CS620T: Compilers – Principles and Implementation

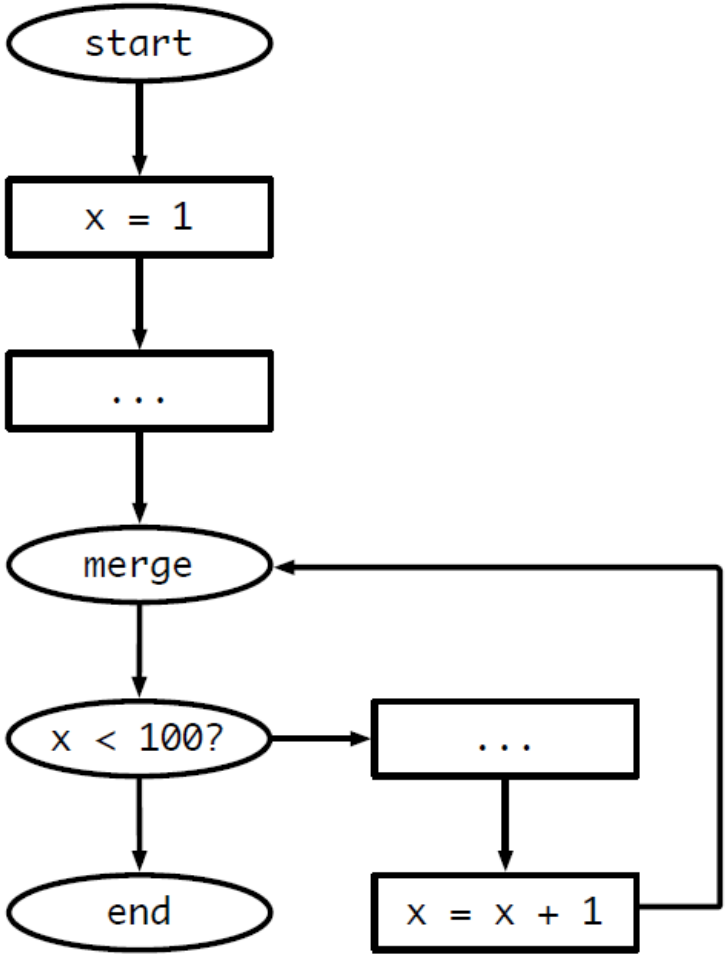
Autumn 2026

Week 13: Dataflow Analysis (Constant Propagation (contd..), Reaching Definitions, Available Expressions)

What do we do about loops?

- Unless a loop never executes, symbolic execution looks like it will keep going around to the same nodes over and over again
- Insight: if the input state vector(s) for a node don't change, then its output doesn't change
  - If input stops changing, then we are done!
- Claim: input will eventually stop changing. Why?

Loop example



Complexity of algorithm

- V = # of variables, E = # of edges
- Height of lattice = 2 → each state vector can be updated at most 2 *V times.
- So each edge is processed at most 2 *V times, so we process at most 2 * E *V elements in the worklist.
- Cost to process a node: O(V)
- Overall, algorithm takes $O(EV^2)$ time

Question

- Can we generalize this algorithm and use it for more analyses?

Constant propagation

- Step 1: choose lattice (which values are you going to track during symbolic execution)?
  - Use constant lattice
- Step 2: choose direction of dataflow (if executing symbolically, can run program backwards!)
  - Run forward through program
- Step 3: create *transfer functions*
  - How does executing a statement change the symbolic state?
- Step 4: choose *confluence operator*
  - What do do at merges? For constant propagation, use join

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

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

- 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, *>

Generalization (Recap)

- **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 gen(b), kill(b), IN(b), OUT(b)
- **Initializations?**

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?

$$IN(S) = \cap_{t \in pred(s)} OUT(t)$$

$$OUT(S) = \mathbf{gen}(s) \cup (IN(S) - \mathbf{kill}(s))$$

- 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 $\rightarrow \bot$
  - Available expressions: an expression that *was definitely* computed earlier
    - We want to have as many available expressions as possible $\rightarrow \top$
  - 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

{}	t6 = 4*i
{"4*i"}	x = a[t6]
S_1 ={"4*i", "a+t6"}	t7 = 4*i
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
set S_3	t10 = 4*j
set S_3	a[t10] = x
set S_3	

Can be rewritten:

t7 = t6

a[t6] = t9

t10 = t8

a[t8] = x

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

{}	t6 = 4*i	apply dead-code elim.
{"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

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);
}
```

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

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
```

```
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):

```
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
```

after dead-code elim.

~~t12=t11~~

a[t11]=x

~~t15=t13~~

a[t13]=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);
}
```

