CS406: Compilers

Spring 2022

Week 13:

More Dataflow Analysis – Uninitialized Variables, Available Expressions, Reaching Definitions

Register Allocation

Uninitialized Variables

- **Goal:** determine a set of variables that are possibly uninitialized at the beginning and end of a basic block.
  - E.g. to know if `x==null`?
- **Direction of the analysis:**
  - How does information flow w.r.t. control flow?
- **Join operator:**
  - What happens at merge points? E.g. what operator to use Union or Intersection?
- **Transfer function:**
  - Define sets UninitIn(b), UninitOut(b), Init(b), Uninit(b)
- **Initializations?**

Worksheet

Available Expressions

- **Goal:** determine a set of expressions that have already been computed.
  - E.g. to perform global CSE
- **Direction of the analysis:**
  - How does information flow w.r.t. control flow?
- **Join operator:**
  - What happens at merge points? E.g. what operator to use Union or Intersection?
- **Transfer function:**
  - Define sets AvailIn(b), AvailOut(b), Compute(b), Kill(b)
- **Initializations?**

Transfer functions for meet

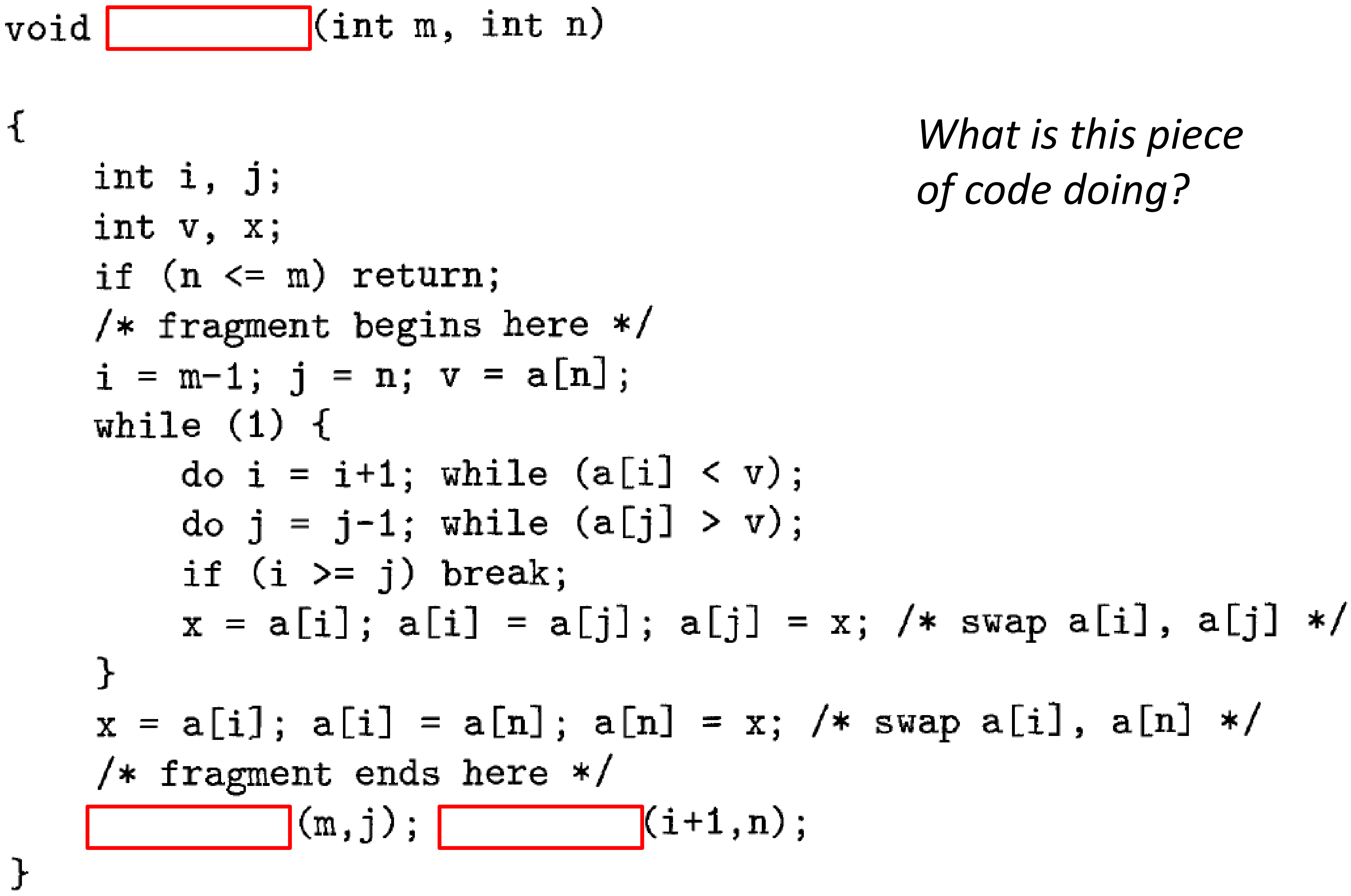
- What do the transfer functions look like if we are doing a meet?

$$\begin{aligned} IN(S) &= \cap_{t \in pred(s)} OUT(t) \\ OUT(S) &= \mathbf{gen}(s) \cup (IN(S) - \mathbf{kill}(s)) \end{aligned}$$

- gen(s): expressions that *must be* computed in this statement
- kill(s): expressions that use variables that *may* be defined in this statement
  - Note difference between these sets and the sets for reaching definitions or liveness
- Insight: gen and kill must never lead to incorrect results
  - Must not decide an expression is available when it isn't, but OK to be safe and say it isn't
  - Must not decide a definition *doesn't* reach, but OK to overestimate and say it does

Analysis initialization

- How do we initialize the sets?
  - If we start with everything initialized to $\bot$, we compute the smallest sets
  - If we start with everything initialized to $\top$, we compute the largest
- Which do we want? It depends!
  - Reaching definitions: a definition that *may* reach this point
    - We want to have as few reaching definitions as possible → $\bot$
  - Available expressions: an expression that *was definitely* computed earlier
    - We want to have as many available expressions as possible → $\top$
  - Rule of thumb: if confluence operator is $\sqcup$, start with $\bot$, otherwise start with $\top$



```
void            (int m, int n)

{
     int i, j;
     int v, x;
     if (n <= m) return;
     /* fragment begins here */
     i = m-1; j = n; v = a[n];
     while (1) {
          do i = i+1; while (a[i] < v);
          do j = j-1; while (a[j] > v);
          if (i >= j) break;
          x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
     }
     x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
     /* fragment ends here */
               (m,j);            (i+1,n);
}
```

What is this piece of code doing?

[1] R. Sedgewick, "Implementing Quicksort Programs," *Comm. ACM*, **21**, 1978, pp. 847–857.

Intermediate code (assuming int is 4 bytes):

(Ignore the temporary counter value for now)

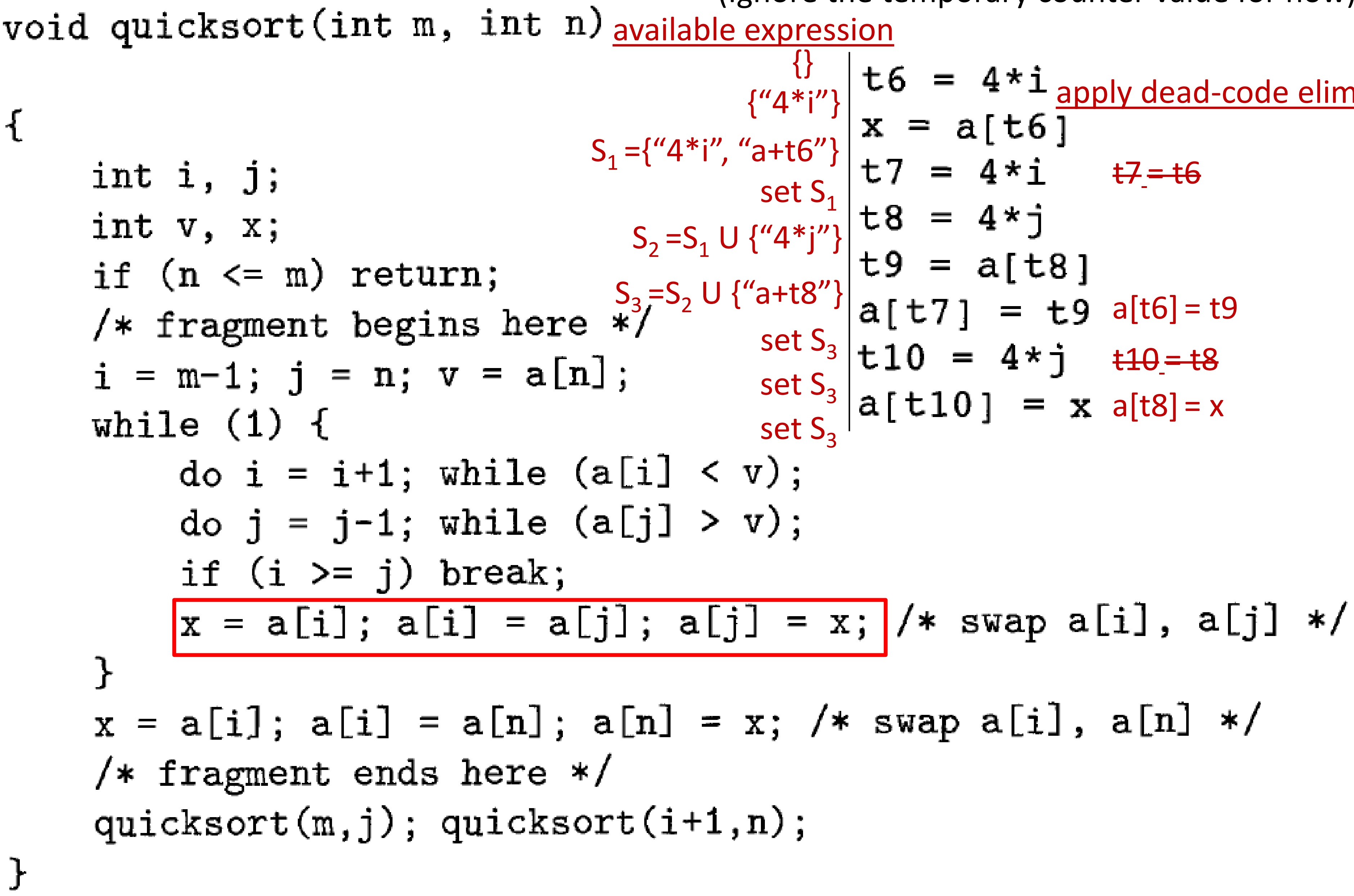
```
void quicksort(int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;  /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

available expression		Can be rewritten:
{}	`t6 = 4*i`	
{“4*i”}	`x = a[t6]`	
S_1 ={“4*i”, “a+t6”}	`t7 = 4*i`	t7 = t6
set S_1	`t8 = 4*j`	
S_2 =S_1 U {“4*j”}	`t9 = a[t8]`	
S_3 =S_2 U {“a+t8”}	`a[t7] = t9`	a[t6] = t9
set S_3	`t10 = 4*j`	t10 = t8
set S_3	`a[t10] = x`	a[t8] = x
set S_3		

copy propagation

Intermediate code (assuming int is 4 bytes):

(Ignore the temporary counter value for now)



