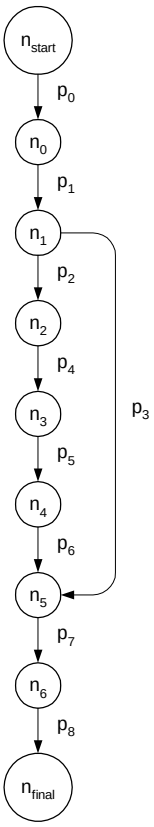
Aufgabe A1

a) Die Abbildung des zu untersuchenden Code auf einen Kontrollflussgraphen:



n_{start}	`private void order(int a,int b)`
	`{`
n_0	`  int tmp;`
n_1	`  if (a>b)`
	`  {`
n_2	`  tmp = a;`
n_3	`  a = b;`
n_4	`  b = tmp;`
	`  }`
n_5	`  global_a=a;`
n_6	`  global_b=b;`
n_{final}	`}`

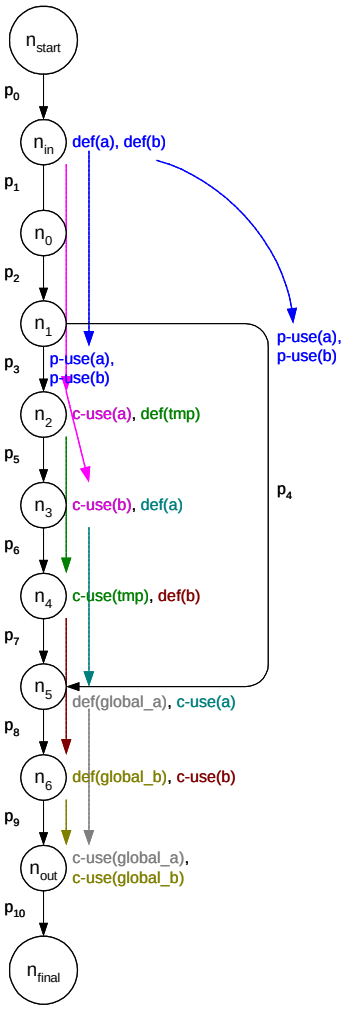
b) Jeder der Knoten n_i ist zu passieren, es reicht ein Testfall aus, der sequentiell den Graphen durchläuft:

Testfall	a	b	n_{start}	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_{final}
1	1	0									

c) Jeder Zweig p_i ist zu passieren. Da eine Alternative im Knoten n_1 auftritt, sind zwei Testfälle notwendig:

Testfall	a	b	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	1	0									
2	0	1									

Aufgabe A2



a) Um datenflussorientierte Verfahren durchführen zu können, muss derontrollflussgraph derart modifiziert werden, dass alle *defs*, *c-uses* und *p-uses* eingetragen werden. Die definitionsfreien Pfade sind farblich gekennzeichnet, um in den nachfolgenden Aufgabenteilen für mehr Klarheit zu sorgen.

b) Für den *all-defs*-Test sind alle Pfade zu durchlaufen, die min. ein *def* enthalten. Jedes *def* muss min. einmal durch ein *use* aufgelöst werden. Die *def*-Pfade sind mit einem einzelnen Test abdeckbar:

Testfall	a	b	n_{start}	n_{in}	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_{out}	n_{final}
1	1	0											

c) Der *p-uses*-Test muss alle *p-uses* überdecken:

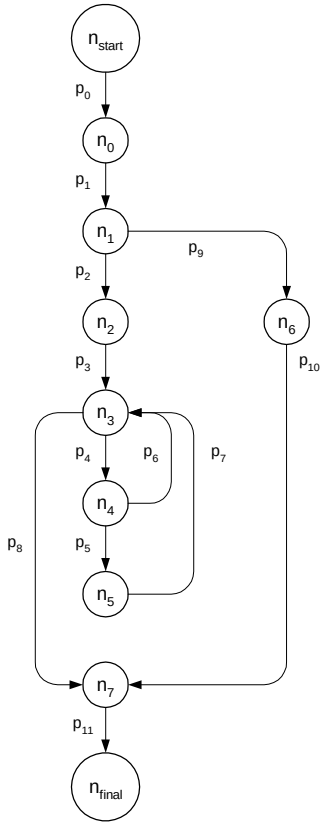
Testfall	a	b	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}
1	1	0											
2	0	1											

d) Der *c-uses*-Test muss alle *c-uses* überdecken:

Testfall	a	b	n_{start}	n_{in}	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_{out}	n_{final}
1	1	0											

Aufgabe B2

a) Der Kontrollflussgraph ist diesmal leider etwas aufwändiger:




```
n_start   private boolean arrayEquals(char[] comp)
          {
n_0          boolean result;

n_1          if (comp.length != string.length)
             {
n_6             result = false;
             }
             else
             {
n_2             result = true;
n_3             for (int i=0; i<string.length &&
                              result; i++)
                {
n_4                if (string[i] != comp[i])
                   {
n_5                   result = false;
                   }
                }
             }
n_7          return result;
n_final   }
```

b) Mit nur zwei Testfällen können alle Anweisungen durchlaufen werden.

Testfall	string	comp	n_{start}	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_{final}
1	abc	ab										
2	abc	abd										

c) Die Zweigüberdeckung ist mit den gleichen Testfällen machbar:

Testfall	string	comp	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
1	abc	ab												
2	abc	abd												

Aufgabe B2

a) Die `for`-Schleife kann mit ihrer Variablen `i` vom Typ `integer` potentiell `MAX_INT`-mal durchlaufen werden. Leider ist es relativ schwierig, mit vernünftigem Aufwand $2^{31} \approx 2 \cdot 10^9$ Pfade zu testen, deshalb sollte man sich eine andere Technik überlegen.

b) Jetzt sind nur noch 10+1 Pfade zu testen, was durchaus machbar ist.

c) Die Schleife darf entweder nie (k=0) oder einmal (k=1) durchlaufen werden. Die Zahlen in der Tabelle geben an, wie oft der jeweilige Pfad durchlaufen wird. Wichtig ist dabei p_4, da es den Eintritt in den Schleifenrumpf darstellt:

k=0:

Testfall	string	comp	p_0	p_1	p_2	p_3	**p_4**	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
1	*(leer)*	*(leer)*	1	1	1	1					1			1
2	ab	abc	1	1								1	1	1

k=1:

Testfall	string	comp	p_0	p_1	p_2	p_3	**p_4**	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
3	a	b	1	1	1	1	**1**	1		1	1			1
4	a	a	1	1	1	1	**1**		1		1			1

d) Der boundary-interior-Test entspricht einem strukturierten Pfadüberdeckungstest mit dem Parameter k=2. Die 4 bisherigen Testfälle für k=0 bzw. k=1 werden unverändert übernommen.

k=2:

Testfall	string	comp	p_0	p_1	p_2	p_3	**p_4**	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
5	abc	add	1	1	1	1	**2**	1	1	1	1			1