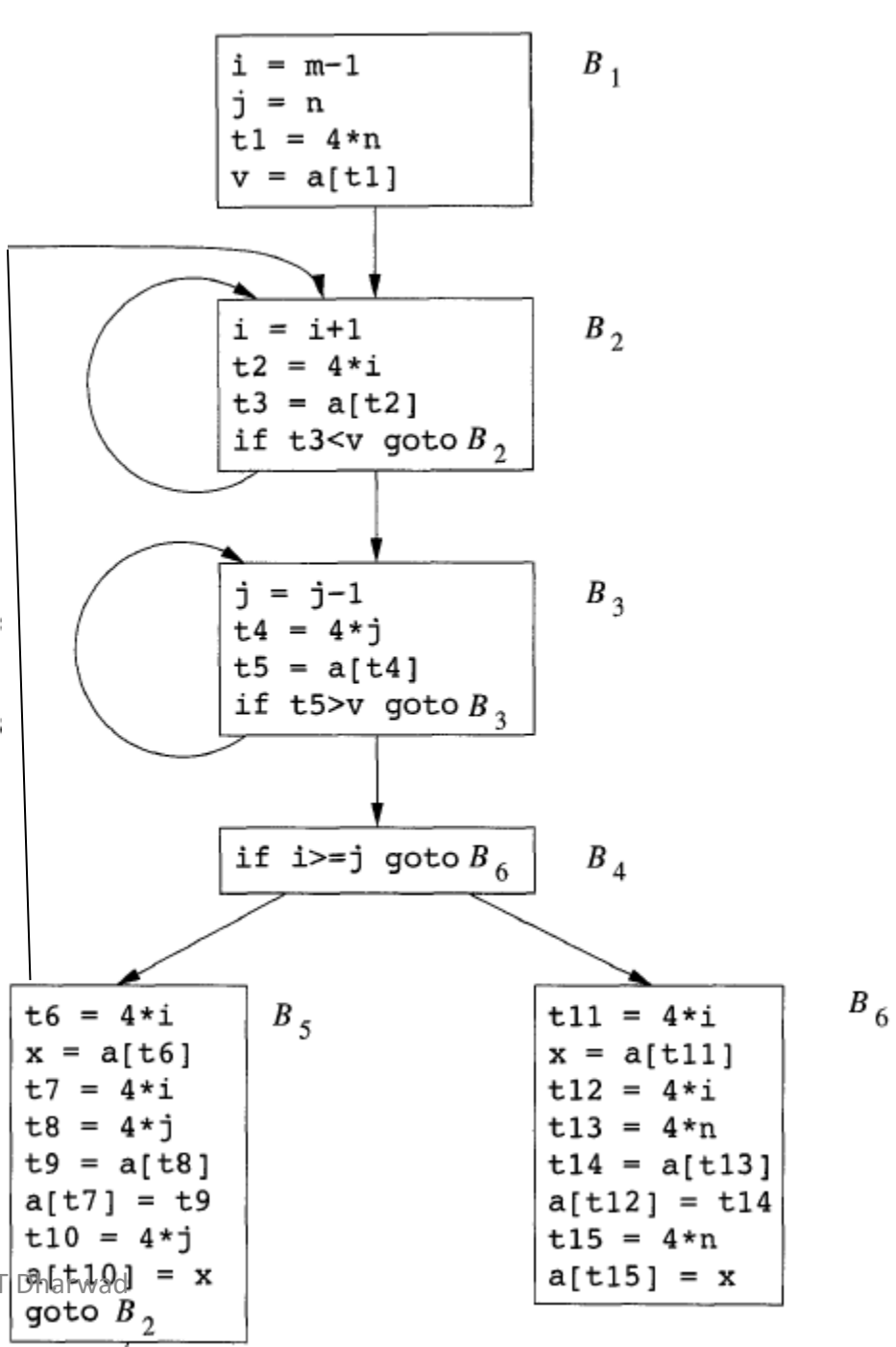
Intermediate code
(after local CSE+copy prop.+dead-code elim.)

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

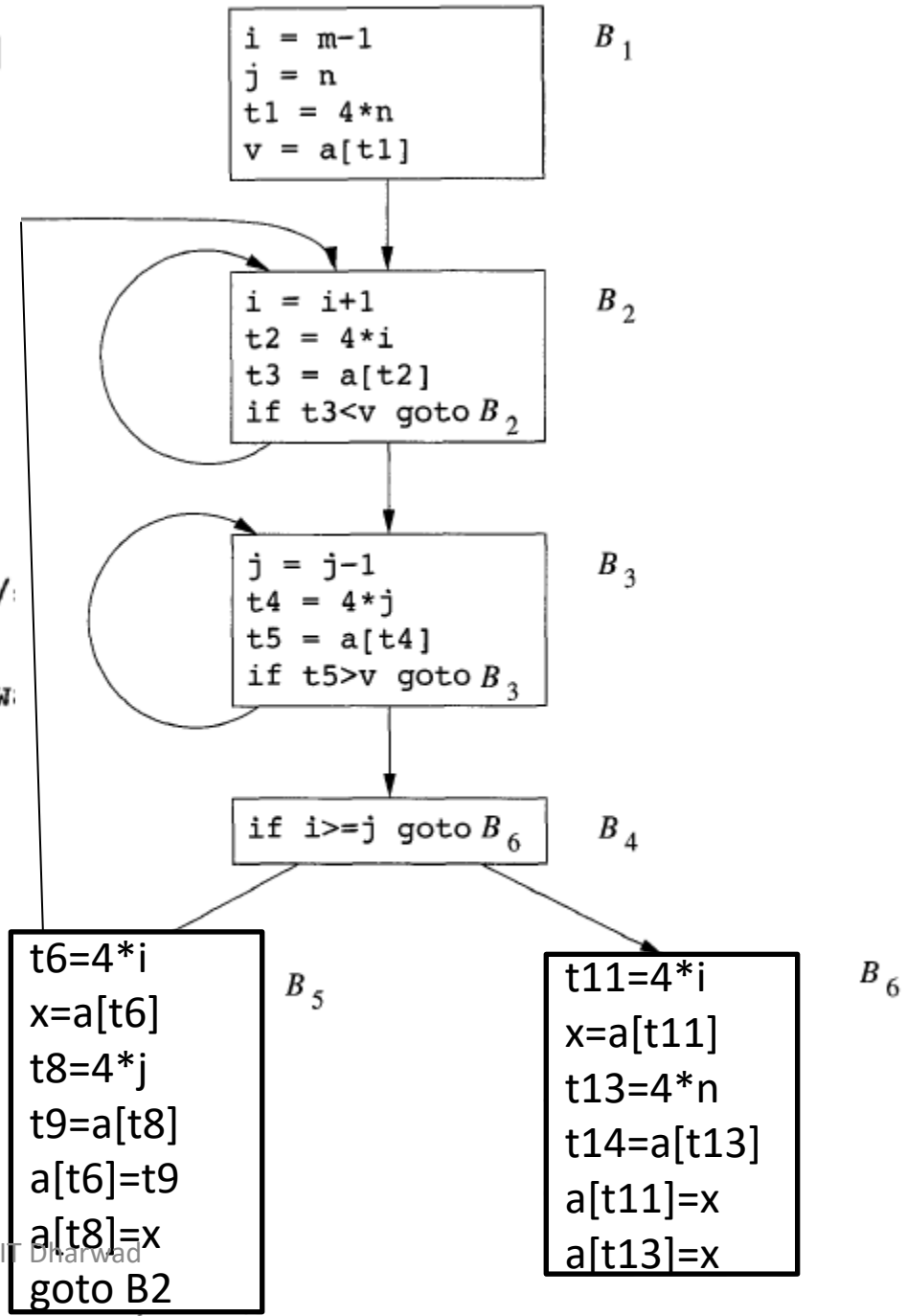
t11=4*I
x=a[t11]
t13=4*n
t14=a[t13]
a[t11]=x
a[t13]=x

```
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);
}
```