```
void quicksort(int m, int n)
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

available expression		apply dead-code elim.
{}	`t6 = 4*i`	
{“4*i”}	`x = a[t6]`	
S_1 ={“4*i”, “a+t6”}	`t7 = 4*i`	~~t7 = t6~~
set S_1	`t8 = 4*j`	
S_2 =S_1 U {“4*j”}	`t9 = a[t8]`	
S_3 =S_2 U {“a+t8”}	`a[t7] = t9`	a[t6] = t9
set S_3	`t10 = 4*j`	~~t10 = t8~~
set S_3	`a[t10] = x`	a[t8] = x
set S_3		

```
void quicksort(int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Intermediate code
(after local CSE+copy prop.+dead-code elim.)

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
```

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
```

```
void quicksort(int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Intermediate code (assuming int is 4 bytes):

(assume next temporary counter value=11)

```
t11 = 4*i
x = a[t11]
t12 = 4*i       t12=t11
t13 = 4*n
t14 = a[t13]
a[t12] = t14    a[t11]=x
t15 = 4*n       t15=t13
a[t15] = x      a[t13]=x
```

Intermediate code (assuming int is 4 bytes):

```
void quicksort(int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

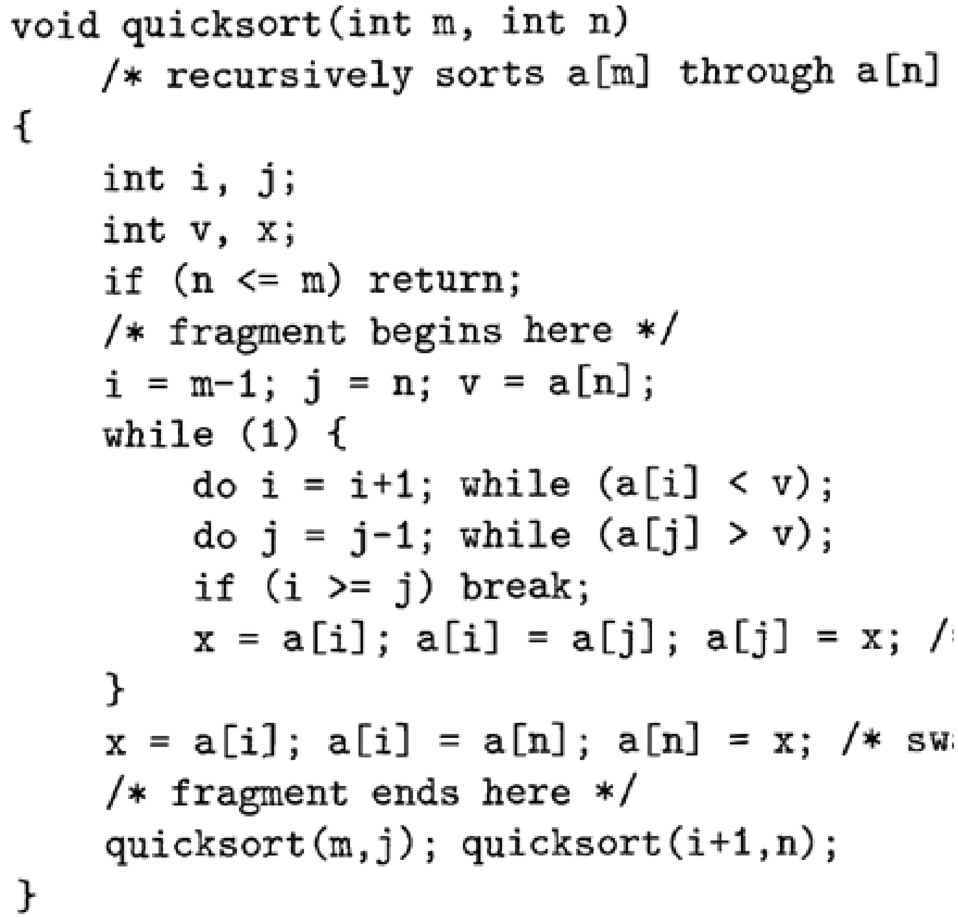
`t11 = 4*i`	after dead-code elim.
`x = a[t11]`	
`t12 = 4*i`	~~t12=t11~~
`t13 = 4*n`	
`t14 = a[t13]`	
`a[t12] = t14`	a[t11]=x
`t15 = 4*n`	~~t15=t13~~
`a[t15] = x`	a[t13]=x

Intermediate code
(after local CSE+copy prop.+dead-code elim.)

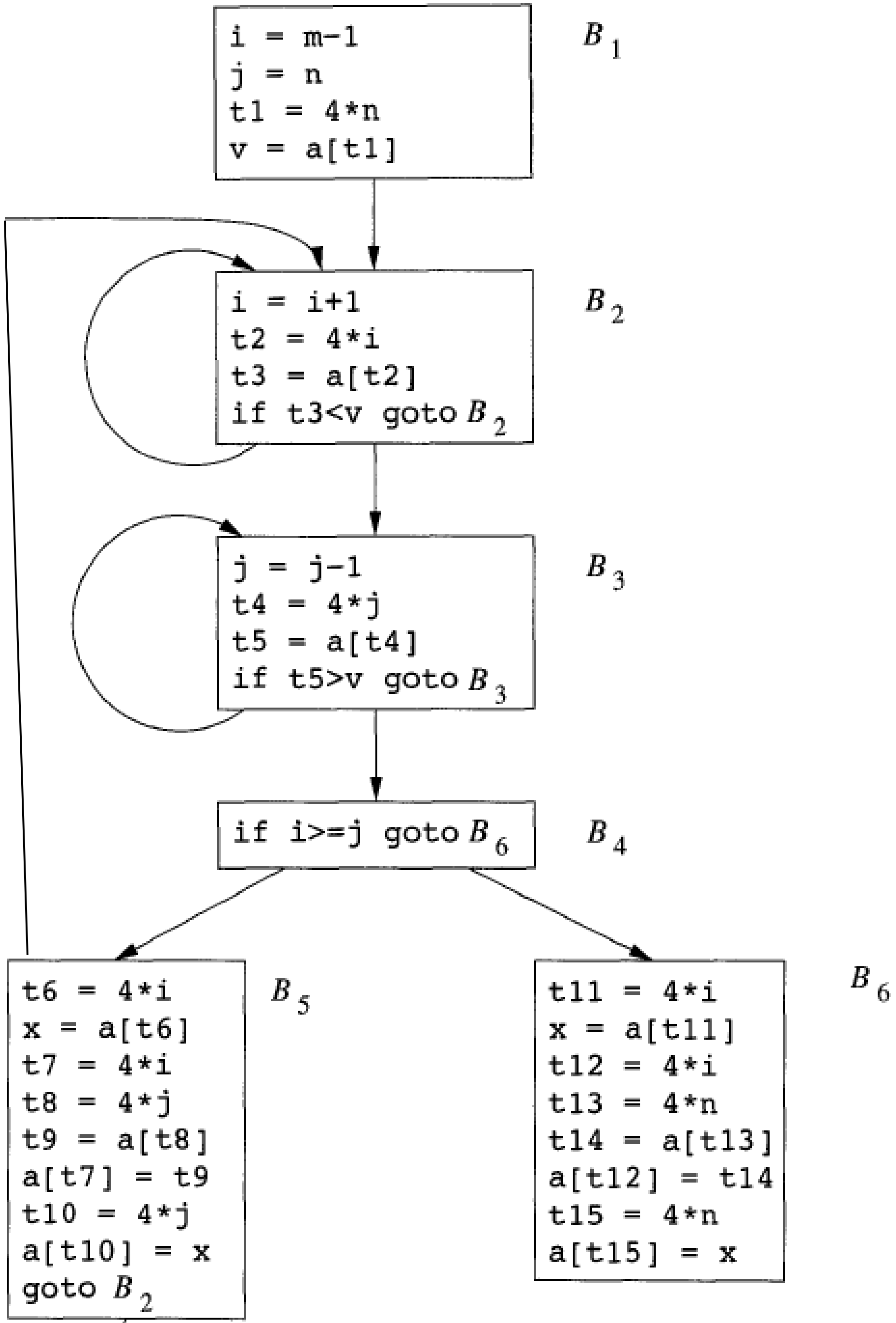
```
void quicksort(int m, int n)

{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

```
t11=4*I
x=a[t11]
t13=4*n
t14=a[t13]
a[t11]=x
a[t13]=x
```



```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

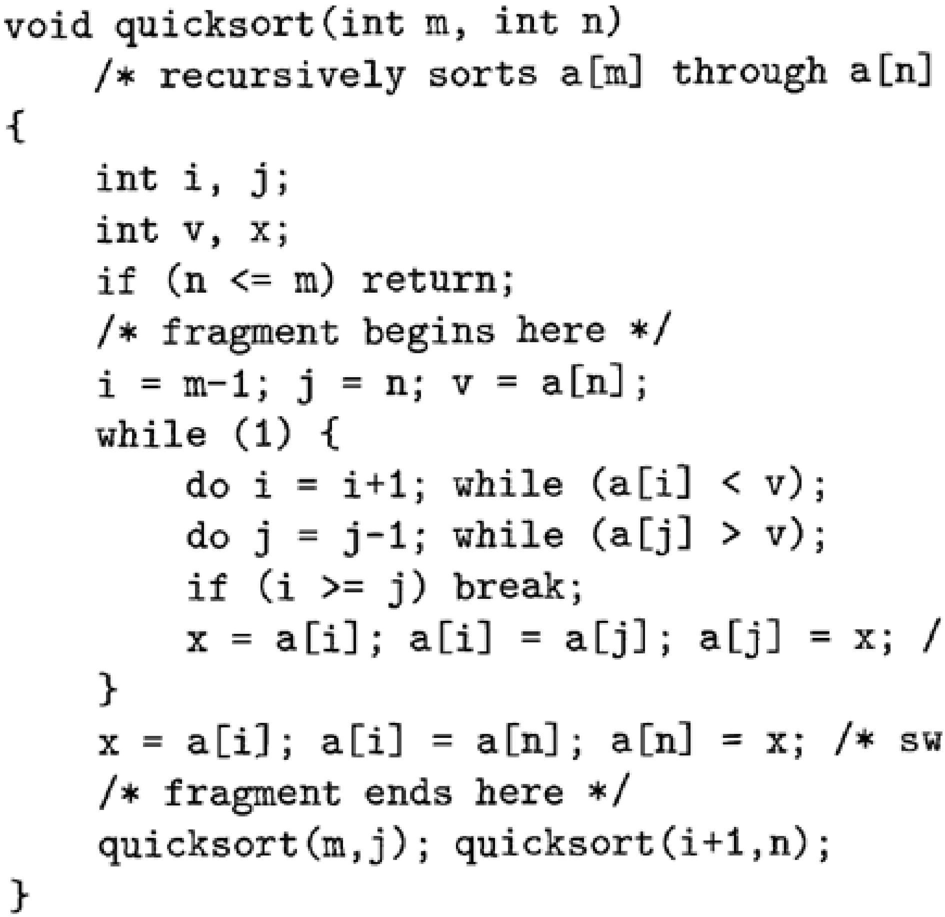
```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

```
if i>=j goto B6
```

