Unit13
Code Optimization

Introduction

- Criteria for Code-Improving Transformation:
  - Meaning must be preserved (correctness)
  - Speedup must occur on average.
  - Work done must be worth the effort.

- Opportunities:
  - Programmer (algorithm, directives)
  - Intermediate code
  - Target code

Peephole Optimizations

1. A Simple but effective technique for locally improving the target code is peephole optimization,
2. a method for trying to improve the performance of the target program
3. by examining a short sequence of target instructions and replacing these instructions by a shorter or faster sequence whenever possible.

Characteristics of peephole optimization

1. Redundant instruction elimination
2. Flow of control information
3. Algebraic Simplification
4. Use of machine Idioms

Peephole Optimizations

- **Constant Folding**

  `x := 32` becomes `x := 64`

  `x := x + 32`

- **Unreachable Code**

  `goto L2`

  `x := x + 1` ← No need

- **Flow of control optimizations**

  `goto L1` becomes `goto L2`

  ...

  `L1: goto L2` ← No needed if no other L1 branch

Peephole Optimizations

- **Algebraic Simplification**

  `x := x + 0` ← No needed

- **Dead code**

  `x := 32` ← where x not used after statement

  `y := x + y` → `y := y + 32`

- **Reduction in strength**

  `x := x * 2` → `x := x + x`

  → x := x << 1

Basic Block Level

1. Common subexpression elimination
2. Constant Propagation
3. Copy Propagation
4. Dead code elimination
5. …

Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges
- A flow graph can be defined at the intermediate code level or target code level

```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

```
    MOV 0,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

Basic Blocks

- A *basic block* is a sequence of consecutive instructions with exactly one entry point and one exit point (with natural flow or a branch instruction)

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```

⇒

```
     MOV 1,R0
     MOV n,R1
     JMP L2
```

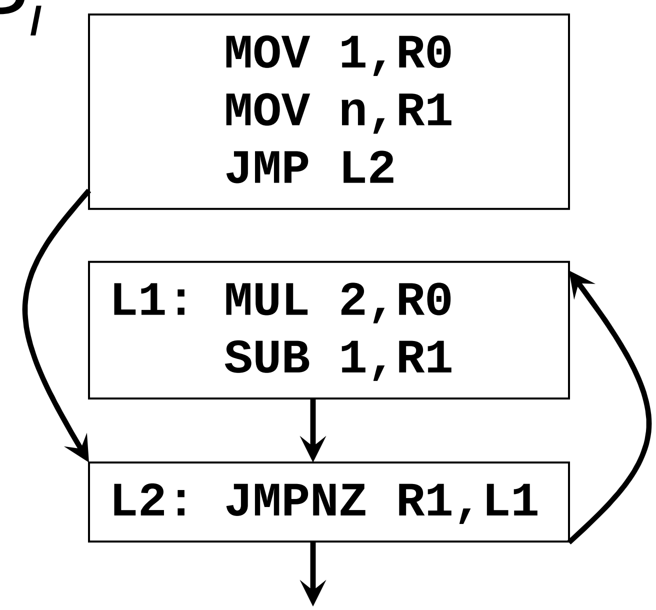
```
L1:  MUL 2,R0
     SUB 1,R1
```

```
L2:  JMPNZ R1,L1
```

Basic Blocks and Control Flow Graphs

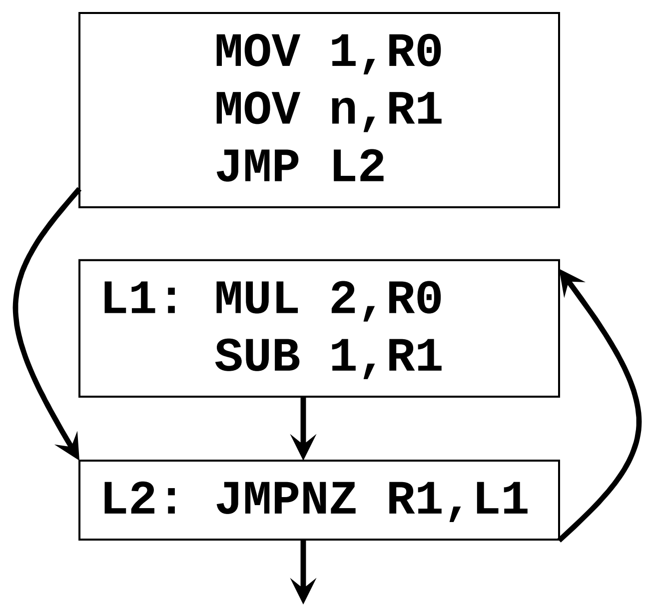
- A *control flow graph* (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i

```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```



Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block B_1 is a *predecessor* of B_2
  - Basic block B_2 is a *successor* of B_1



Partition Algorithm for Basic Blocks

Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a goto is a leader
   c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Common expression can be eliminated

Simple example: a[i+1] = b[i+1]

- t1 = i+1
- t2 = b[t1]
- t3 = i + 1
- a[t3] = t2

- t1 = i + 1
- t2 = b[t1]
- t3 = i + 1 ←
  no longer live
- a[t1] = t2

Now, suppose i is a constant:

```
i = 4
t1 = i+1
t2 = b[t1]
a[t1] = t2
```

```
i = 4
t1 = 5
t2 = b[t1]
a[t1] = t2
```

```
i = 4
t1 = 5
t2 = b[5]
a[5] = t2
```

Final Code:

```
i = 4
t2 = b[5]
a[5] = t2
```

Optimizations on CFG

- Must take control flow into account
  - Common Sub-expression Elimination
  - Constant Propagation
  - Dead Code Elimination
  - Partial redundancy Elimination
  - …
- Applying one optimization may raise opportunities for other optimizations.

Simple Loop Optimizations

- **Code Motion**

  Move invariants out of the loop.

  Example:

```
 while (i <=  limit - 2)
```

  becomes

```
t := limit - 2
while (i <= t)
```

Three Address Code of Quick Sort

1	i = m - 1
2	j = n
3	t_1 =4 * n
4	v = a[t_1]
5	i = i + 1
6	t_2 = 4 * i
7	t_3 = a[t_2]
8	if t_3 < v goto (5)
9	j = j – 1
10	t_4 = 4 * j
11	t_5 = a[t_4]
12	if t_5 > v goto (9)
13	if i >= j goto (23)
14	t_6 = 4 * i
15	x = a[t_6]

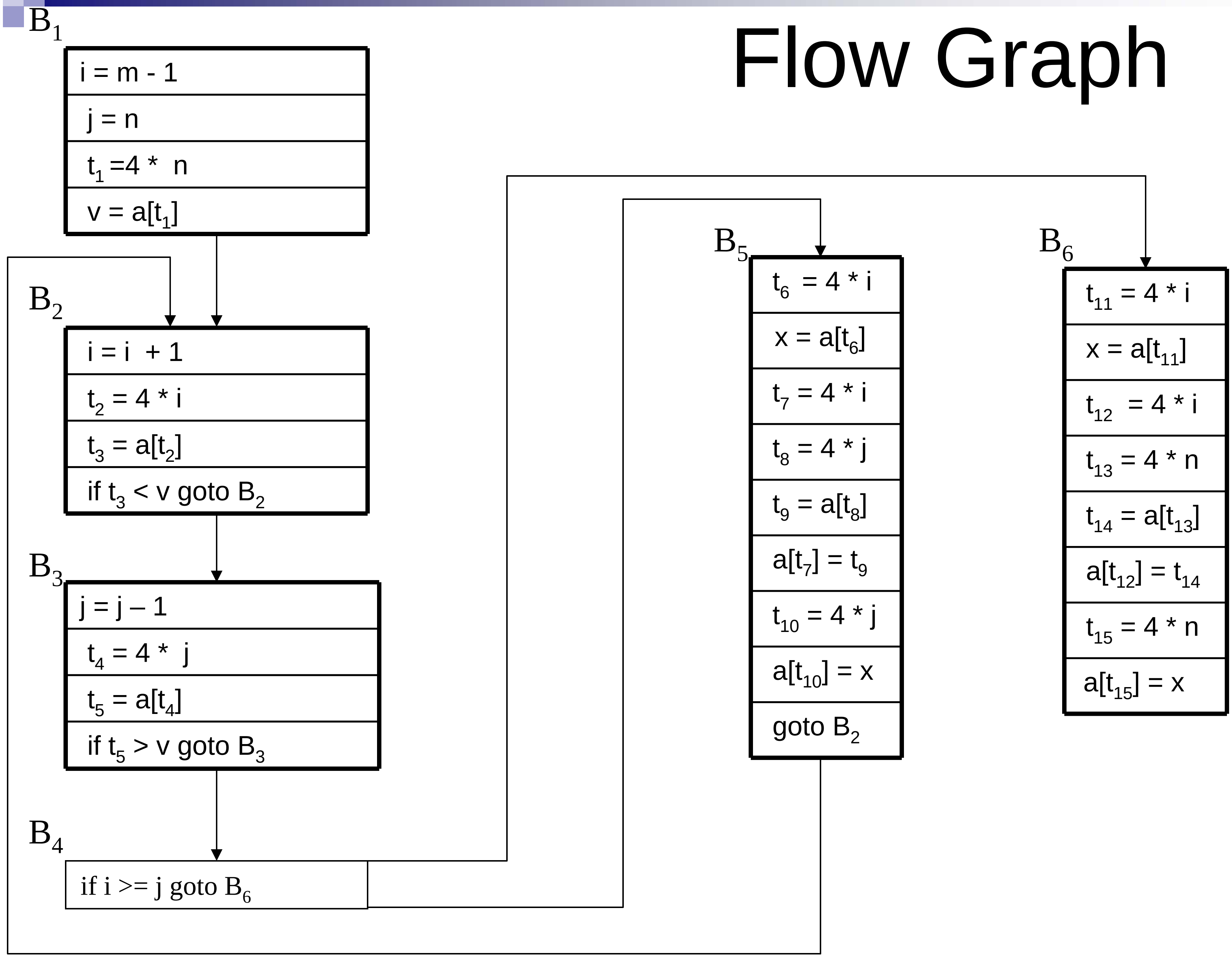
16	t_7 = 4 * I
17	t_8 = 4 * j
18	t_9 = a[t_8]
19	a[t_7] = t_9
20	t_{10} = 4 * j
21	a[t_{10}] = x
22	goto (5)
23	t_{11} = 4 * I
24	x = a[t_{11}]
25	t_{12} = 4 * i
26	t_{13} = 4 * n
27	t_{14} = a[t_{13}]
28	a[t_{12}] = t_{14}
29	t_{15} = 4 * n
30	a[t_{15}] = x

1	i = m - 1
2	j = n
3	t_1 =4 * n
4	v = a[t_1]