- CFG for quicksort



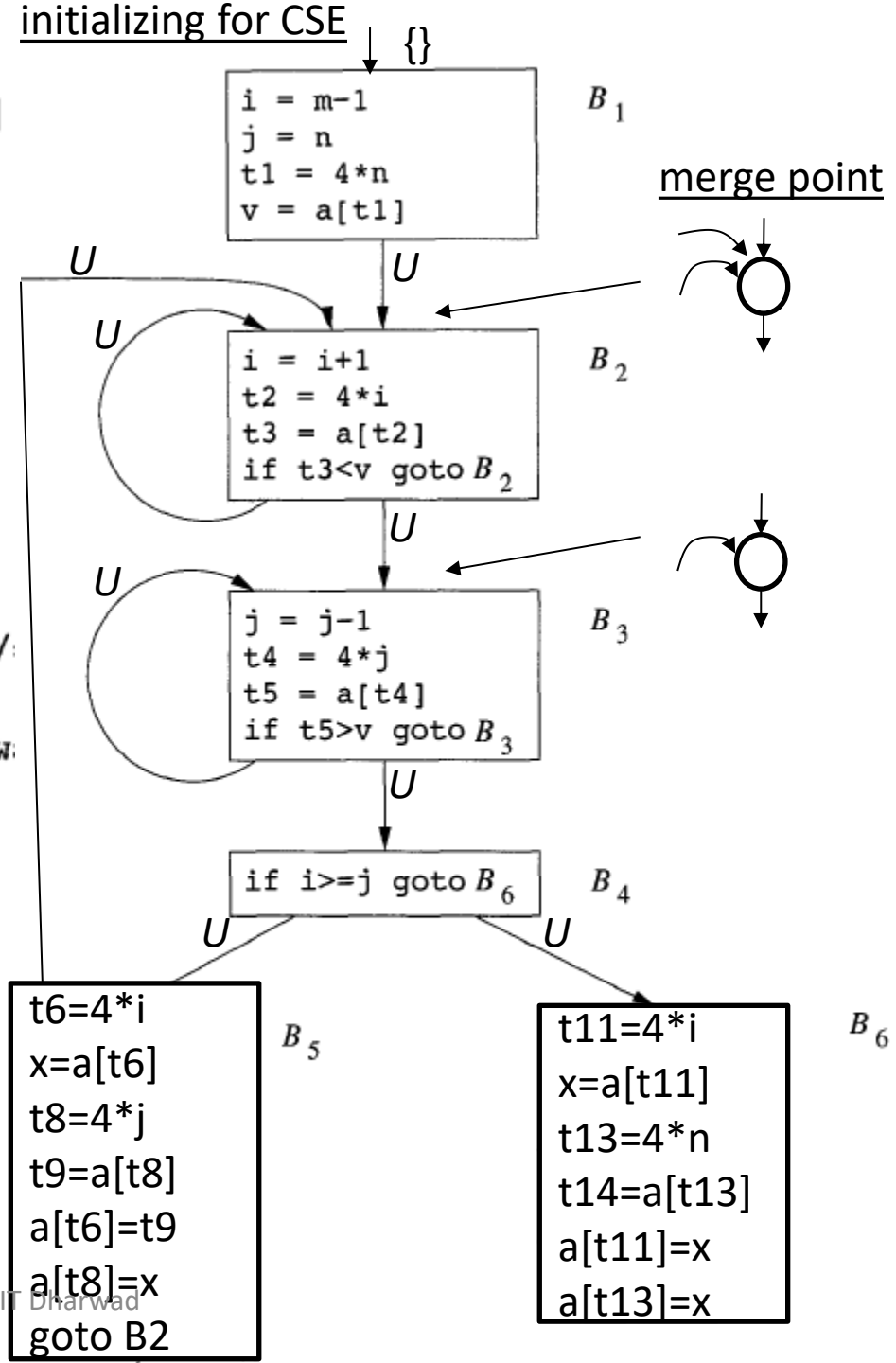
```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j)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

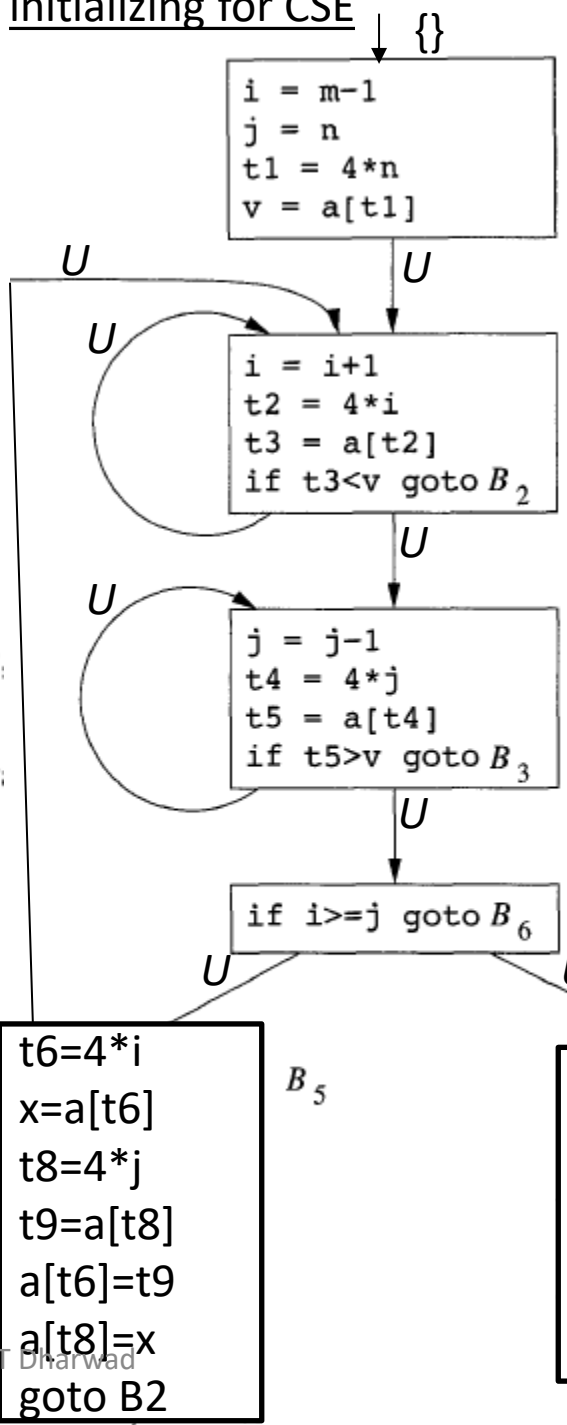


- CFG for quicksort