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

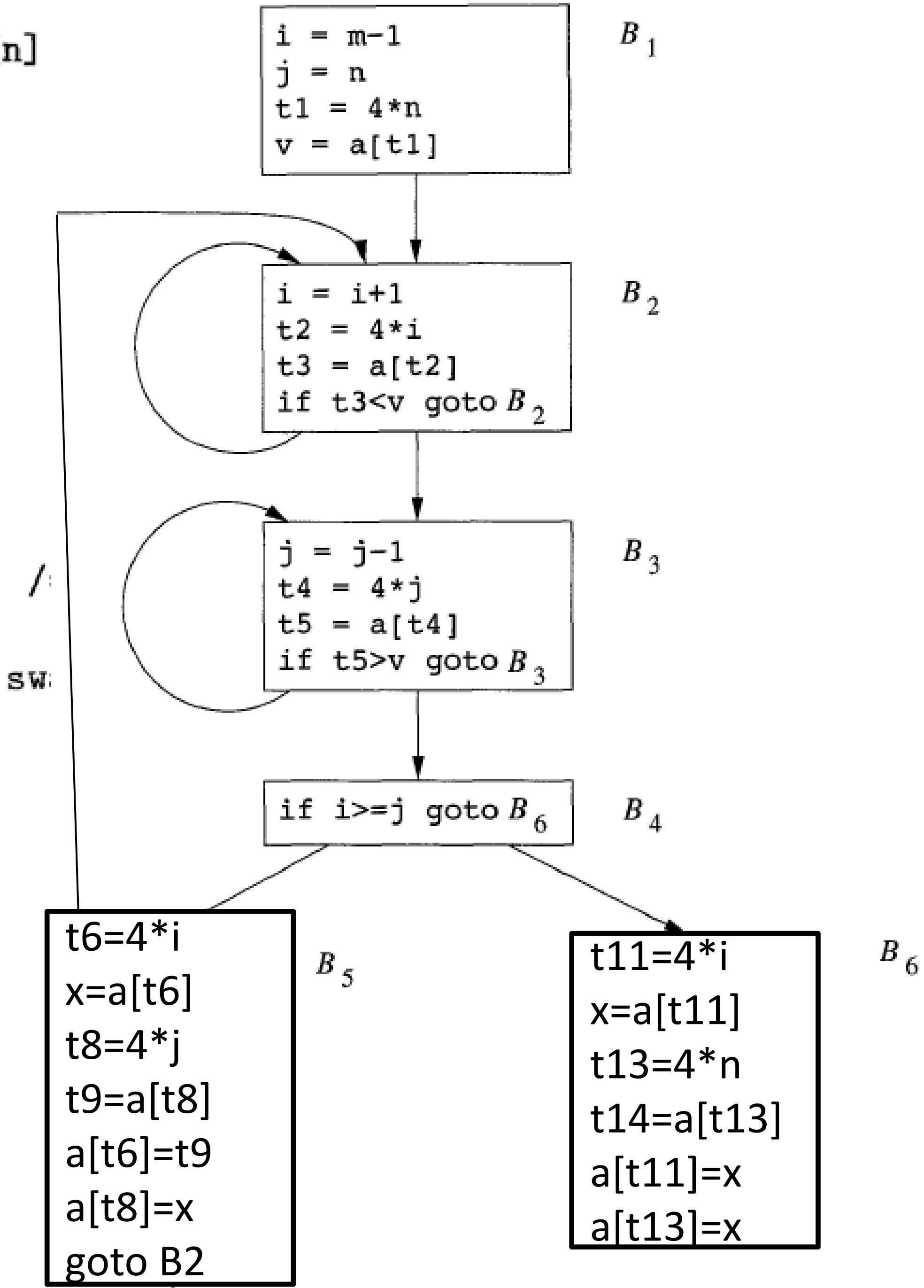
- CFG for quicksort

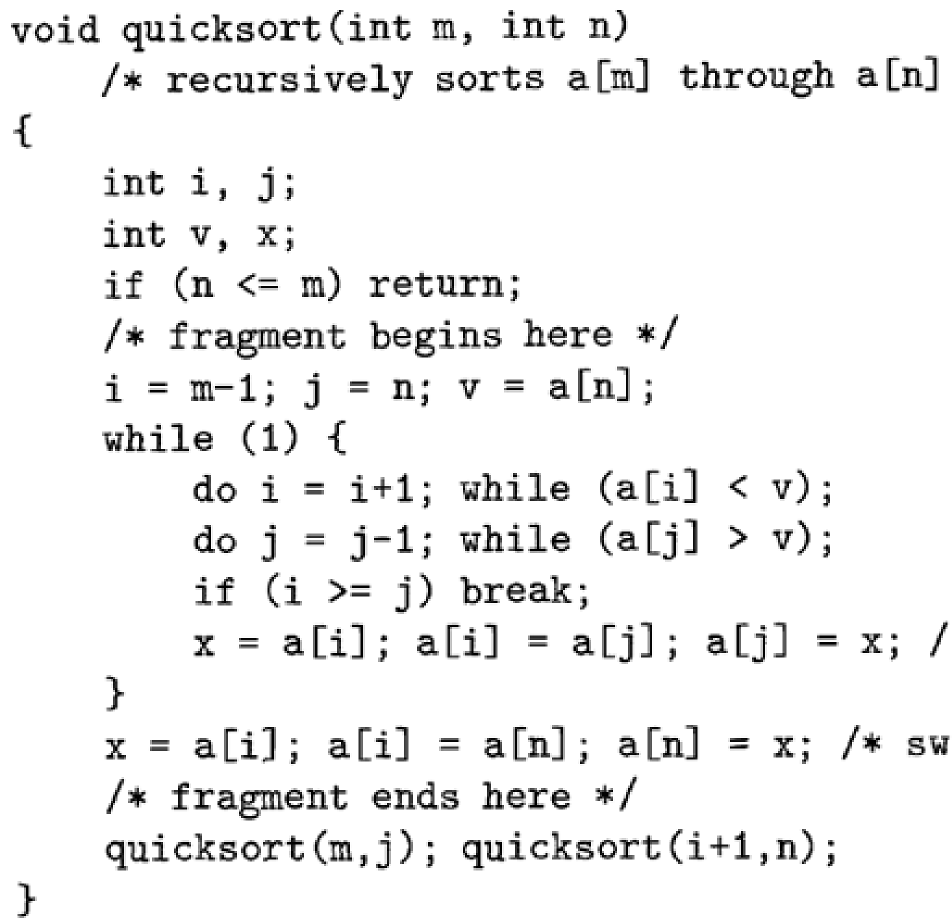


```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

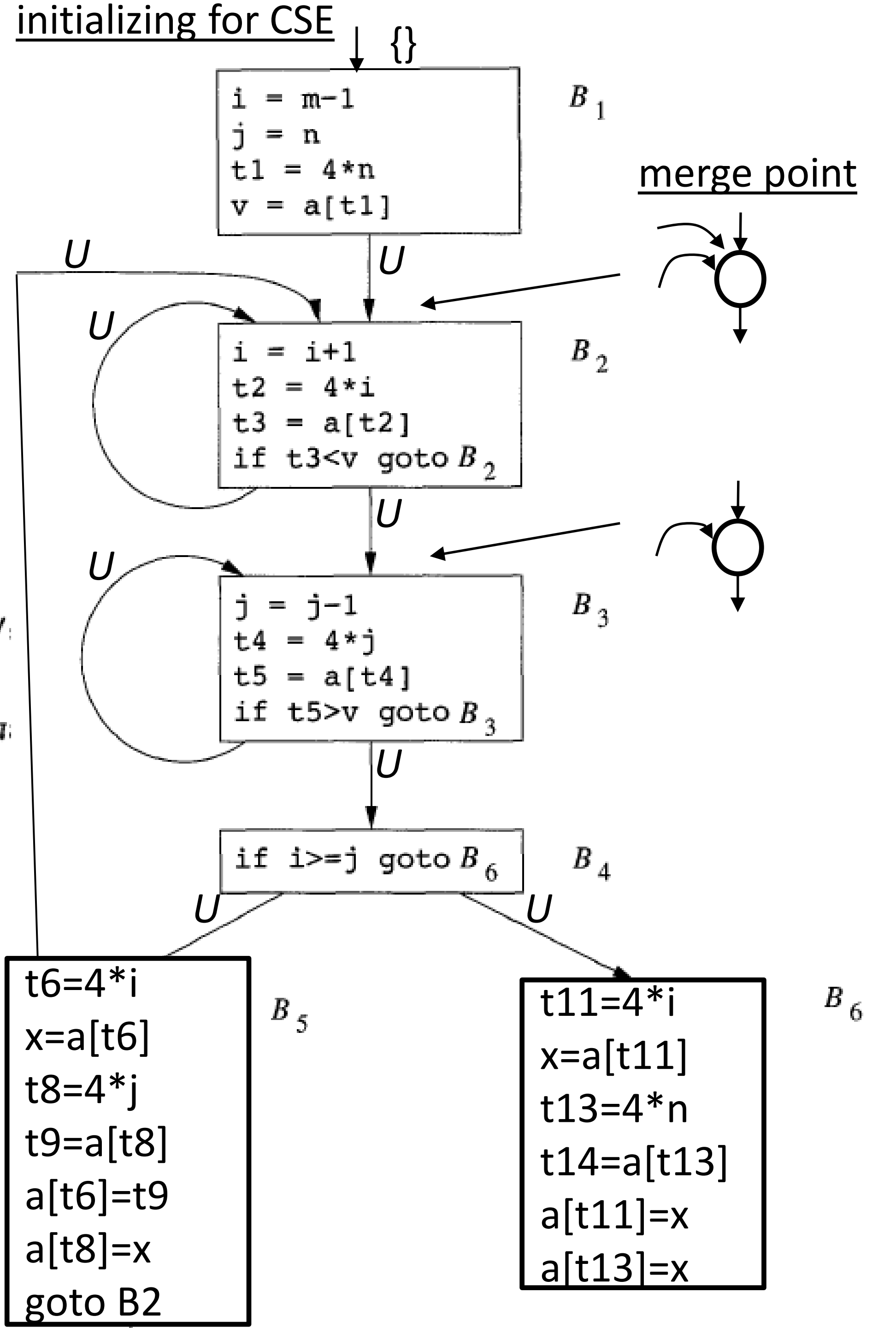
- CFG for quicksort

(after optimizing B5 and B6)