5	i = i + 1
6	t_2 = 4 * i
7	t_3 = a[t_2]
8	if t_3 < v goto (5)
9	j = j – 1
10	t_4 = 4 * j
11	t_5 = a[t_4]
12	if t_5 > v goto (9)
13	if i >= j goto (23)
14	t_6 = 4 * i
15	x = a[t_6]

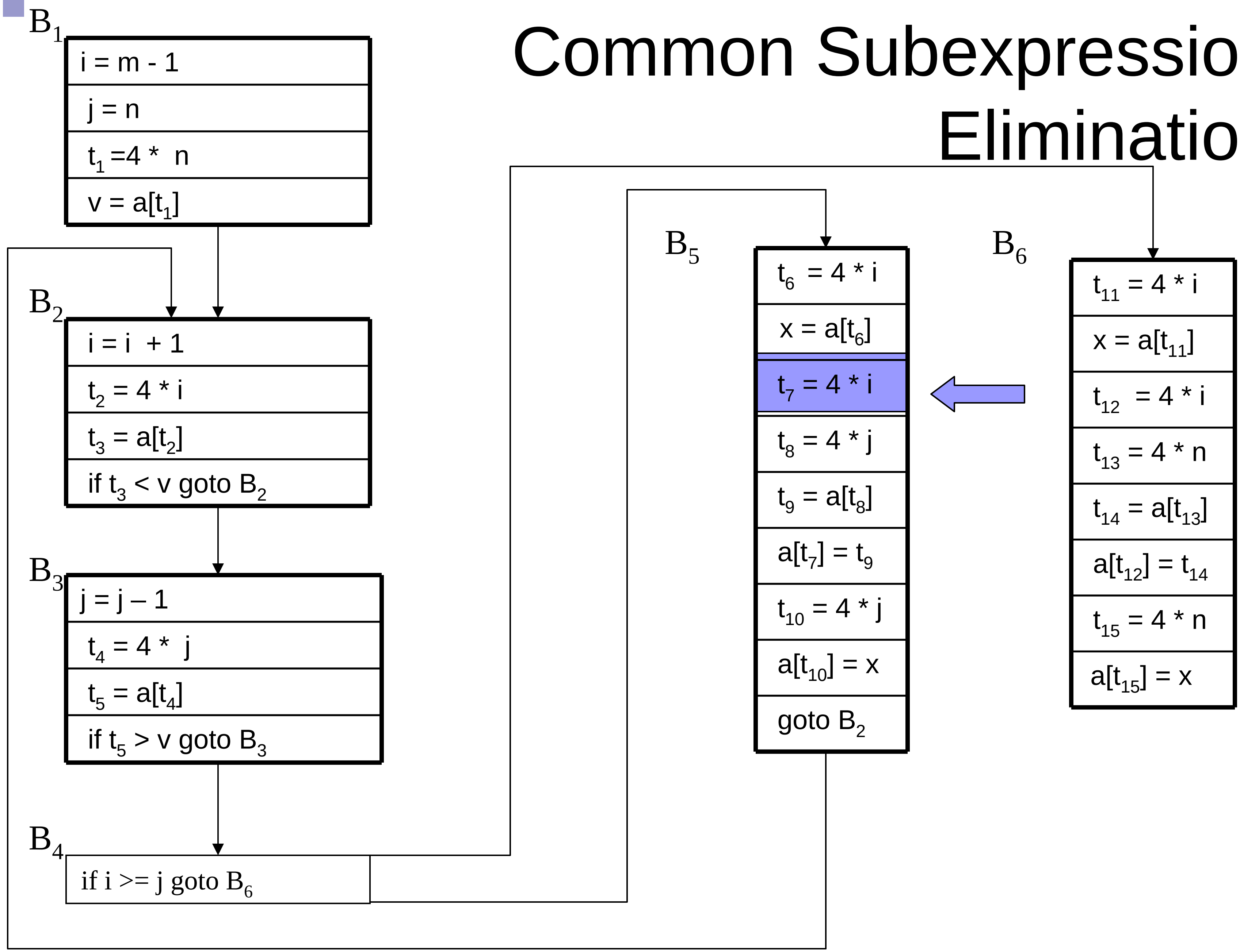
Find The Basic Block

16	t_7 = 4 * I
17	t_8 = 4 * j
18	t_9 = a[t_8]
19	a[t_7] = t_9
20	t_{10} = 4 * j
21	a[t_{10}] = x