(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

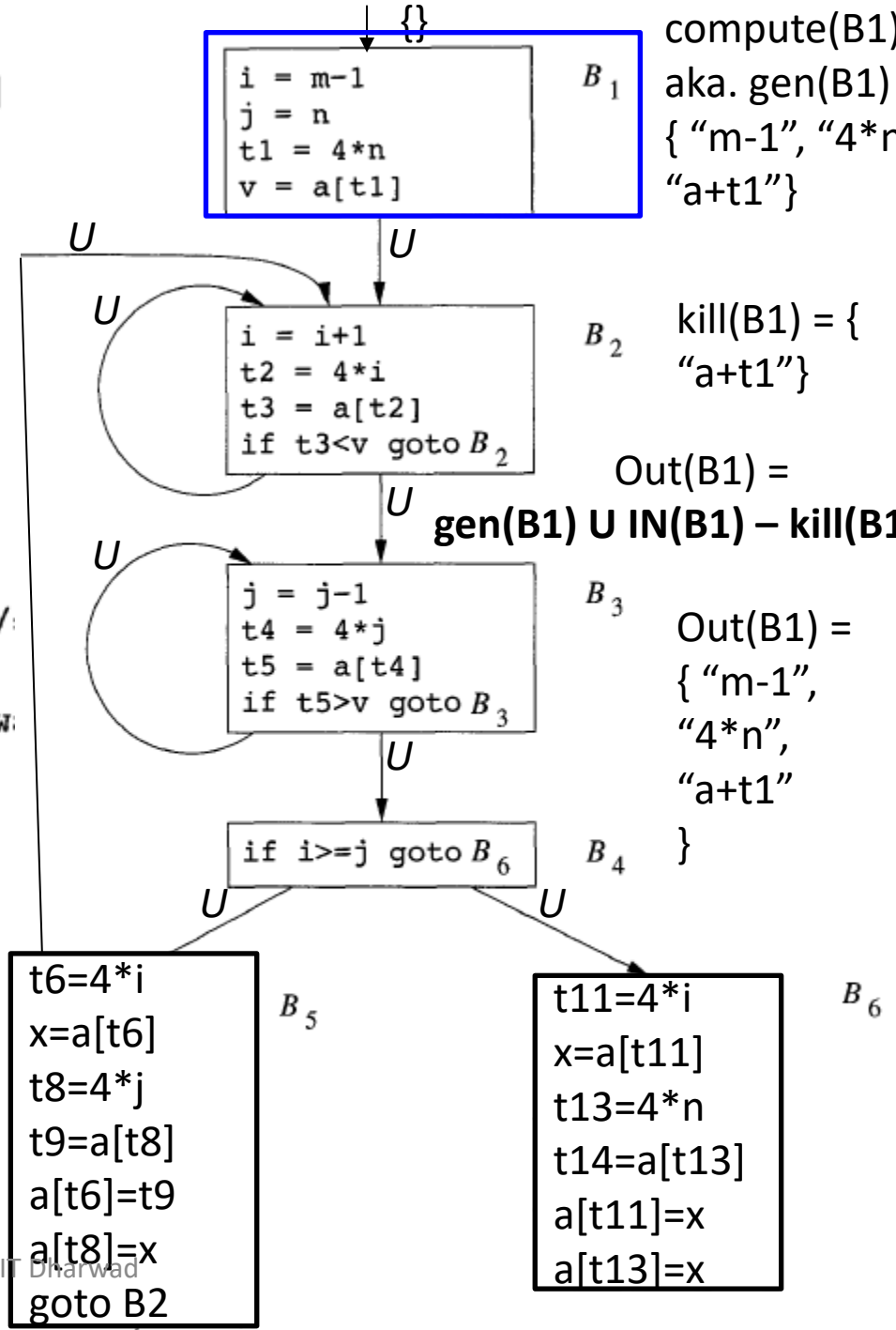


Set U={"m-1",
"4*n",
"a+t1",
"4*i",
"i+1",
"a+t2",
"j-1",
"4*j",
"a+t4",
"a+t6",
"a+t8",
"a+t7",
"a+t11",
"a+t13"
}

- 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);
}
```