```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```



initializing for CSE

merge point

B1
```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

B2
```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

B3
```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

B4
```
if i>=j goto B6
```

B5
```
t6=4*i
x=a[t6]
t8=4*j
t9=a[t8]
a[t6]=t9
a[t8]=x
goto B2
```

B6
```
t11=4*i
x=a[t11]
t13=4*n
t14=a[t13]
a[t11]=x
a[t13]=x
```

- CFG for quicksort

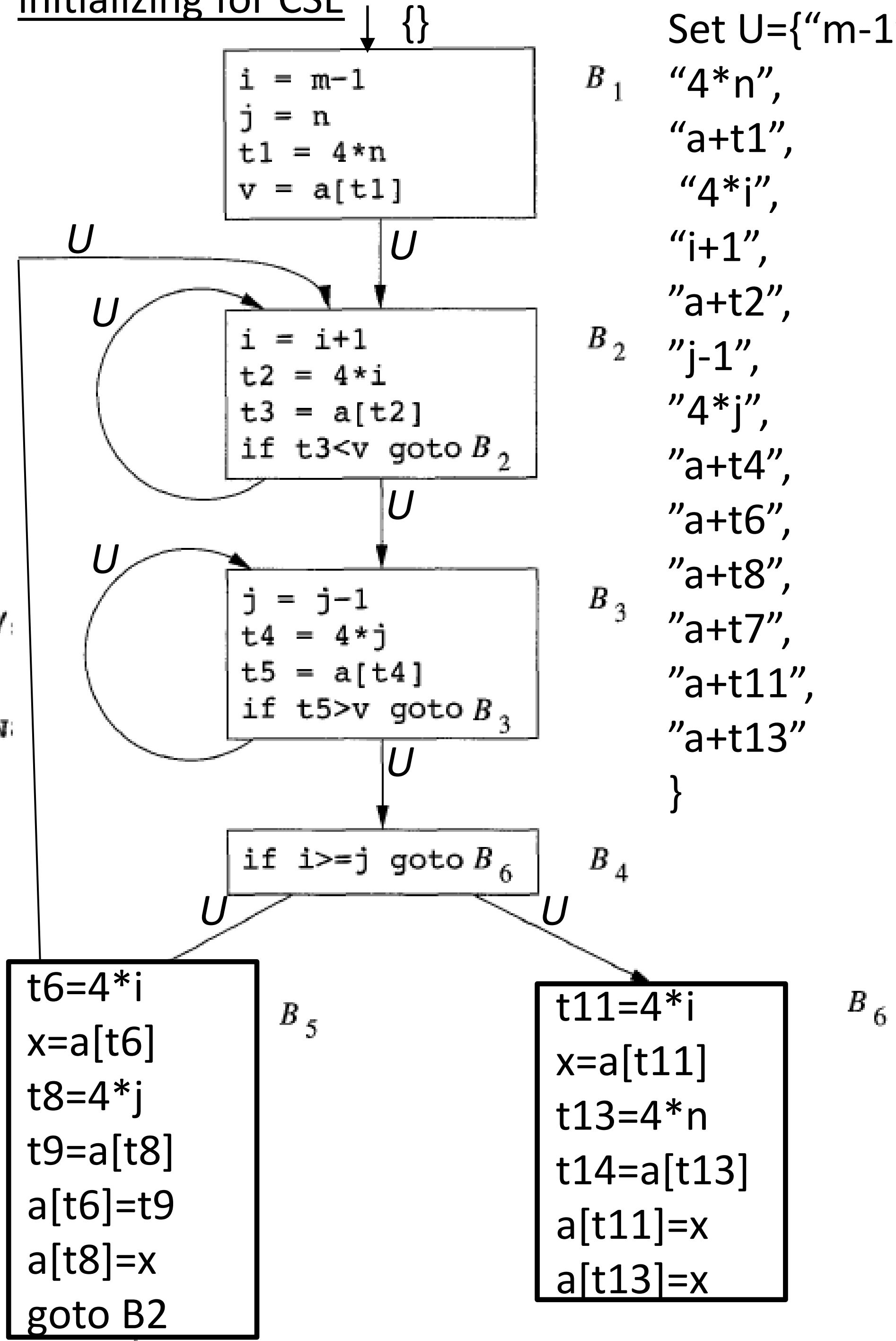
(after optimizing B5 and B6)

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

- CFG for quicksort

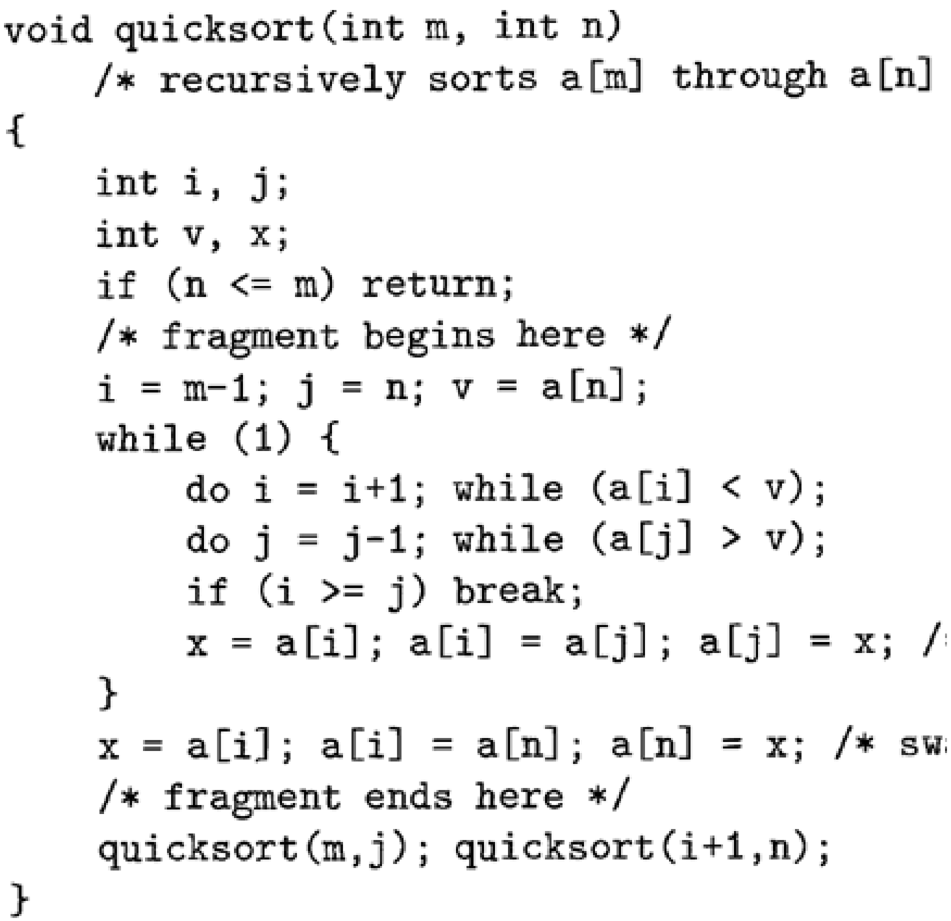
(after optimizing B5 and B6)

initializing for CSE



```
B1:
i = m-1
j = n
t1 = 4*n
v = a[t1]

B2:
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2

B3:
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3

B4:
if i>=j goto B6

B5:
t6=4*i
x=a[t6]
t8=4*j
t9=a[t8]
a[t6]=t9
a[t8]=x
goto B2

B6:
t11=4*i
x=a[t11]
t13=4*n
t14=a[t13]
a[t11]=x
a[t13]=x
```

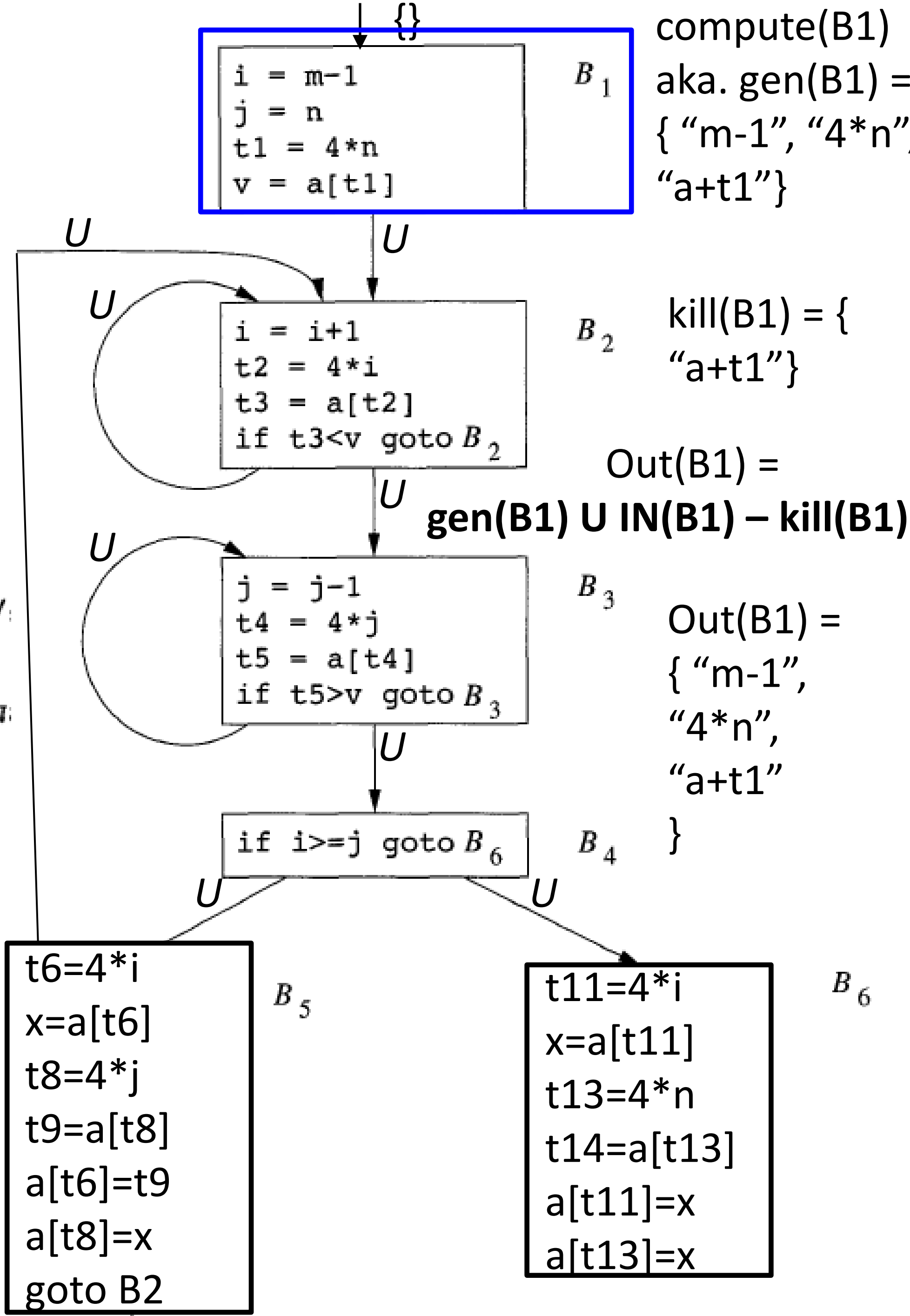
Set U={“m-1”,
“4*n”,
“a+t1”,
“4*i”,
“i+1”,
”a+t2”,
”j-1”,
”4*j”,
”a+t4”,
”a+t6”,
”a+t8”,
”a+t7”,
”a+t11”,
”a+t13”
}