22	goto (5)
23	t_{11} = 4 * i
24	x = a[t_{11}]
25	t_{12} = 4 * i
26	t_{13} = 4 * n
27	t_{14} = a[t_{13}]
28	a[t_{12}] = t_{14}
29	t_{15} = 4 * n
30	a[t_{15}] = x

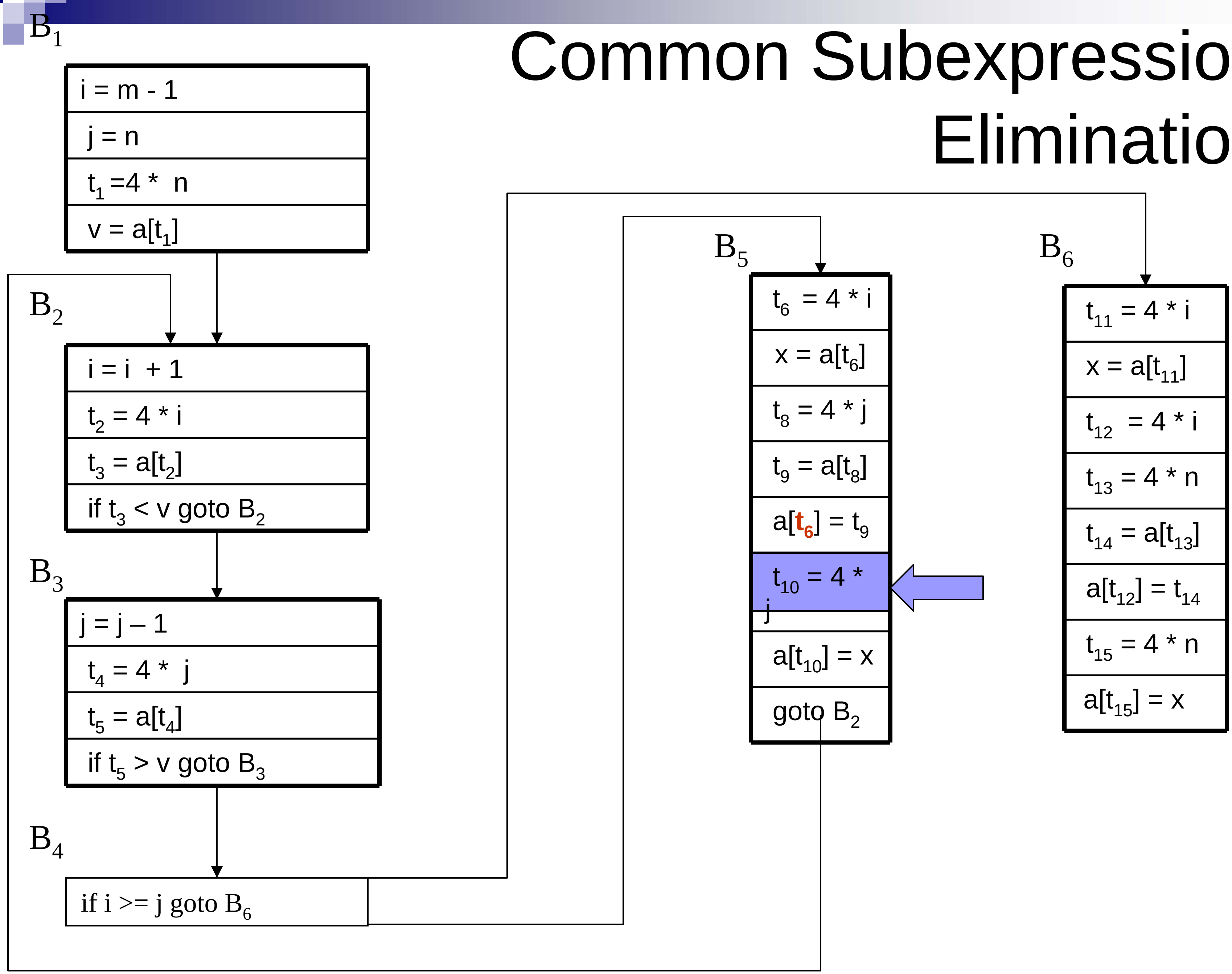
Flow Graph


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]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
B6
t11 = 4 * i