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

kill(B1) = { “a+t1”}

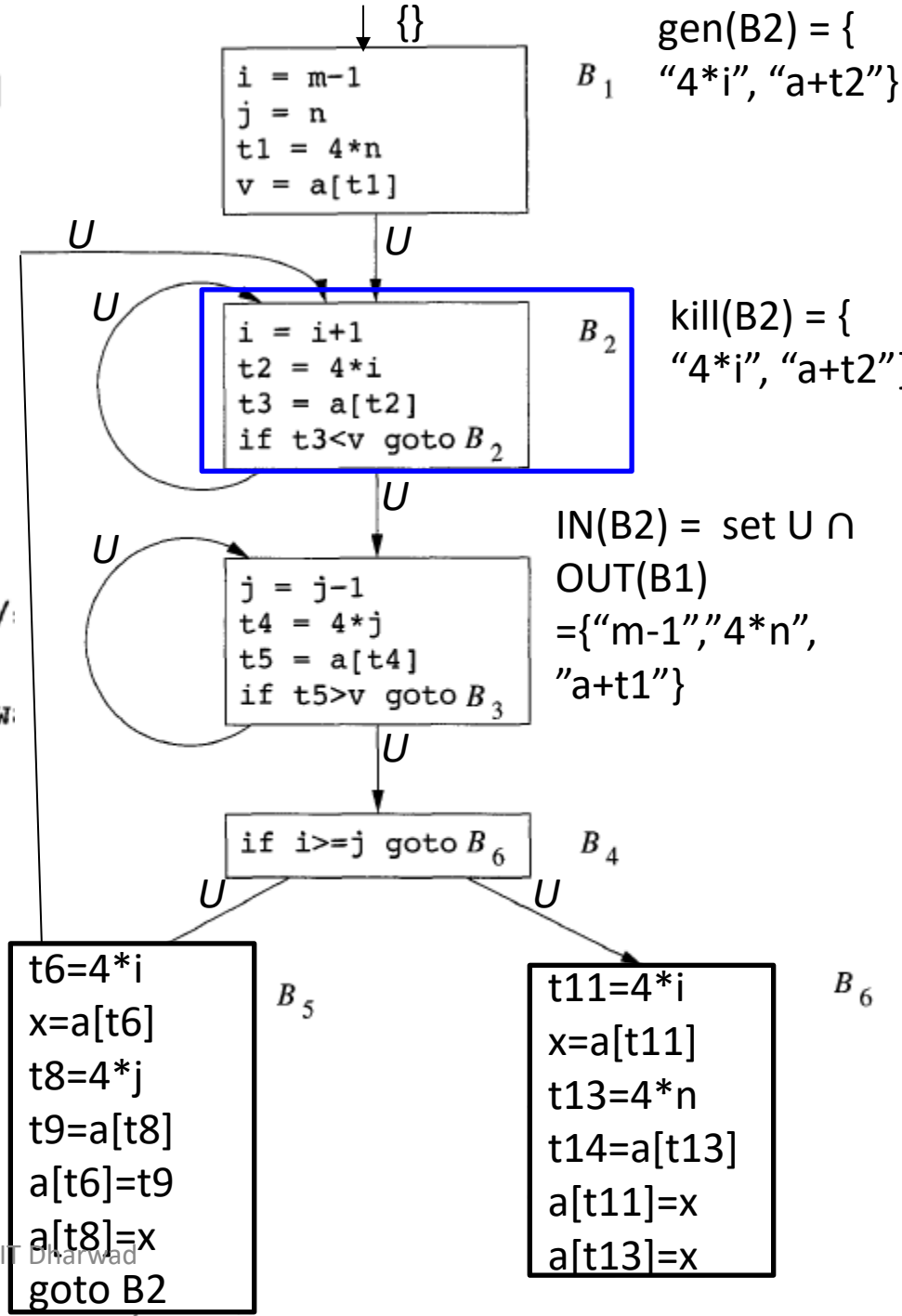
Out(B1) = **gen(B1) U IN(B1) – kill(B1)**

Out(B1) = { “m-1”, “4*n”, “a+t1” }

- CFG for quicksort

(after optimizing B5 and B6)