```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

- CFG for quicksort

(after optimizing B5 and B6)



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

```
if i>=j goto B6
```

```
t6=4*i
x=a[t6]
t8=4*j
t9=a[t8]
a[t6]=t9
a[t8]=x
goto B2
```

```
t11=4*i
x=a[t11]
t13=4*n
t14=a[t13]
a[t11]=x
a[t13]=x
```

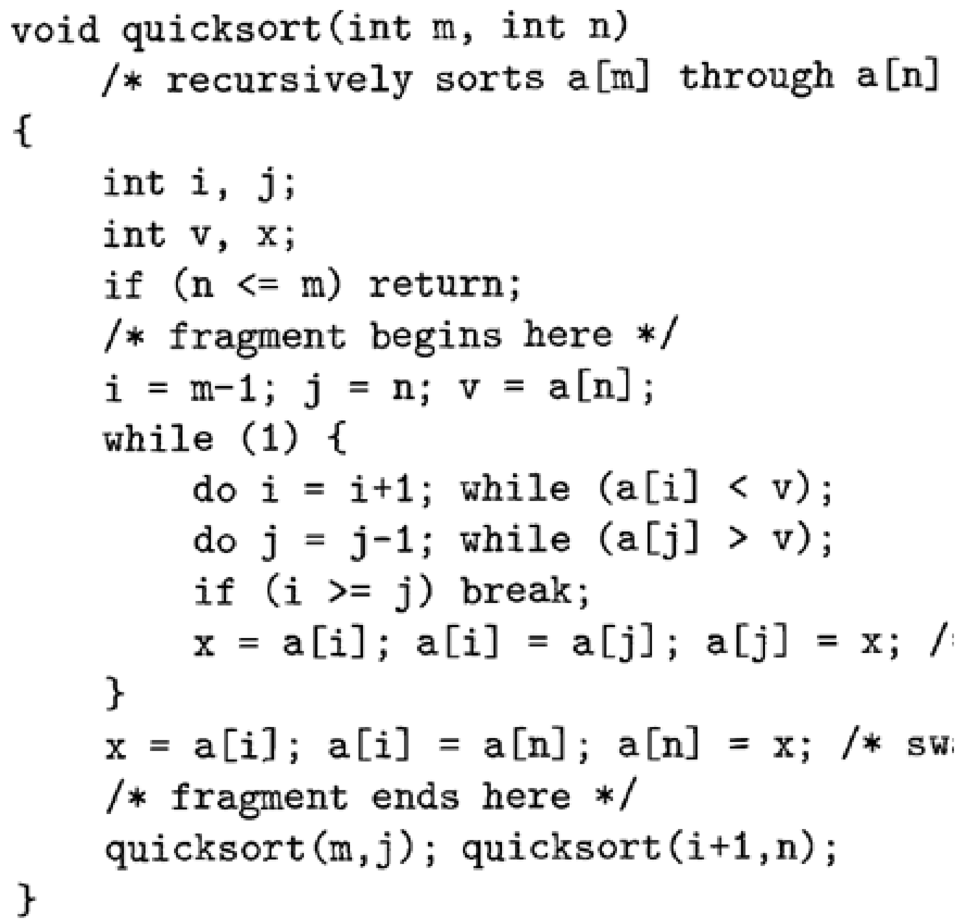
compute(B1) aka. gen(B1) = { "m-1", "4*n", "a+t1"}

kill(B1) = { "a+t1"}

Out(B1) =

gen(B1) U IN(B1) – kill(B1)