x = a[t11]
t12 = 4 * i
t13 = 4 * n
t14 = a[t13]
a[t12] = t14
t15 = 4 * n
a[t15] = x

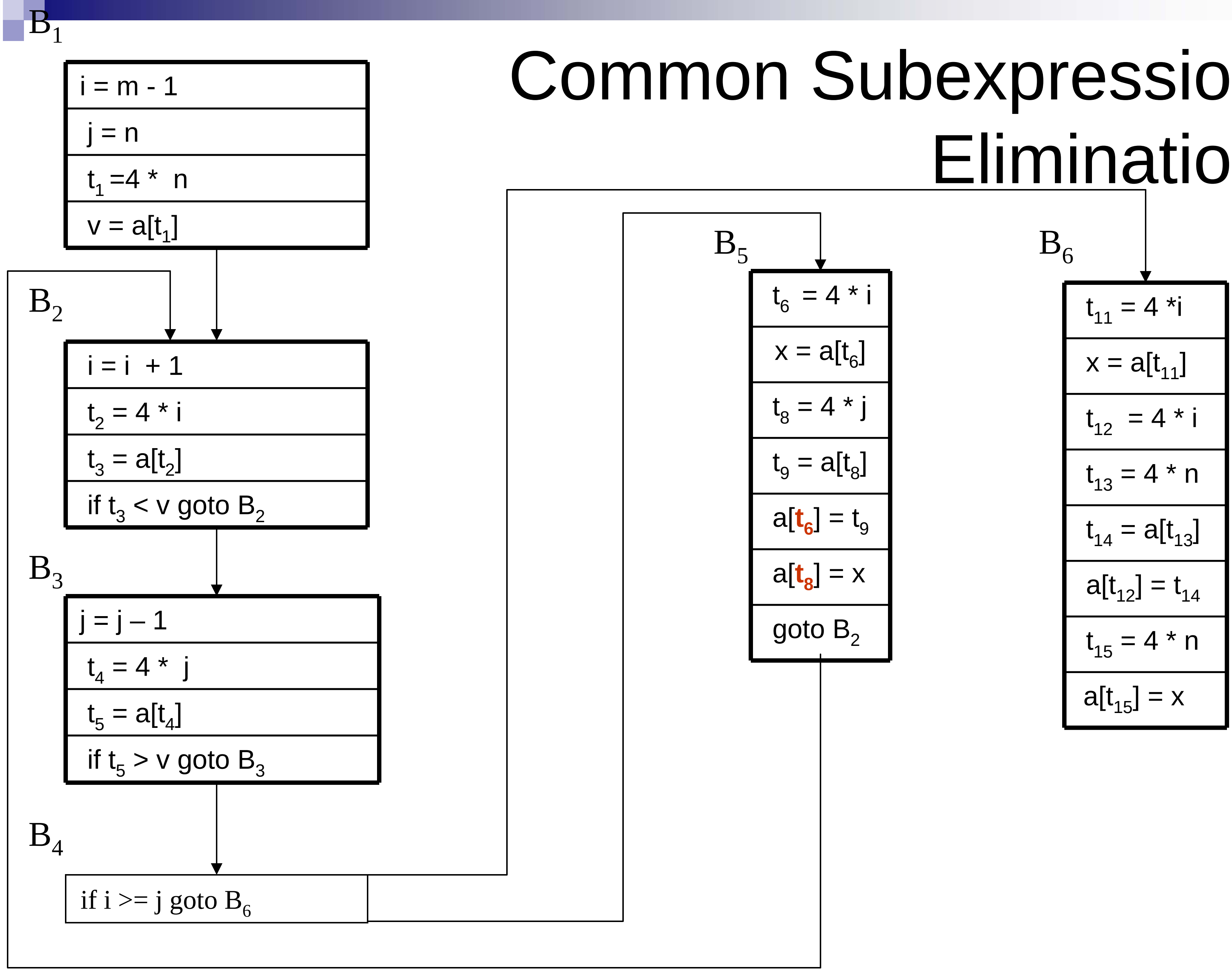
Common Subexpression Elimination



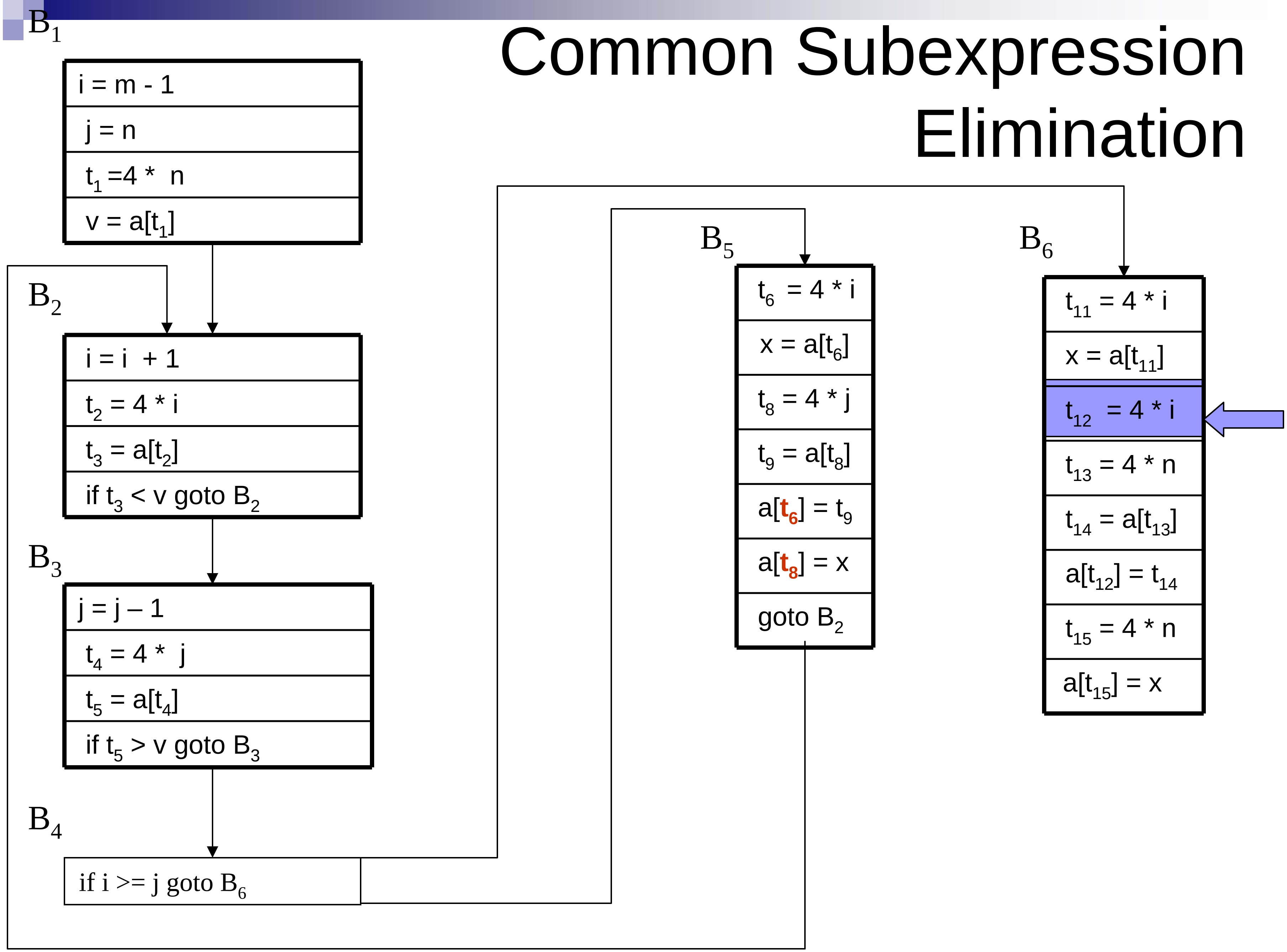
Common Subexpression Elimination



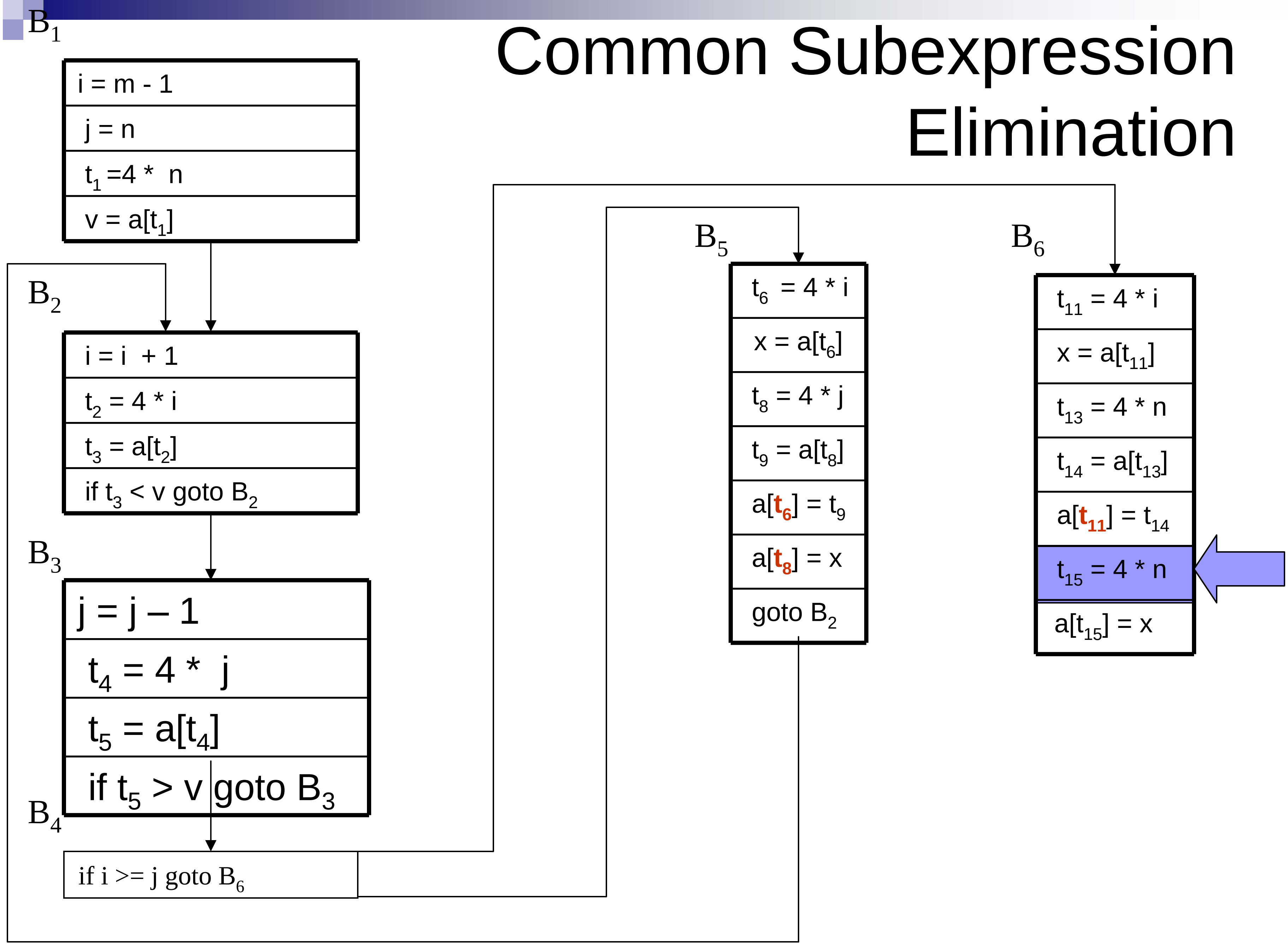
Common Subexpression Elimination



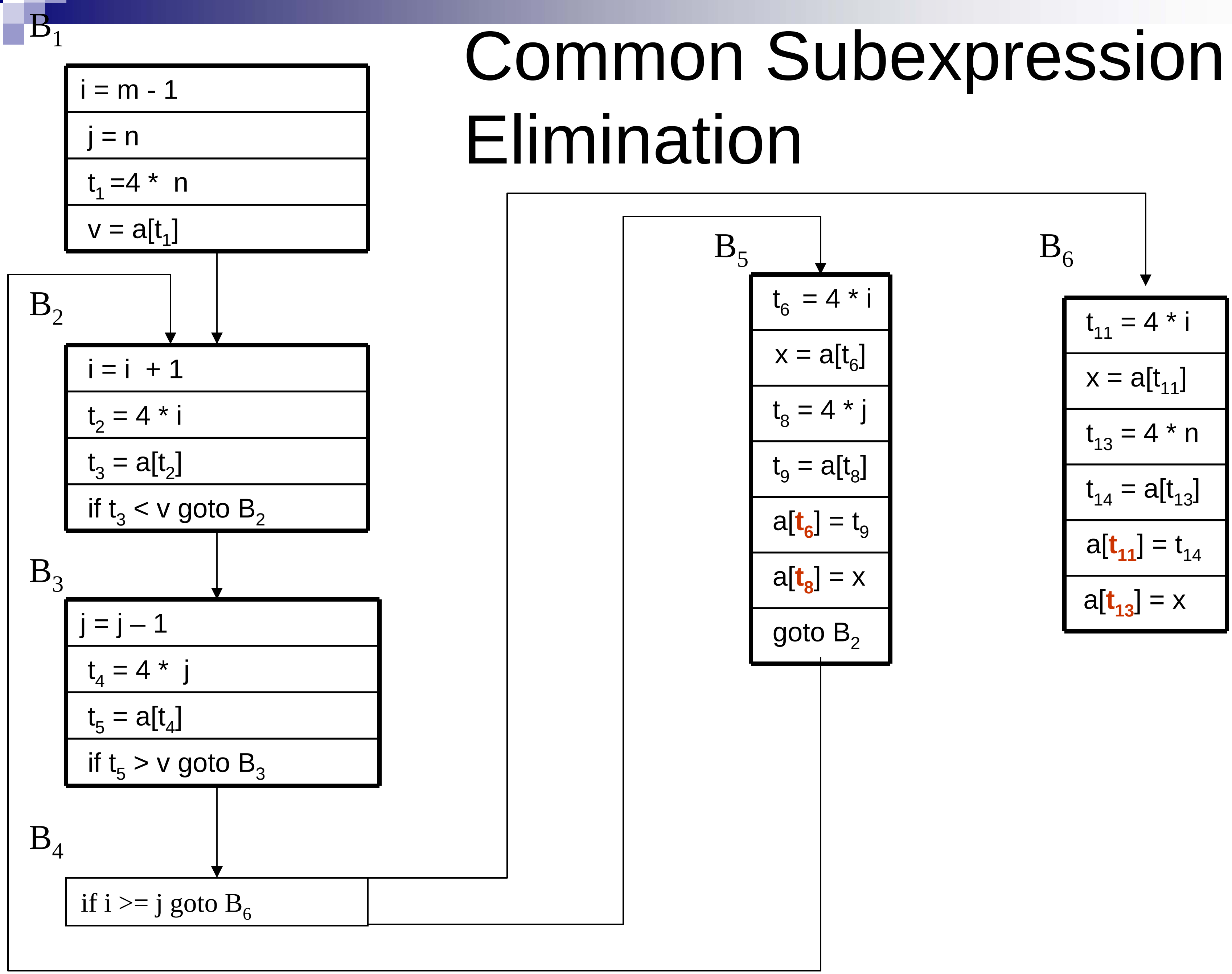