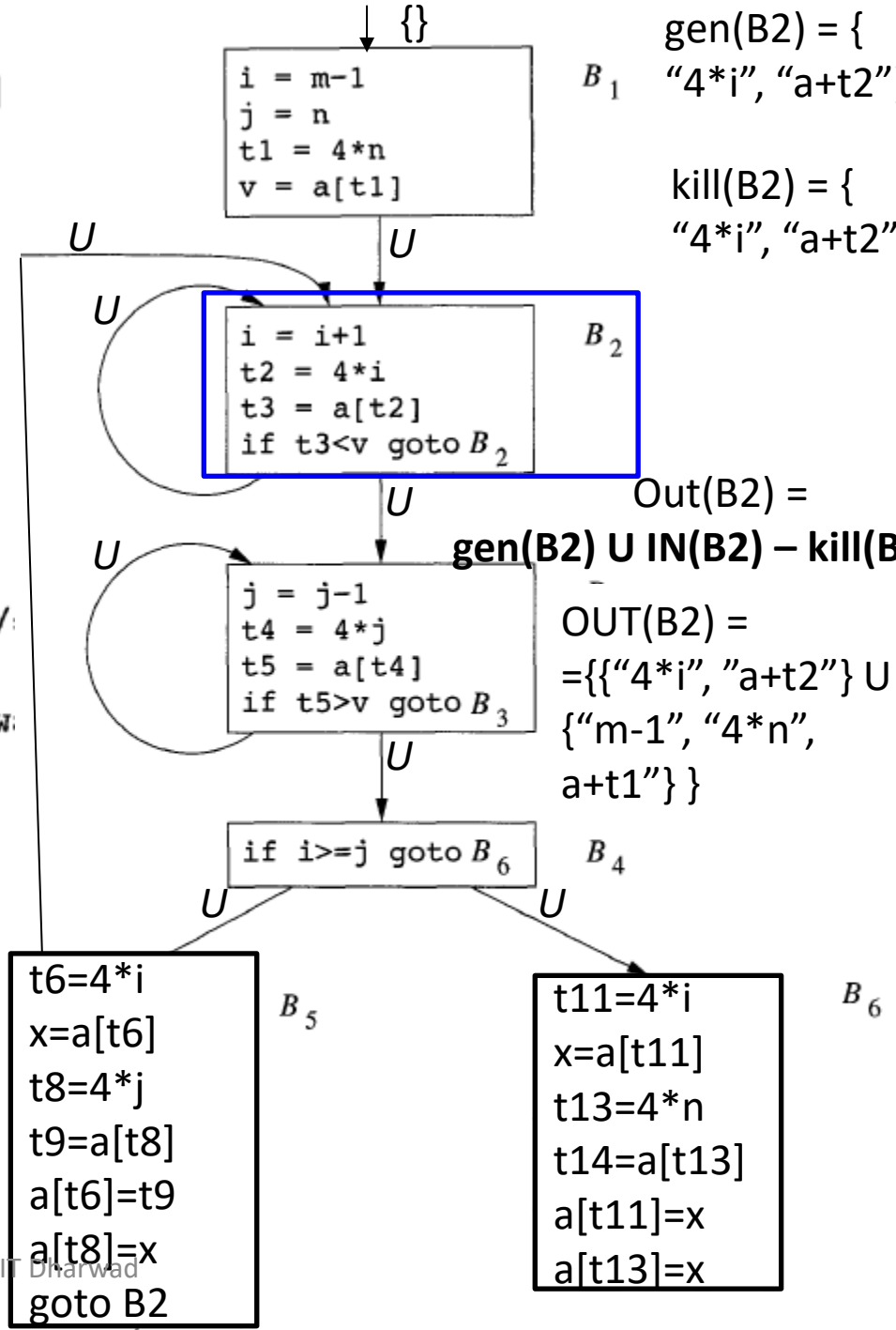
```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n]
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j)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

```
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"}

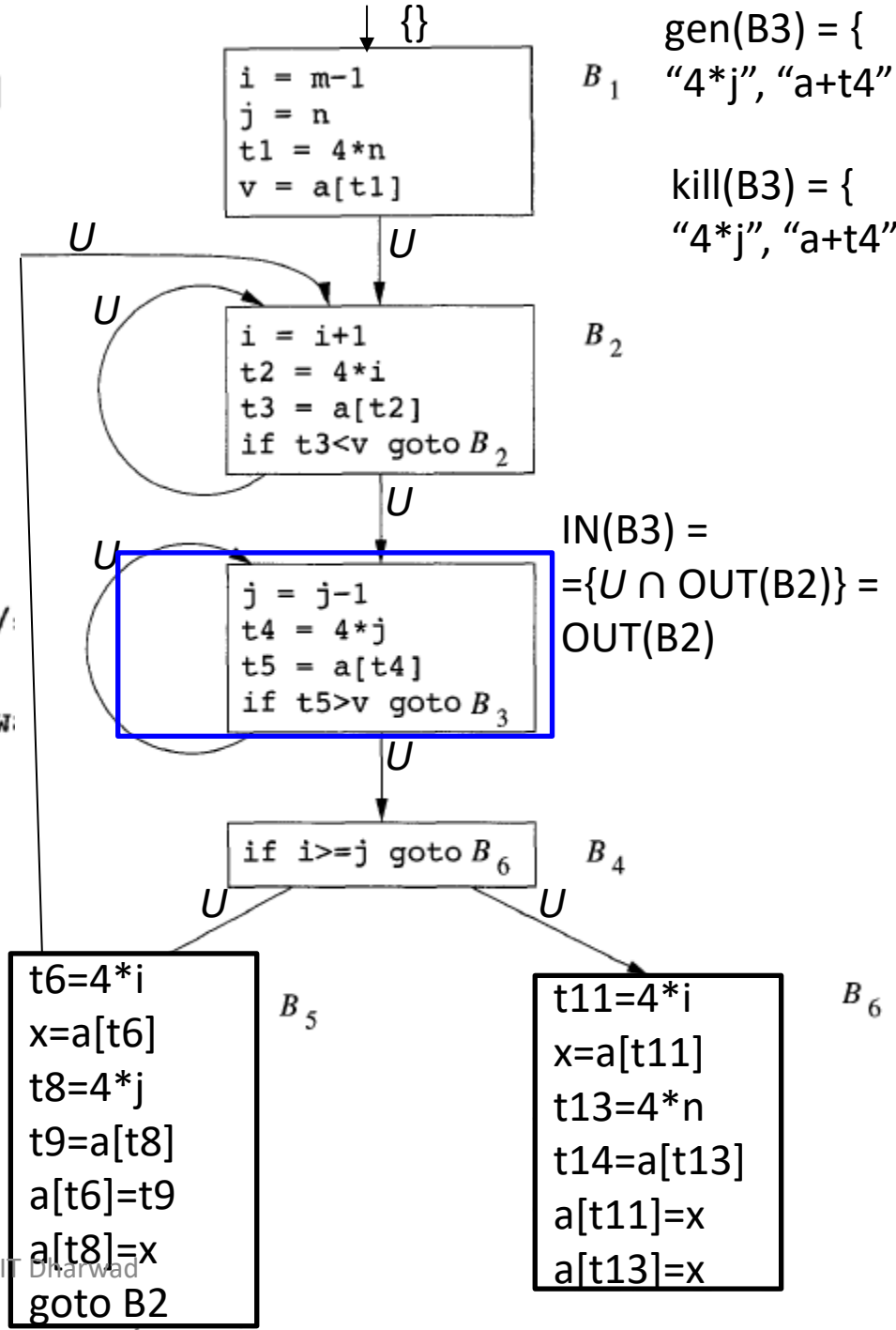
Out(B2) = **gen(B2) U IN(B2) – kill(B2**