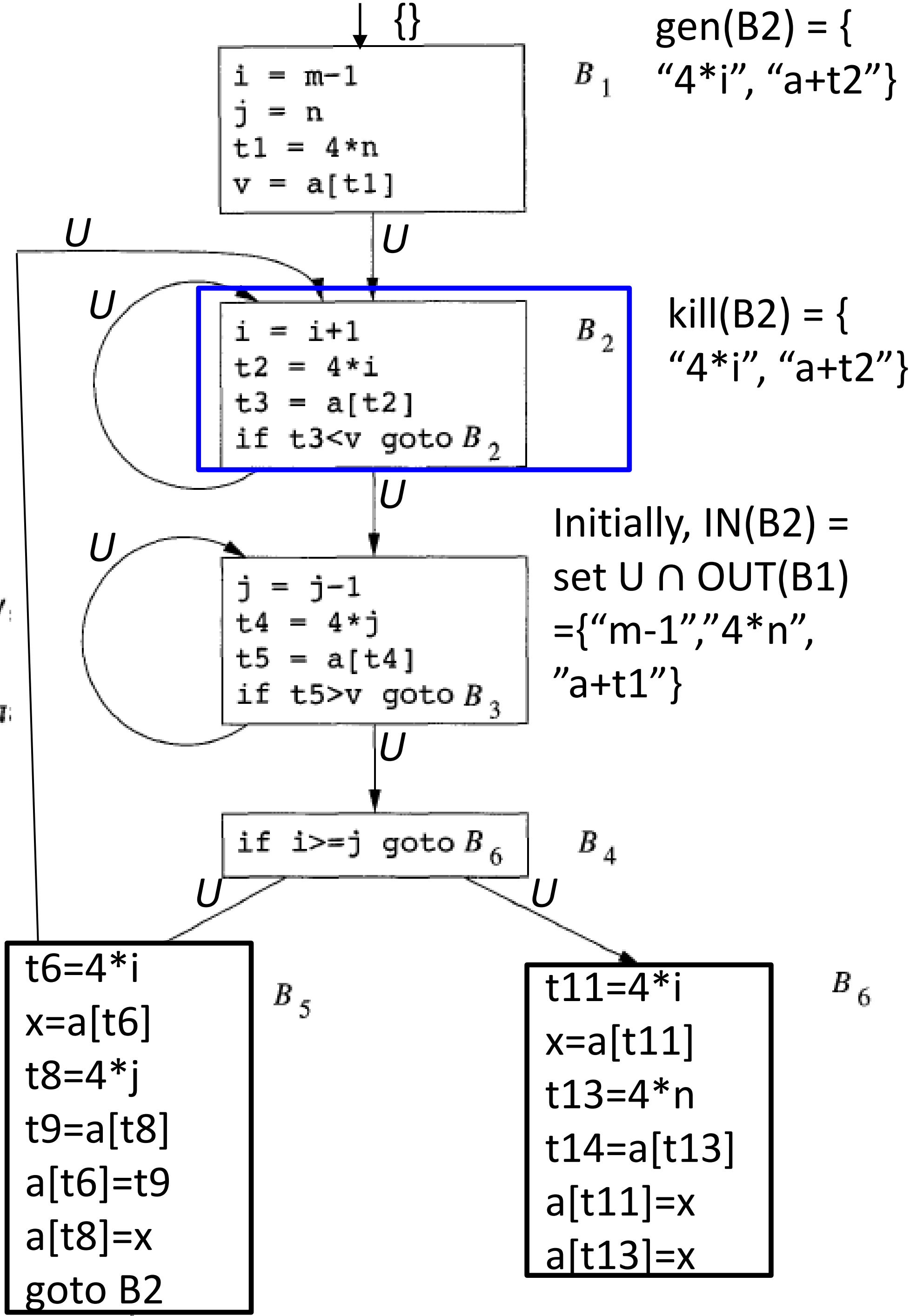
Out(B1) = { "m-1", "4*n", "a+t1" }



```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

- CFG for quicksort

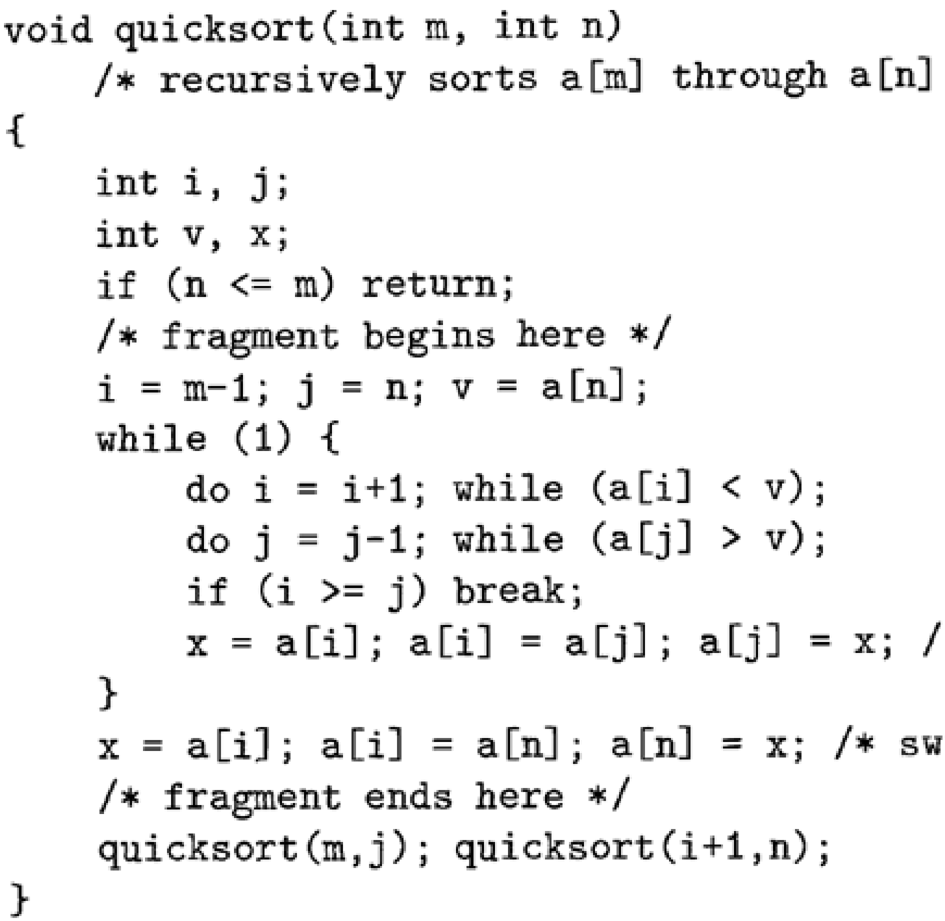
(after optimizing B5 and B6)



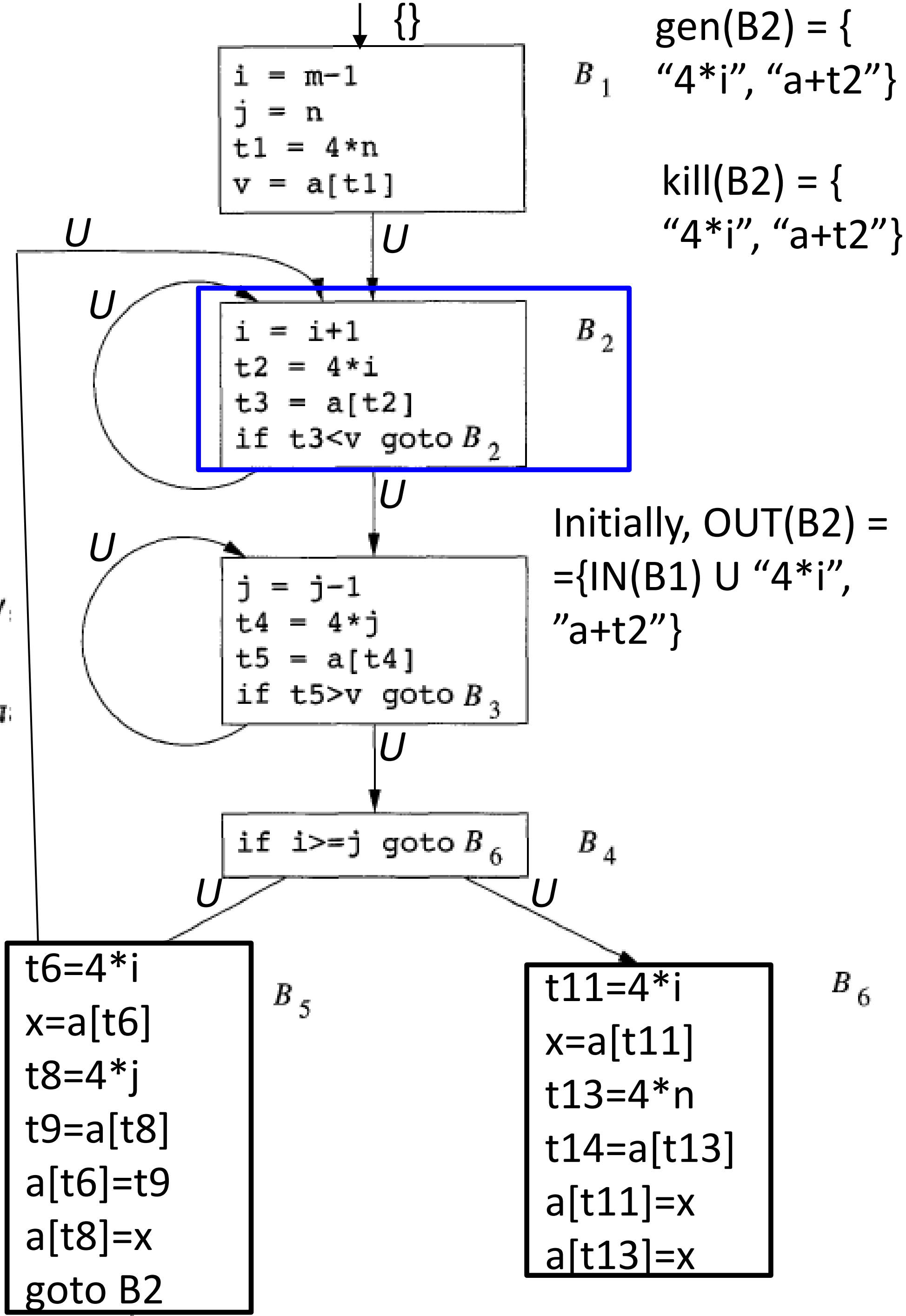
gen(B2) = { “4*i”, “a+t2”}

kill(B2) = { “4*i”, “a+t2”}

Initially, IN(B2) = set U ∩ OUT(B1) ={“m-1”,”4*n”, ”a+t1”}