Common Subexpression Elimination



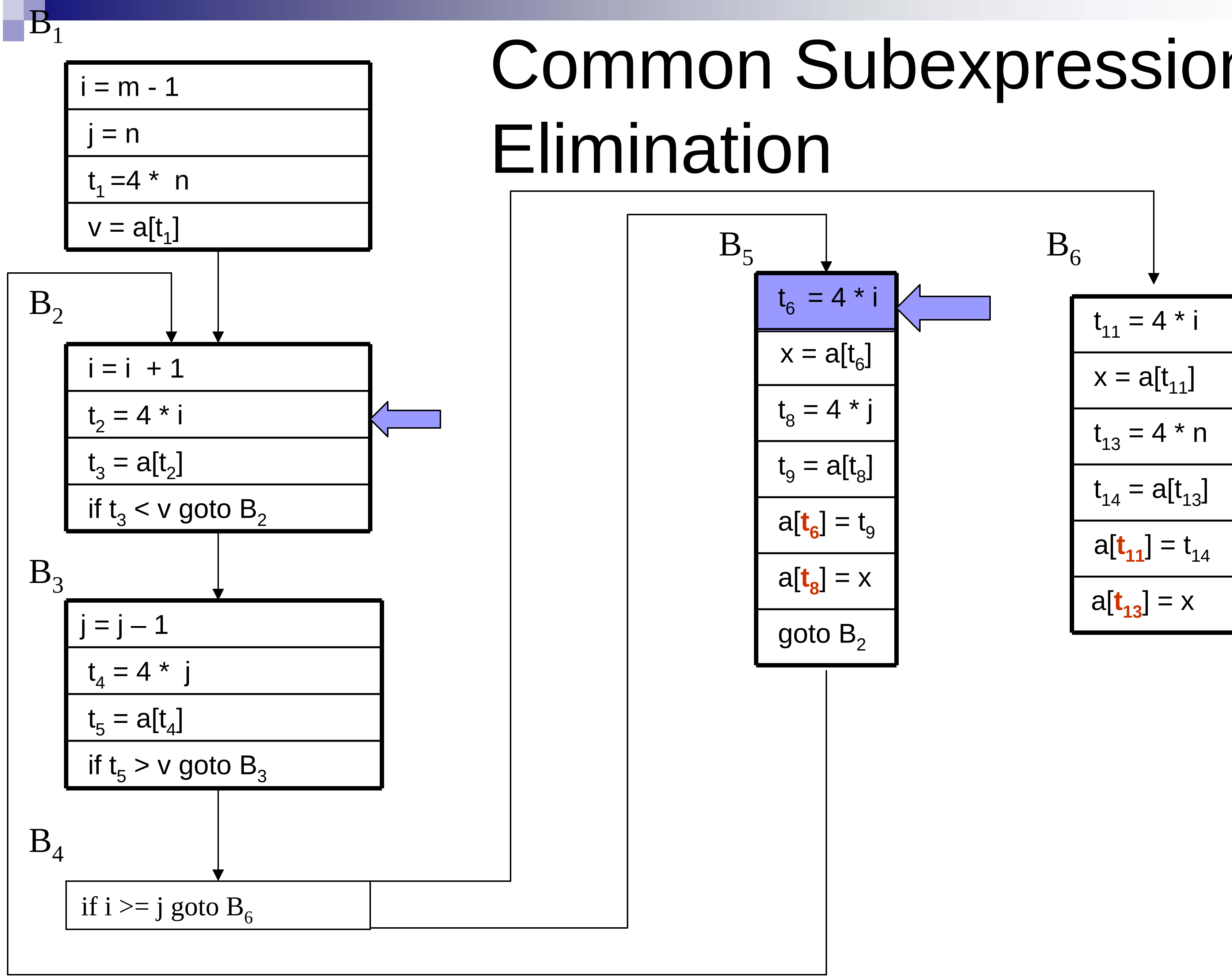
Common Subexpression Elimination



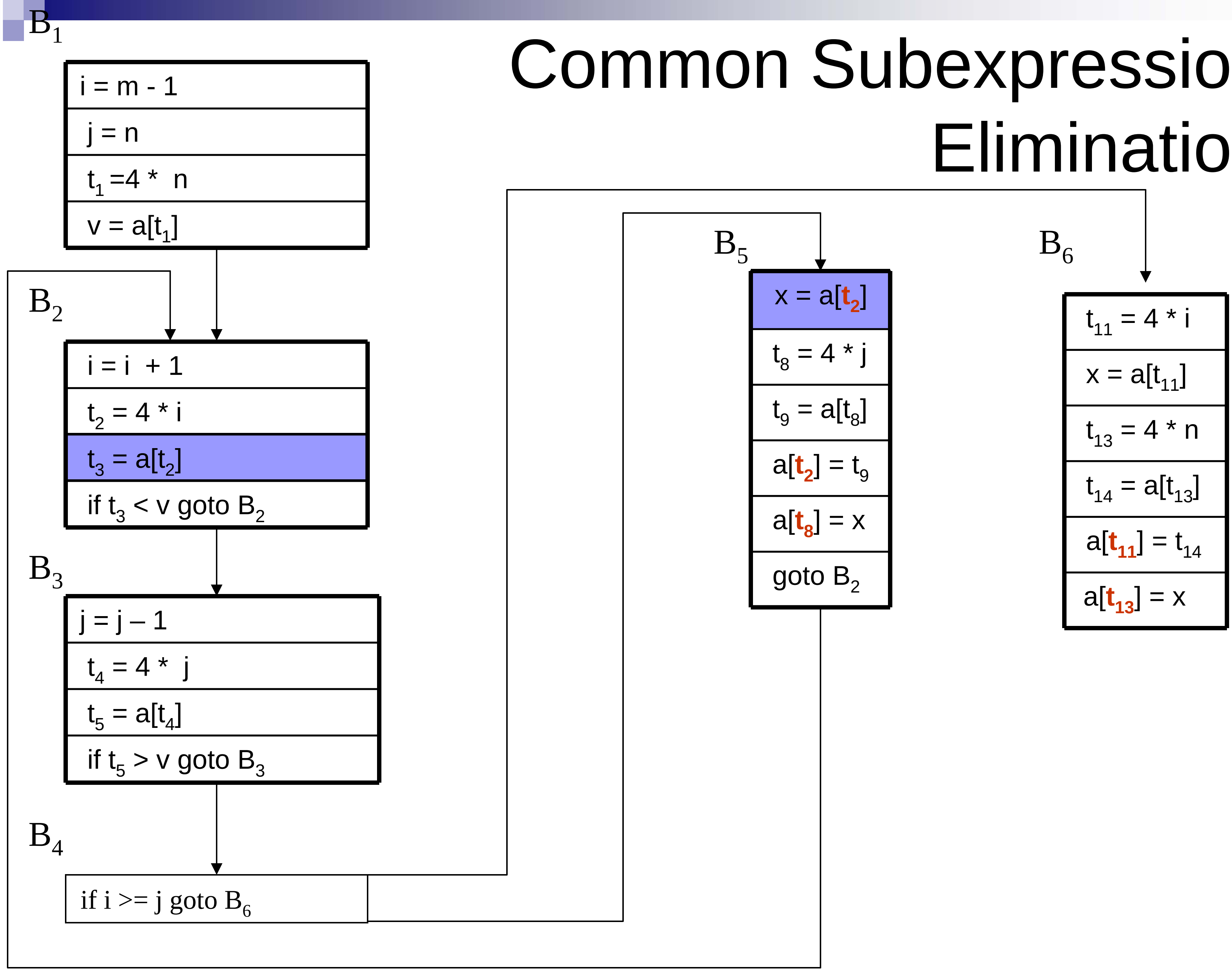
Common Subexpression Elimination



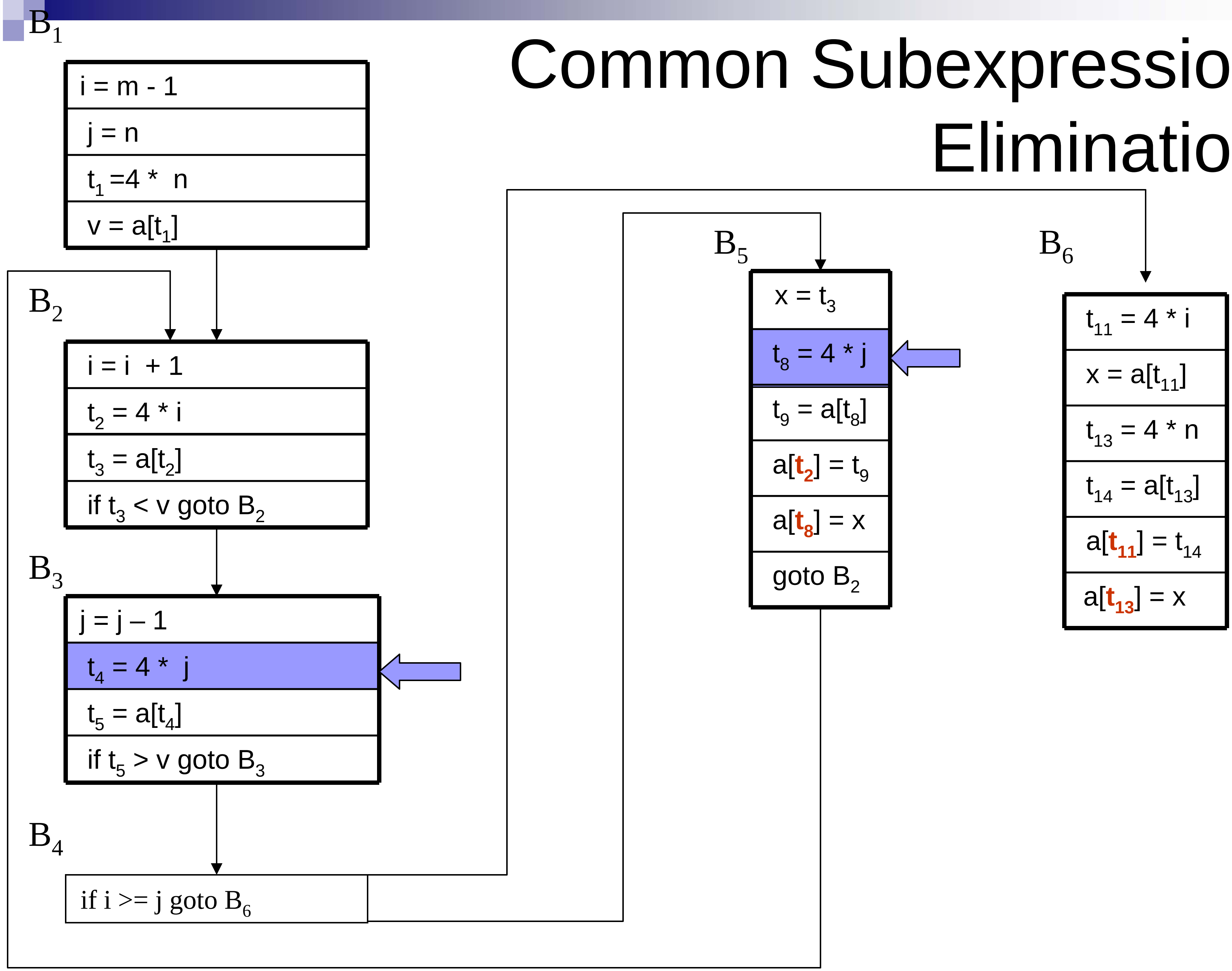
Common Subexpression Elimination



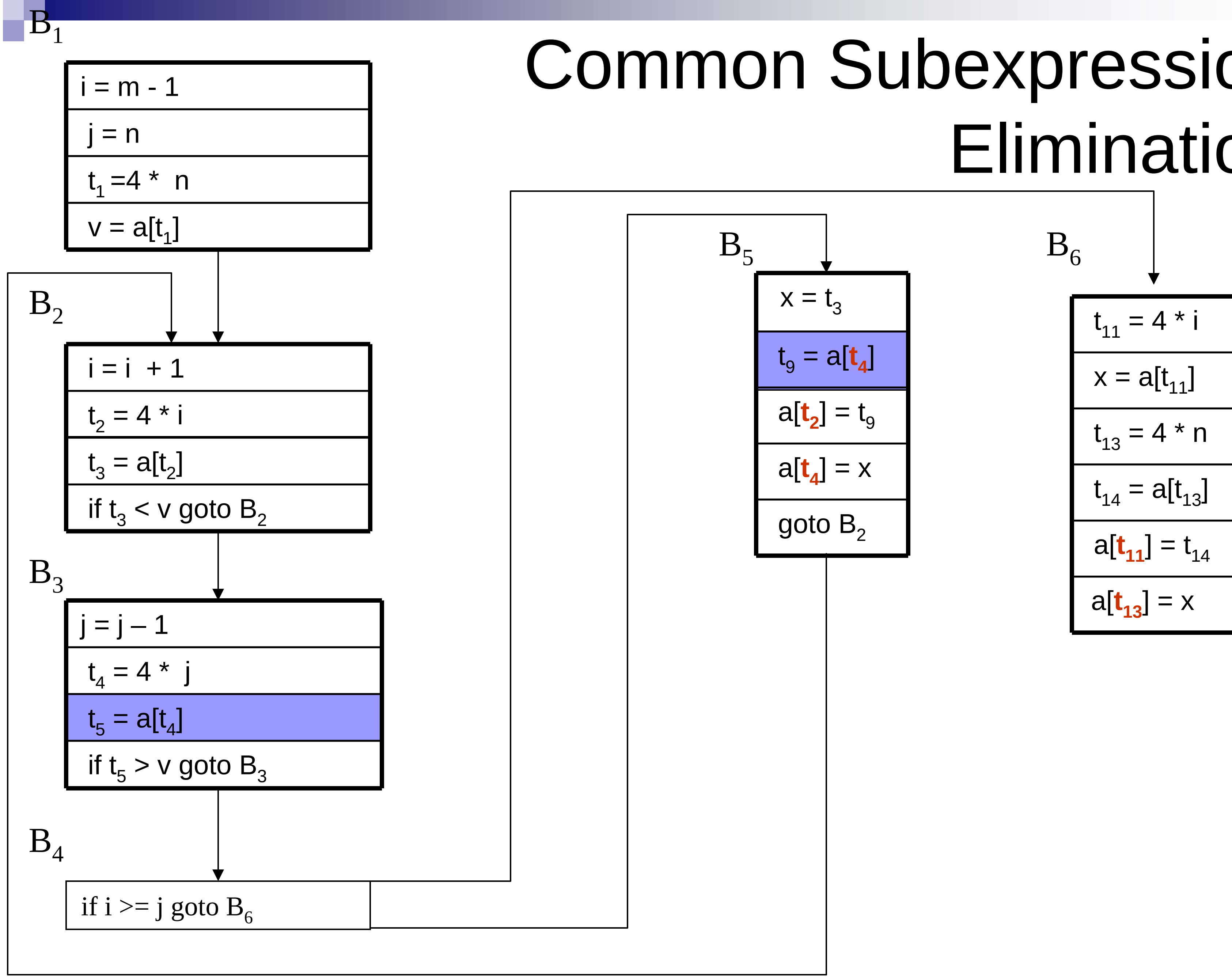
Common Subexpression Elimination



