla x = 5

Podział

Algorytm:

1. $x \leftarrow a[l]$, $i \leftarrow l$, $j \leftarrow p$
2. dopóki $i < j$ powtarzaj:
   1. zwiększaj i o 1 aż do spełnienia warunku $a[i] \geq x$
   2. zmniejszaj j o 1 aż do spełnienia warunku $a[j] \leq x$
   3. jeśli i<j:
      - zamień $a[i]$ z $a[j]$, zwiększ i o 1, zmniejsz j o 1
3. zwróć j

Podział – implementacja

```
podzial(int l, int p, int a[])
{ int i,j,y,x;
  x=a[l]; i=l; j=p;
  while (i<j)
  { while (a[j]>x) j--;
    while (a[i]<x) i++;
    if (i<j)
     { y=a[j]; a[j]=a[i]; a[i]=y;
       i++; j--; }
  }
  return j;
}
```

Podział – implementacja

```
def podzial(l,p,a):
    x=a[l]
    i=l
    j=p
    while i<j:
        while a[j]>x: j=j-1
        while a[i]<x: i=i+1
        if i<j:
            y=a[j]
            a[j]=a[i]
            a[i]=y
            i=i+1
            j=j-1
    return j
```

Podział

Pytania:

1. Dlaczego *i* „zatrzymuje się” na elemencie $\geq$x (a nie na elemencie >x)?
2. Dlaczego *j* „zatrzymuje się” na elemencie $\leq$x (a nie na elemencie <x)?

Sprawdź, …. kiedy możemy mieć nieskończone pętle wewnętrzne?

```
podzial(int l, int p, int a[])
{ int i,j,y,x;
  x=a[l]; i=l; j=p;
  while (i<j)
  { while (a[j]>x) j--;
    while (a[i]<x) i++;
    if (i<j)
        { y=a[j]; a[j]=a[i]; a[i]=y;
      i++; j--; }
  }
  return j;
}
```

Podział – częściowa poprawność

Częściowa poprawność – intuicja:

- „na lewo” od *i* tylko elementy ≤x
- „na prawo” od *j* tylko elementy ≥x
- miejsce podziału wyznaczamy w momencie „spotkania” indeksów *i* oraz *j*

```
podzial(int l, int p, int a[])
{ int i,j,y,x;
  x=a[l]; i=l; j=p;
  while (i<j)
  { while (a[j]>x) j--;
    while (a[i]<x) i++;
    if (i<j)
        { y=a[j]; a[j]=a[i]; a[i]=y;
      i++; j--; }
  }
  return j;
}
```

Podział – częśc. popr.

```
podzial(int l, int p, int a[])
{ int i,j,y,x;
  x=a[l]; i=l; j=p;
  while (i<j)
  { while (a[j]>x) j--;
    while (a[i]<x) i++;
    if (i<j)
        { y=a[j]; a[j]=a[i]; a[i]=y;
      i++; j--; }
  }
  return j;
}
```

Niezmiennik głównej pętli:

1) $i \le j+1$
2) $a[s] \le x$ dla $l \le s < i$ //na lewo od i nie ma większych od x
3) $a[s] \ge x$ dla $j < s \le p$ //na prawo od j nie ma mniejszych od x
4) $l \le j \le p$, $l \le i \le p$ //i,j są pomiędzy l a p
5) w ciągu a[i], a[i+1],…,a[p] występuje element $\ge x$
   //i „zatrzyma się” najpóźniej na p
6) w ciągu a[l], a[l+1],…,a[j] występuje element $\le x$

   //j „zatrzyma się” najpóźniej na l

Podział – częściowa poprawność

Niezmiennik głównej pętli - dlaczego tak skomplikowany:

- Musi być spełniony również po obrocie pętli
- Powinien pomóc w dowodzie poprawności całej funkcji podzial.

Podział – własność stopu

Obserwacje:

- różnica *j* – *i* zmniejsza się w każdym obrocie głównej pętli,
- gdy *j* – *i* $\leq$0 – kończymy pętlę
- więc liczba obrotów głównej pętli ograniczona
- pętle wewnętrzne kończą się dzięki (5) i (6)

```
podzial(int l, int p, int a[])
{ int i,j,y,x;
  x=a[l]; i=l; j=p;
  while (i<j)
  { while (a[j]>x) j--;
    while (a[i]<x) i++;
    if (i<j)
        { y=a[j]; a[j]=a[i]; a[i]=y;
      i++; j--; }
  }
  return j;
}
```

Podział – złożoność

Czas: O($p - l + 1$)

- liczba kroków potrzebna do uzyskania $j - i \leq 0$

Pamięć:

- O($p - l + 1$), a właściwie….
- $p - l + 1$ + O(1): poza pamięcią z danymi potrzebujemy tylko O(1) zmiennych, czyli algorytm działa w miejscu.

„w miejscu”

Algorytm działa w miejscu jeśli wykorzystuje tylko:

- pamięć zajmowaną przez dane wejściowe [założenie – dane już w pamięci]
- **oraz** dodatkową pamięć rozmiaru O(1)

Podział – sprytniejsza implementacja w C

```
podzial(int l, int r, int a[])
{ int y;
  l--;
  r++;
  do
  { while (a[--r]>x);
    while (a[++l]<x);
    if (l<r) {y=a[r]; a[r]=a[l]; a[l]=y;}
    else return r;
  } while (1);
}
```