```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```



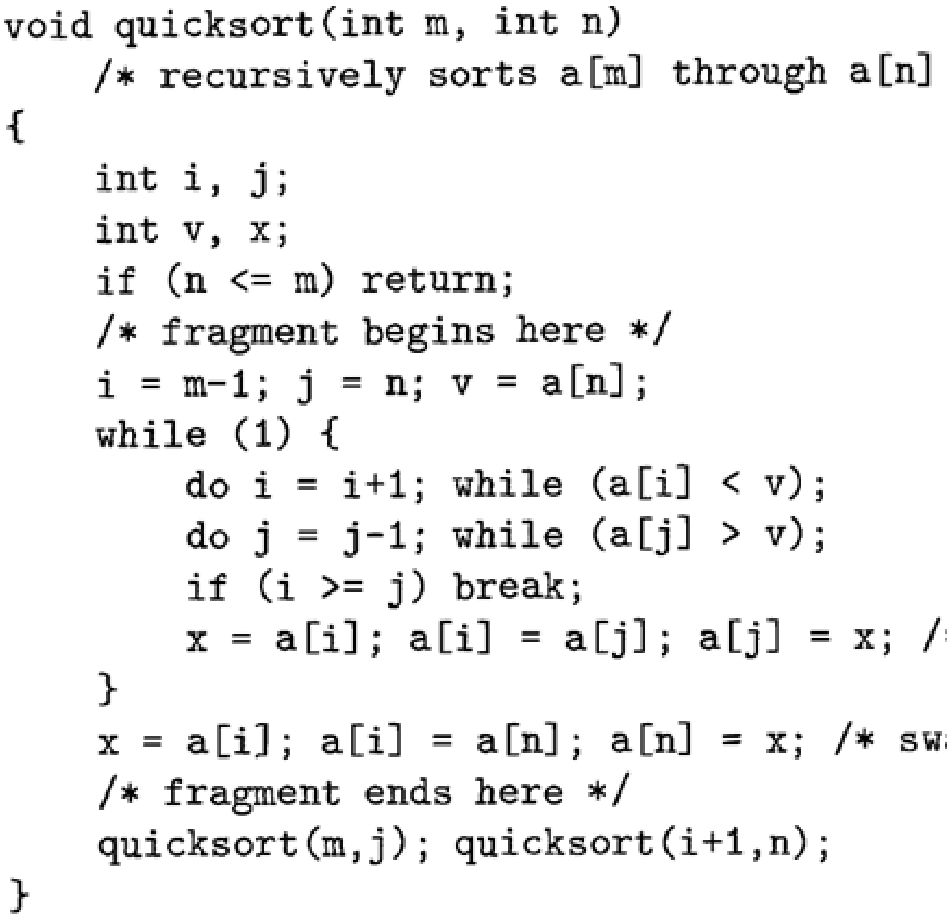
gen(B2) = {
“4*i”, “a+t2”}

kill(B2) = {
“4*i”, “a+t2”}

Initially, OUT(B2) =
={IN(B1) U “4*i”,
”a+t2”}

- CFG for quicksort

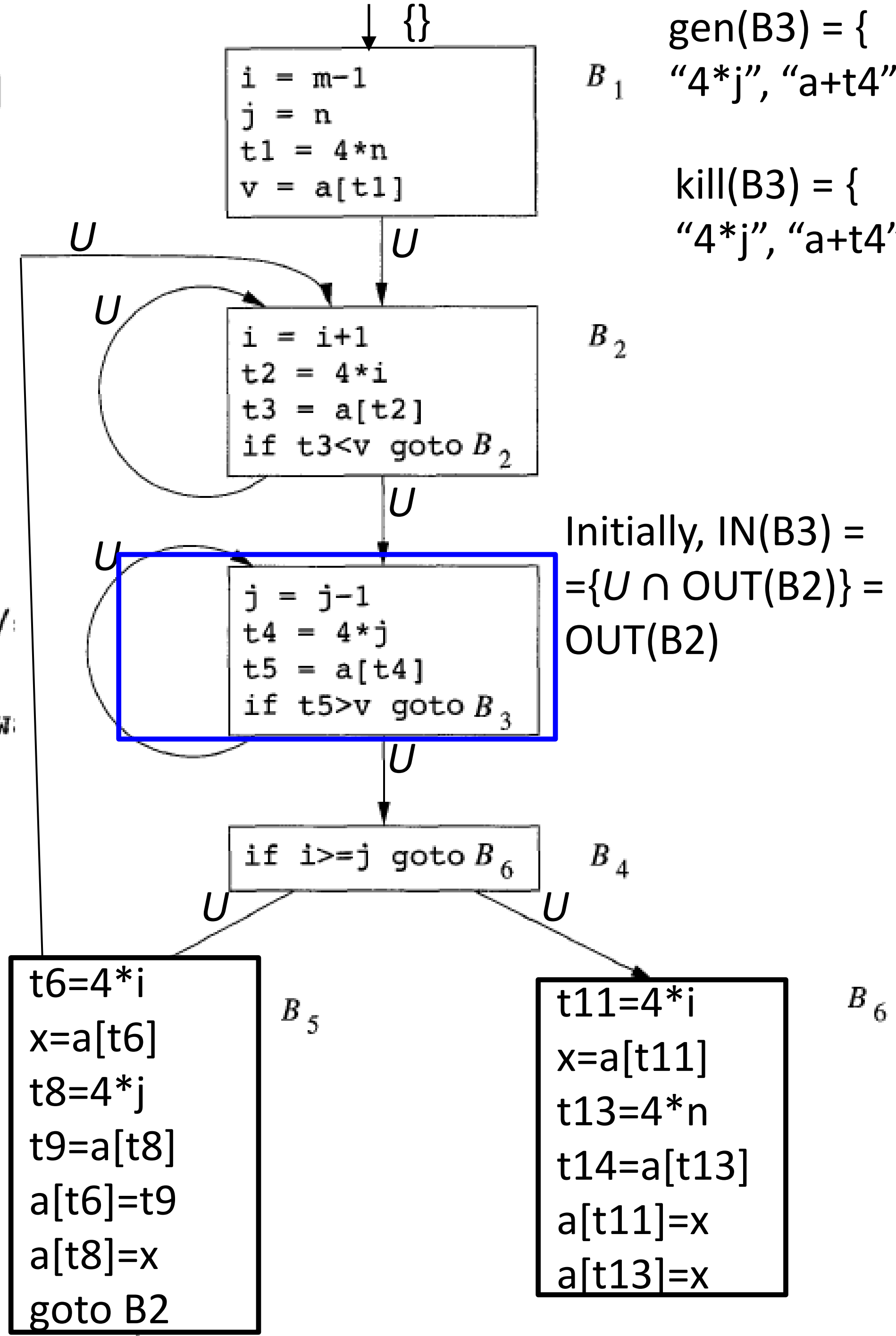
(after optimizing B5 and B6)



```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

gen(B3) = {
“4*j”, “a+t4”}

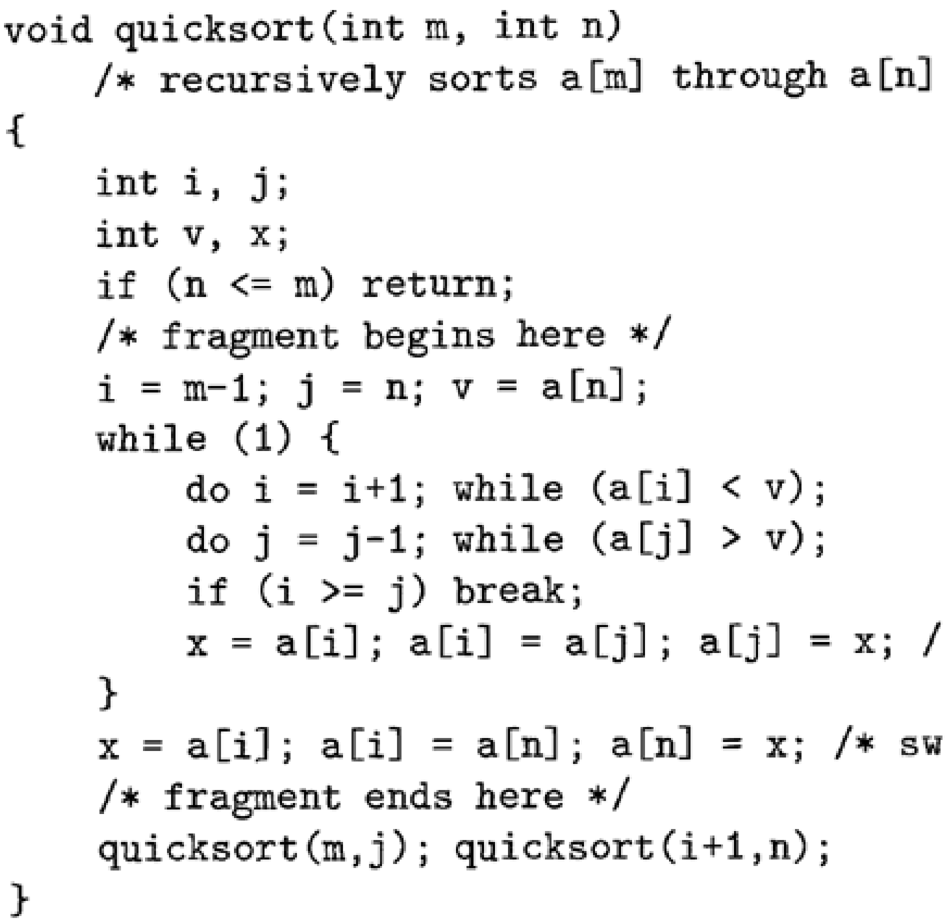
kill(B3) = {
“4*j”, “a+t4”}



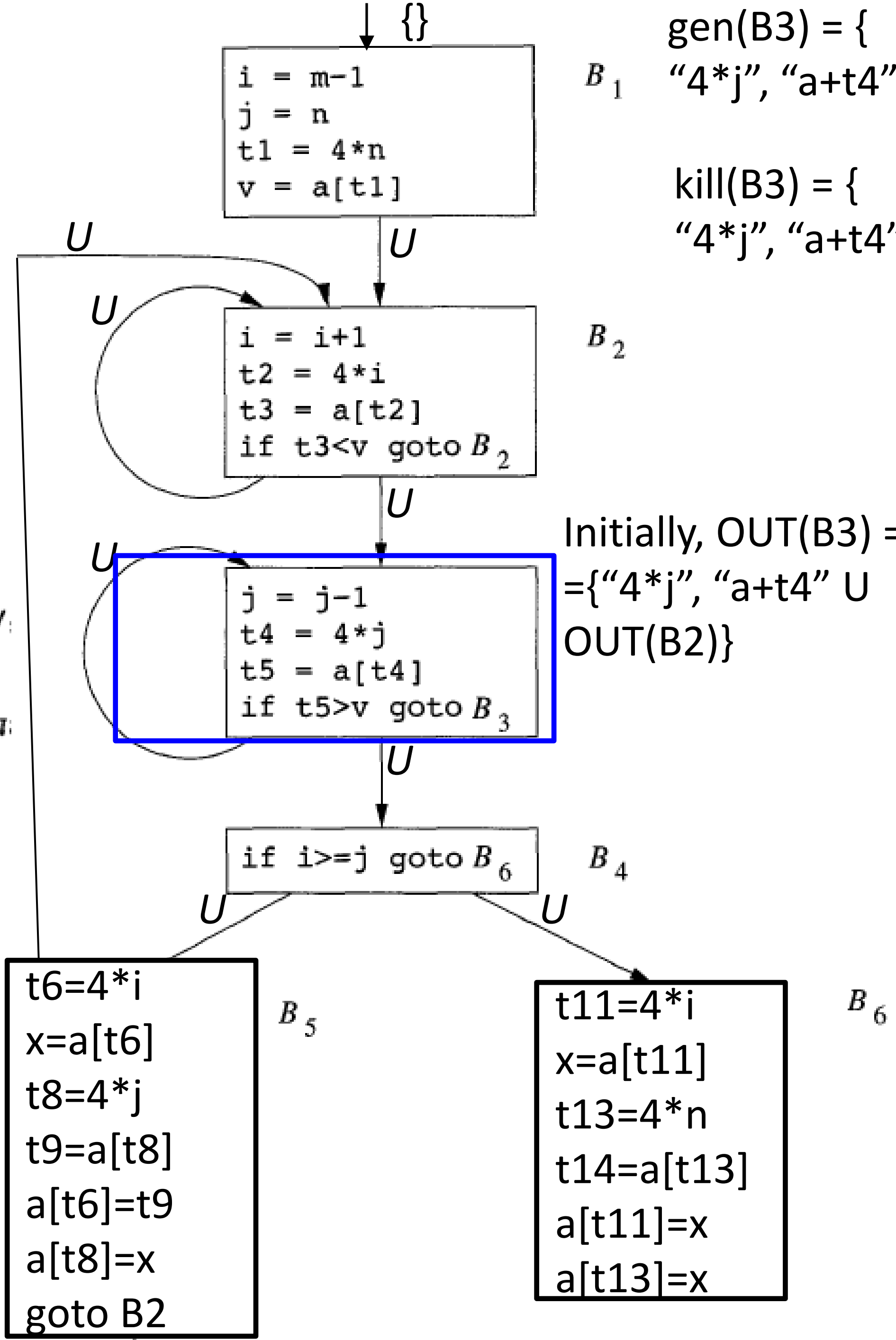
Initially, IN(B3) = ={*U* ∩ OUT(B2)} = OUT(B2)

- CFG for quicksort

(after optimizing B5 and B6)



```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* sw
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```



gen(B3) = {
“4*j”, “a+t4”}

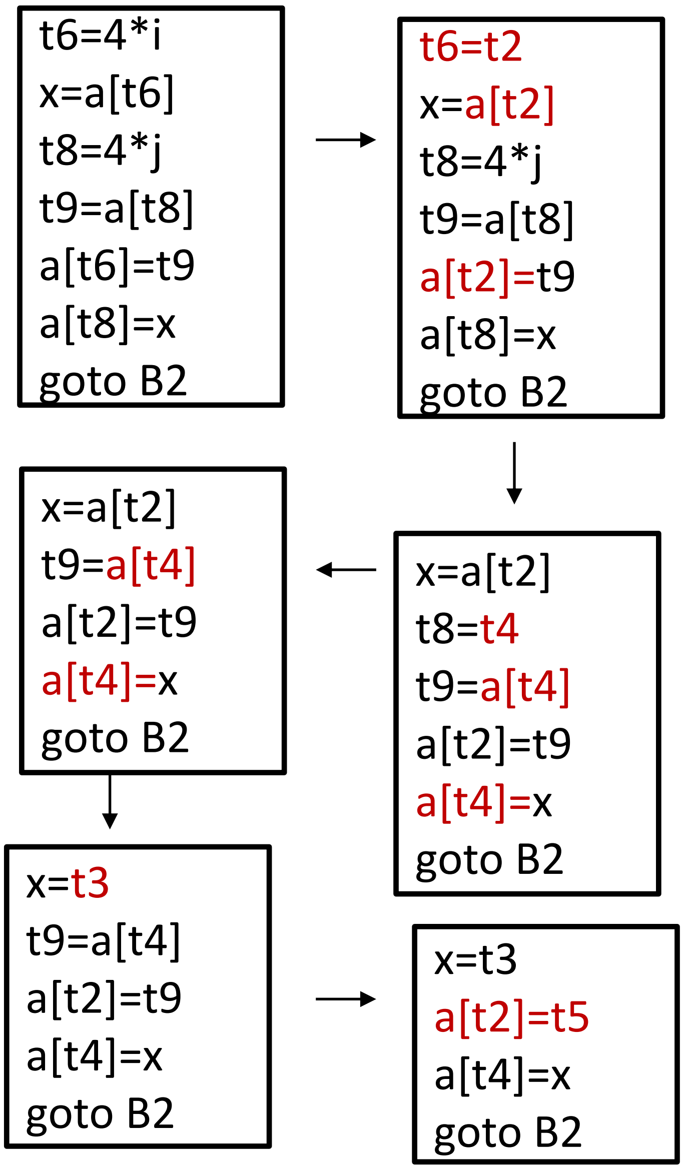
kill(B3) = {
“4*j”, “a+t4”}

Initially, OUT(B3) =
={“4*j”, “a+t4” U
OUT(B2)}

- CFG for quicksort

(after optimizing B5 and B6)