OUT(B2) = ={{"4*i", "a+t2"} U {"m-1", "4*n", a+t1"} }

- 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"}

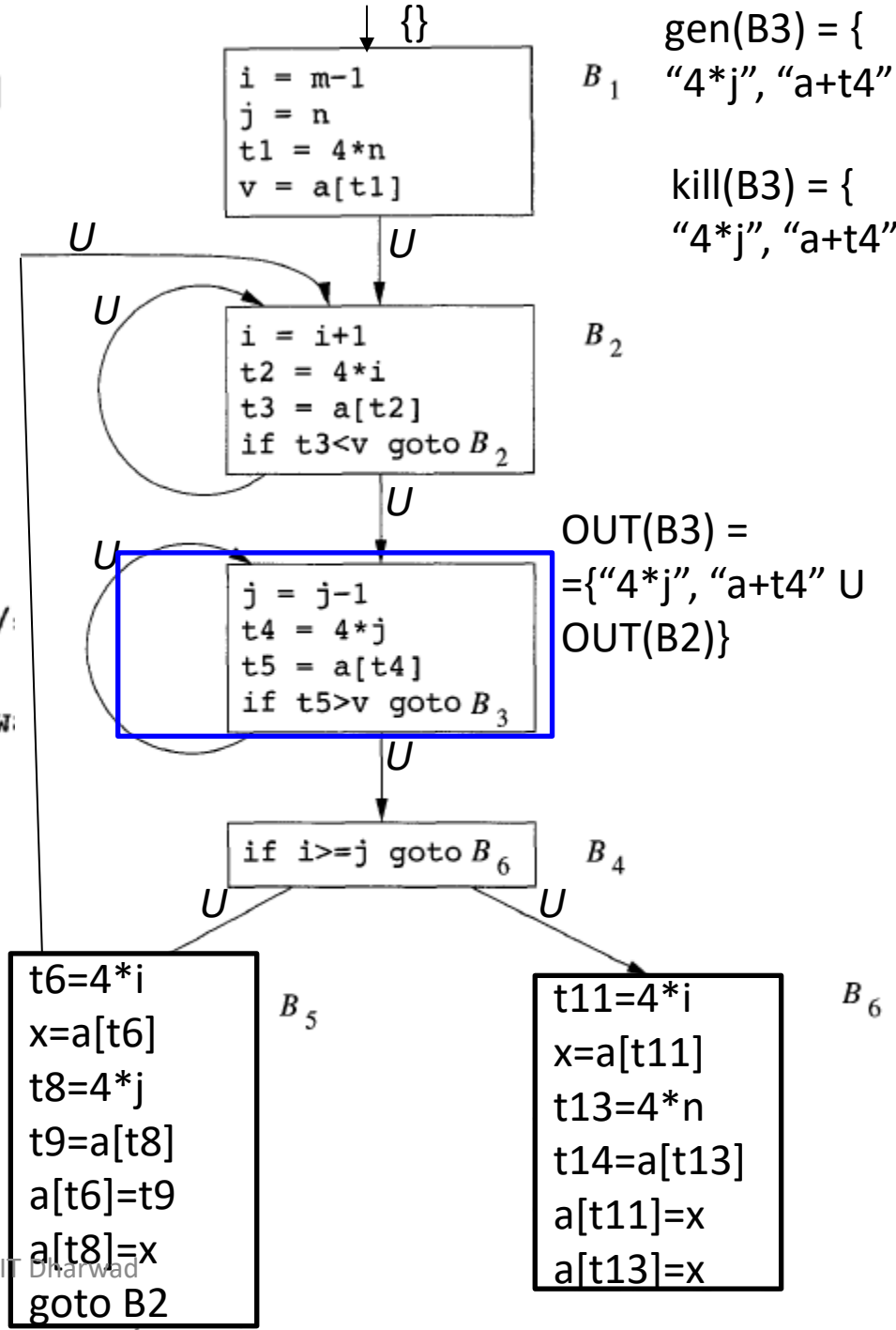
kill(B3) = { "4*j", "a+t4"}

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"}

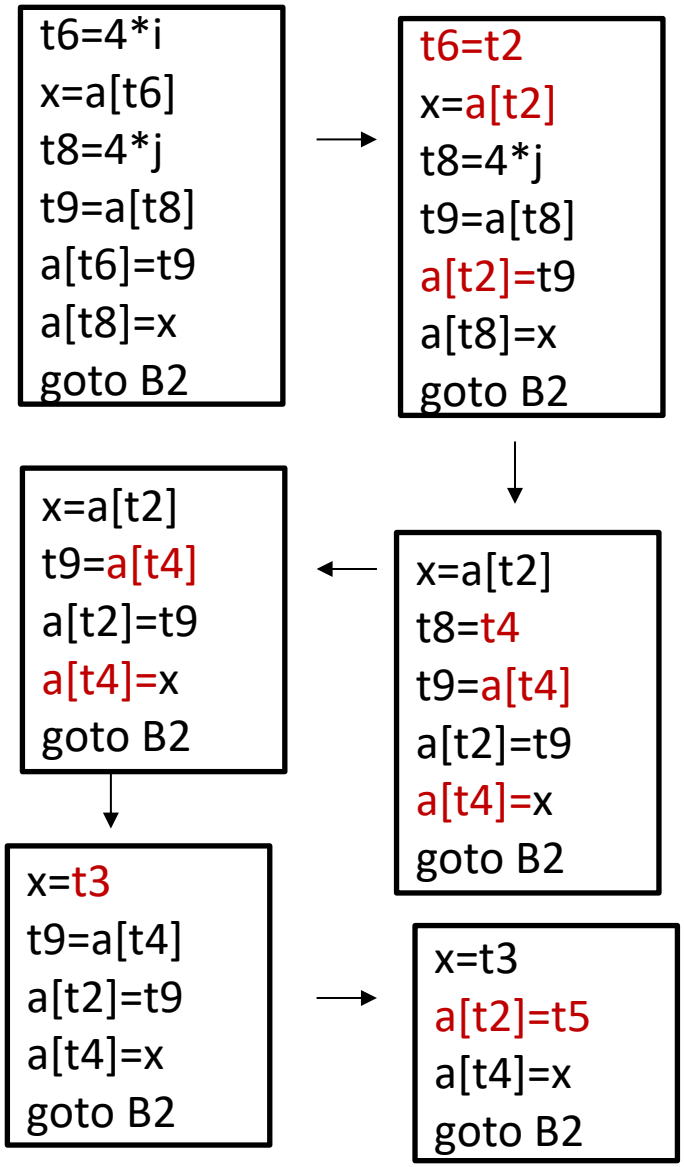
kill(B3) = { "4*j", "a+t4"}

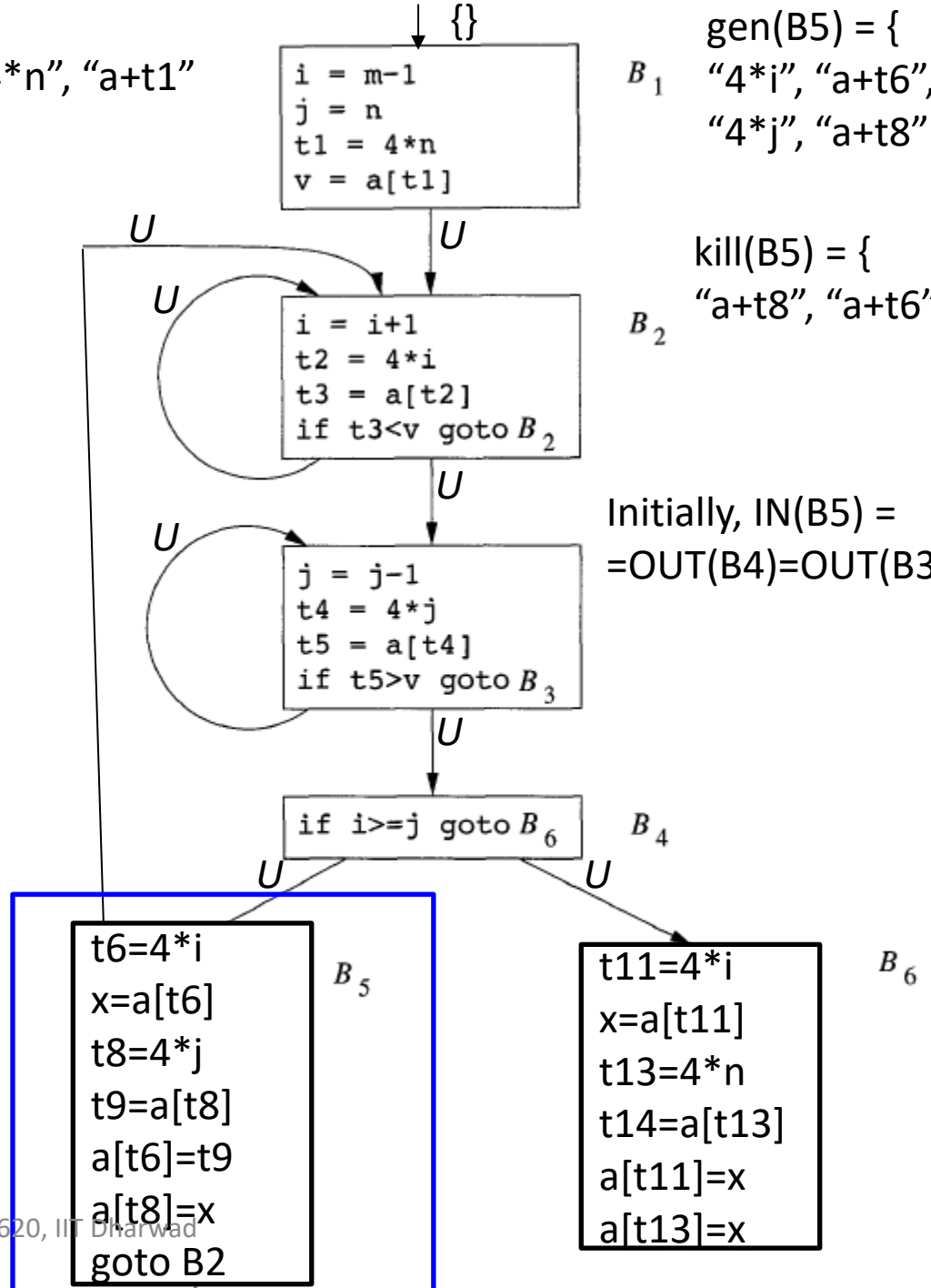
OUT(B3) = ={"4*j", "a+t4" U OUT(B2)}

- CFG for quicksort

(after optimizing B5 and B6)

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





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

Exercise

- Analysis of uninitialized variables

  - What is the direction of analysis?
  - What is the transfer function?