IN(B5)=“4*j”, “a+t4”, “4*i”, “a+t2”, “m-1”, “4*n”, “a+t1”



```
t6=4*i
x=a[t6]
t8=4*j
t9=a[t8]
a[t6]=t9
a[t8]=x
goto B2
```

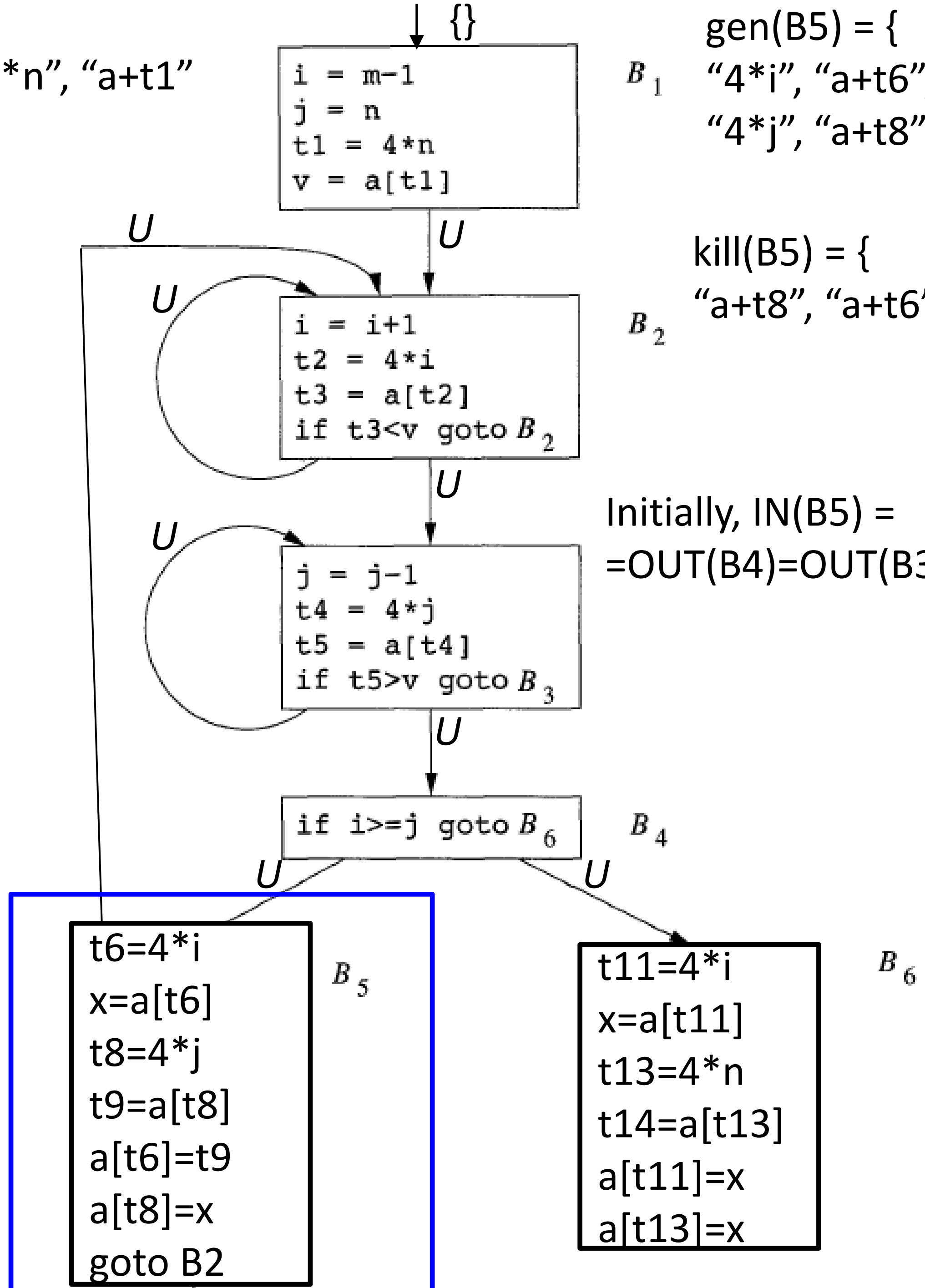
```
t6=t2
x=a[t2]
t8=4*j
t9=a[t8]
a[t2]=t9
a[t8]=x
goto B2
```

```
x=a[t2]
t8=t4
t9=a[t4]
a[t2]=t9
a[t4]=x
goto B2
```

```
x=a[t2]
t9=a[t4]
a[t2]=t9
a[t4]=x
goto B2
```

```
x=t3
t9=a[t4]
a[t2]=t9
a[t4]=x
goto B2
```

```
x=t3
a[t2]=t5
a[t4]=x
goto B2
```



```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

```
if i>=j goto B6
```

```
t6=4*i
x=a[t6]
t8=4*j
t9=a[t8]
a[t6]=t9
a[t8]=x
goto B2
```

```
t11=4*i
x=a[t11]
t13=4*n
t14=a[t13]
a[t11]=x
a[t13]=x
```

gen(B5) = {
“4*i”, “a+t6”,
“4*j”, “a+t8”}

kill(B5) = {
“a+t8”, “a+t6”}

Initially, IN(B5) =
=OUT(B4)=OUT(B3)

Dataflow Analysis – Problem Categorization

- All path problem:
  - we want the property to hold at all the paths reaching a program point.
- Any path problem:
  - we want the property to hold at some path reaching a program point.

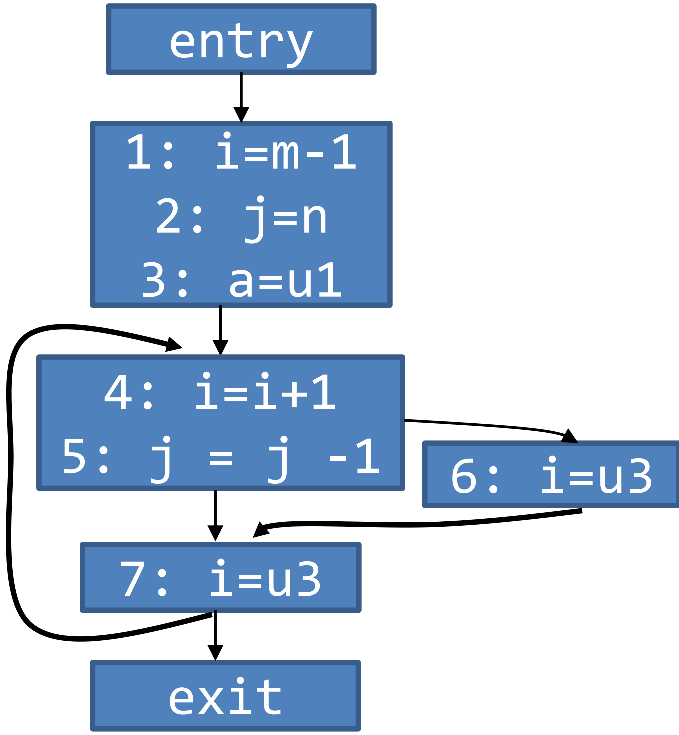
Orthogonal to the above categorization we can have:

- Forward flow problem:
  - Transfer of information done along the direction of the control flow
- Backward flow problem:
  - Transfer of information done opposite to the direction of the control flow

Reaching Definitions - Example

- **Goal:** to know where in a program each variable x may have been defined when control reaches block b
- Definition d reaches block b if there is a path from point immediately following d to b, such that the variable defined in d is not redefined / killed along that path

$$\mathtt{In(b)} = \cup_{i \in Pred(b)} Out(i)$$



entry

1: i=m-1
2: j=n
3: a=u1

4: i=i+1
5: j = j -1

6: i=u3

7: i=u3

exit

$$\mathtt{Out(b)} = \mathtt{gen(b)} \cup (\mathtt{In(b)} - \mathtt{kill(b)})$$

```
//set that contains all statements
that may define some variable x in
b. E.g. gen(1:a=3;2:a=4)={2}
```

```
//set that contains all statements
that define a variable x that is
also defined in b. E.g.
kill(1:a=3; 2:a=4)={1,2}
```

Reaching definitions

- What definitions of a variable *reach* a particular program point
    - A definition of variable x from statement s reaches a statement t if there is a path from s to t where x is not redefined
- Especially important if x is used in t
    - Used to build *def-use* chains and *use-def* chains, which are key building blocks of other analyses
        - Used to determine dependences: if x is defined in s and that definition reaches t then there is a flow dependence from s to t
    - We used this to determine if statements were loop invaraint
        - All definitions that reach an expression must originate from outside the loop, or themselves be invariant

Creating a reaching-def analysis

- Can we use a powerset lattice?
- At each program point, we want to know which definitions have reached a particular point
  - Can use powerset of set of definitions in the program
    - V is set of variables, S is set of program statements
    - Definition: $d \in V \times S$
      - Use a tuple, <v, s>
  - How big is this set?
    - At most $|V \times S|$ definitions

Forward or backward?

- What do you think?

Choose confluence operator

- Remember: we want to know if a definition *may* reach a program point
- What happens if we are at a merge point and a definition reaches from one branch but not the other?
    - We don't know which branch is taken!
    - We should union the two sets – any of those definitions can reach
- We want to avoid getting too many reaching definitions → should start sets at $\perp$

Transfer functions for RD

- Forward analysis, so need a slightly different formulation
  - Merged data flowing into a statement

$$\begin{aligned} IN(s) &= \bigcup_{t \in pred(s)} OUT(t) \\ OUT(s) &= \mathbf{gen}(s) \cup (IN(s) - \mathbf{kill}(s)) \end{aligned}$$

- What are gen and kill?
  - gen(s): the set of definitions that *may* occur at s
    - e.g., gen(s_1: x = e) is <x, s_1>
  - kill(s): all previous definitions of variables that are *definitely* redefined by s
    - e.g., kill(s_1: x = e) is <x, *>