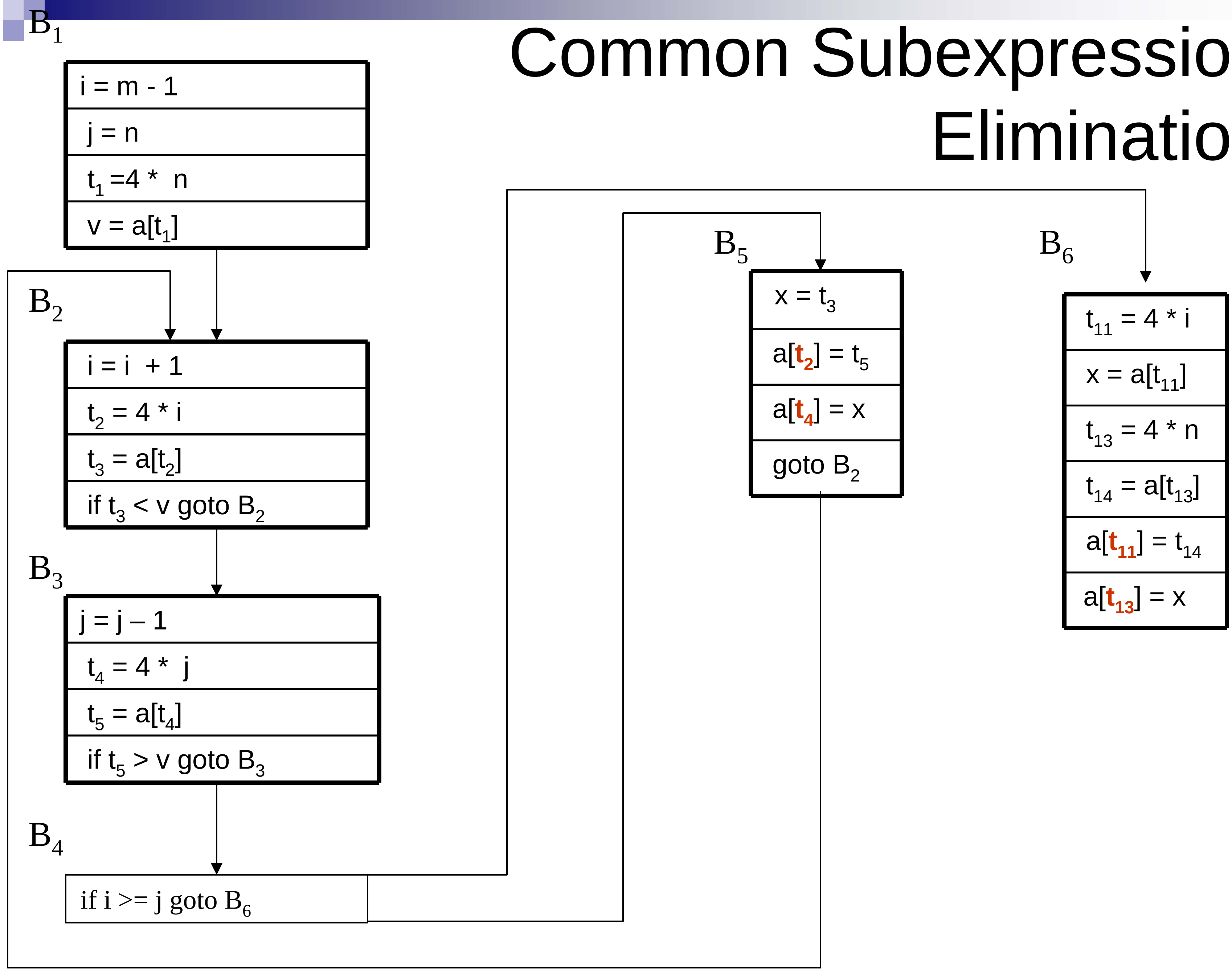
Common Subexpression Elimination



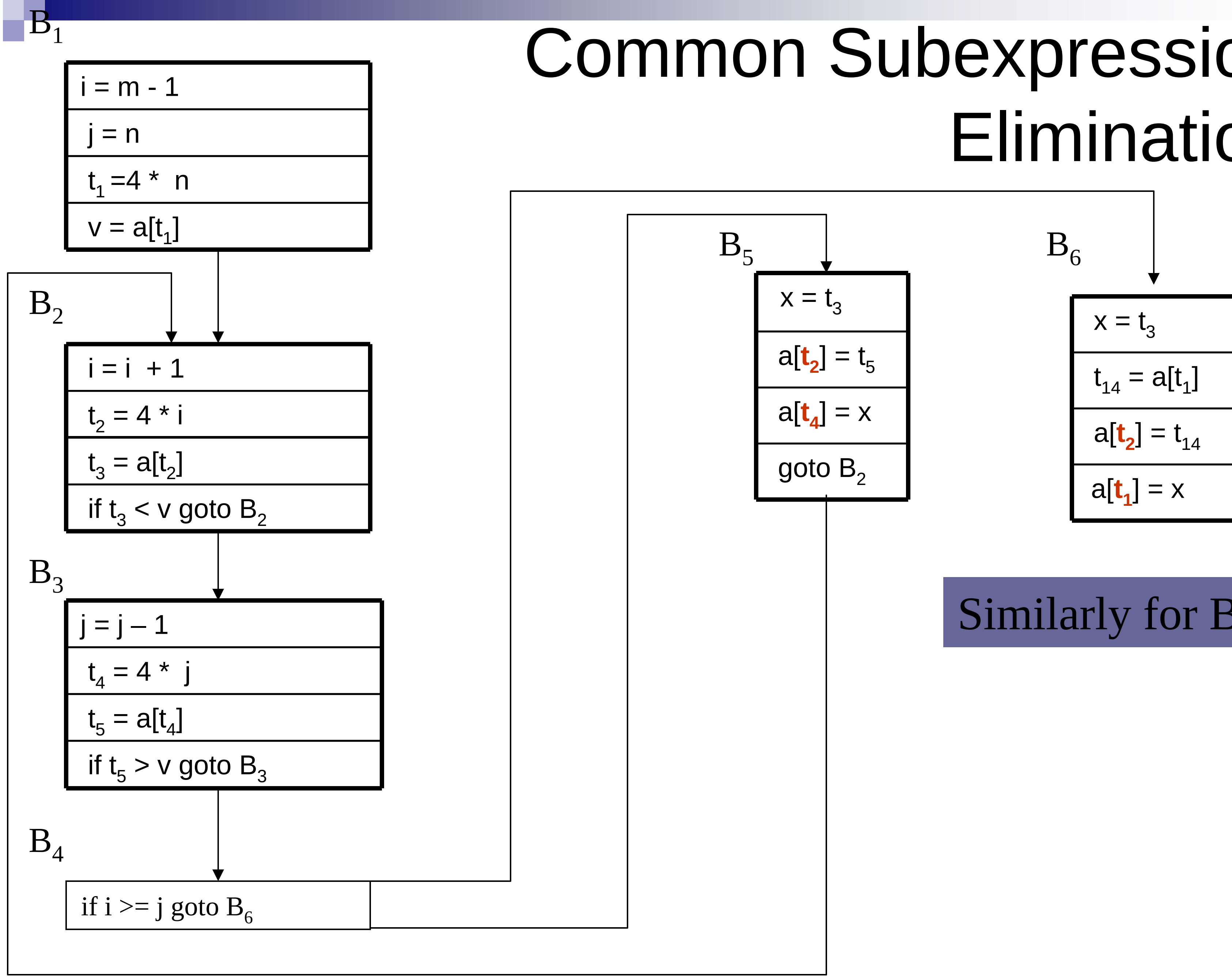
Common Subexpression Elimination



Common Subexpression Elimination

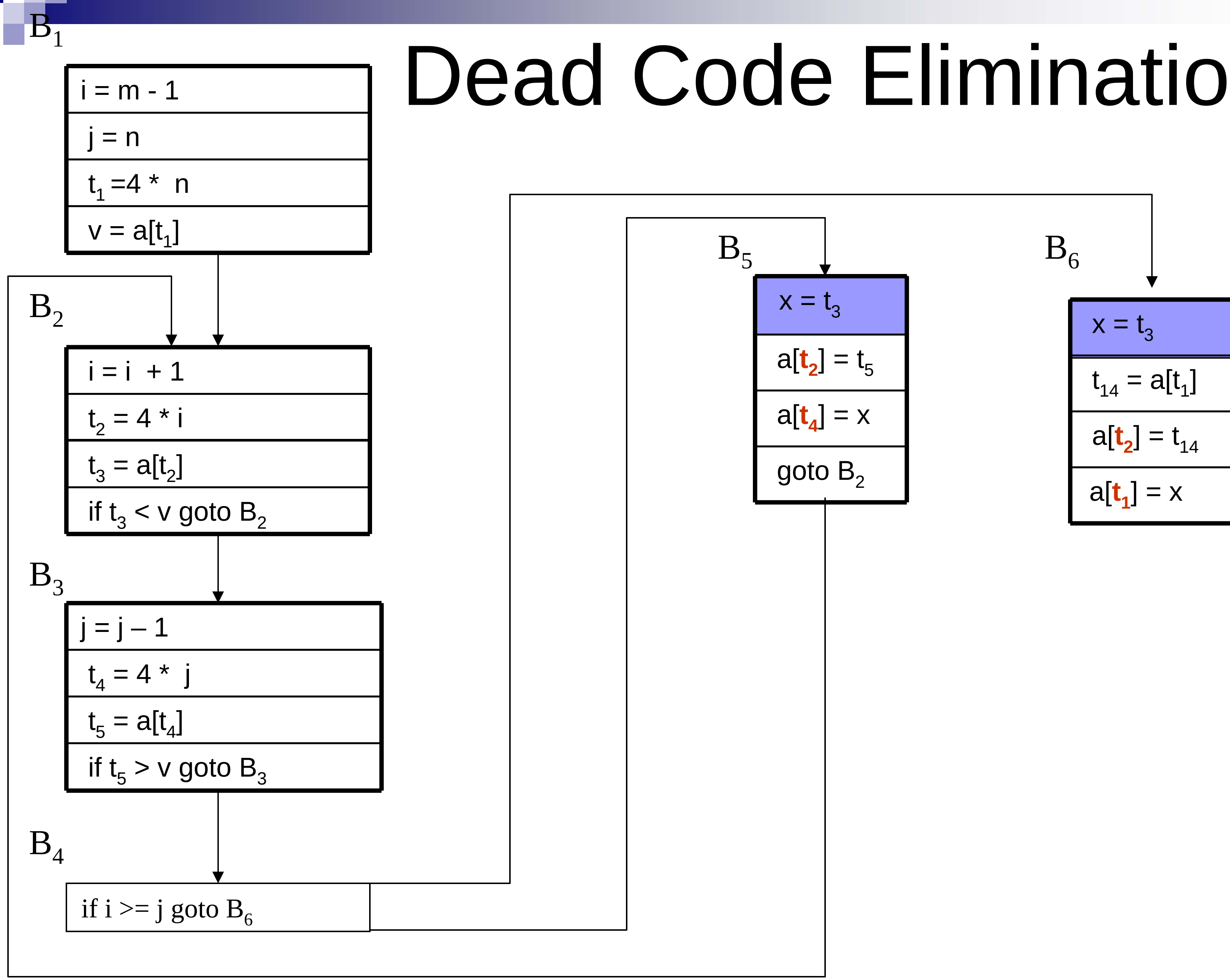


Common Subexpression Elimination

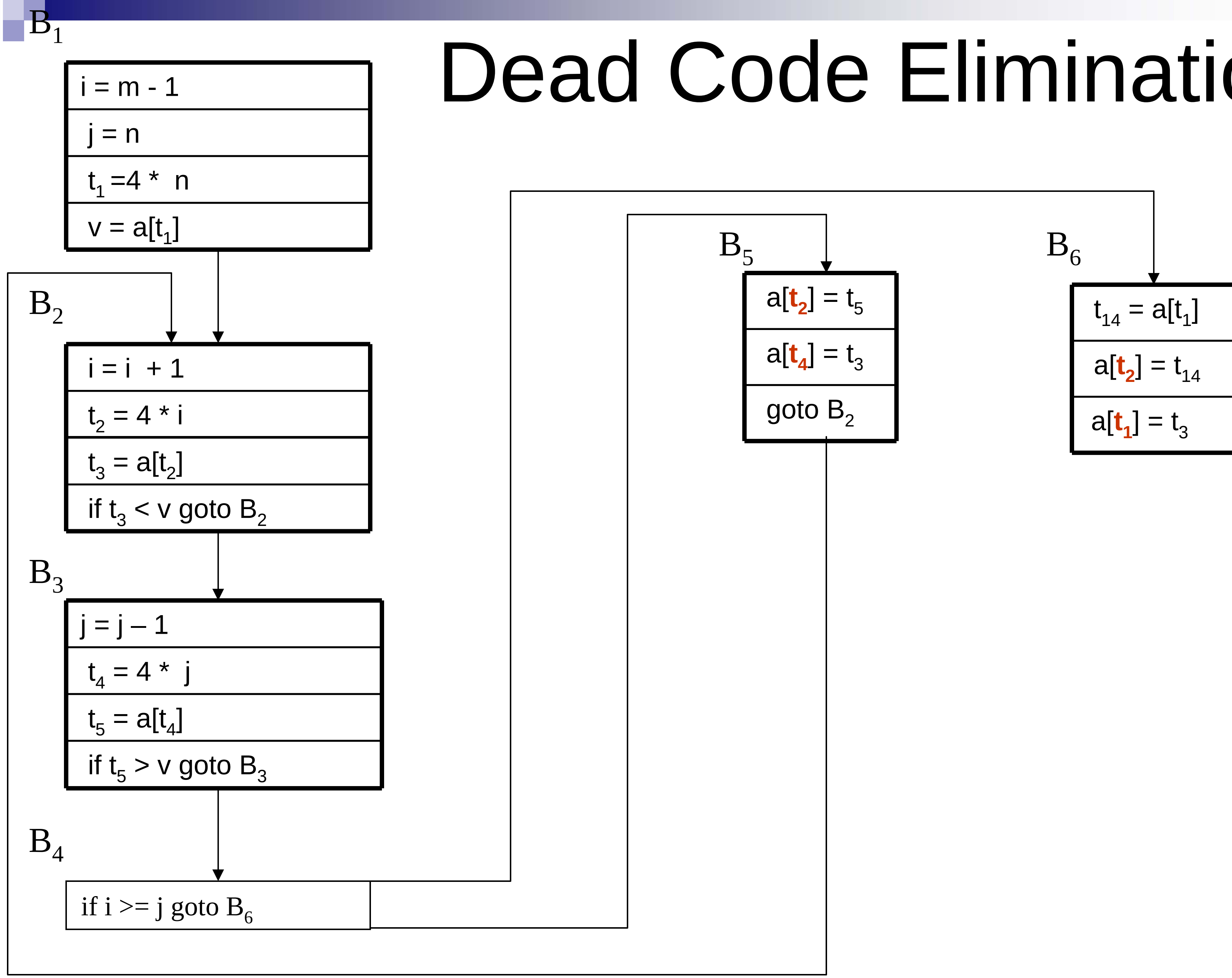


Similarly for B_6

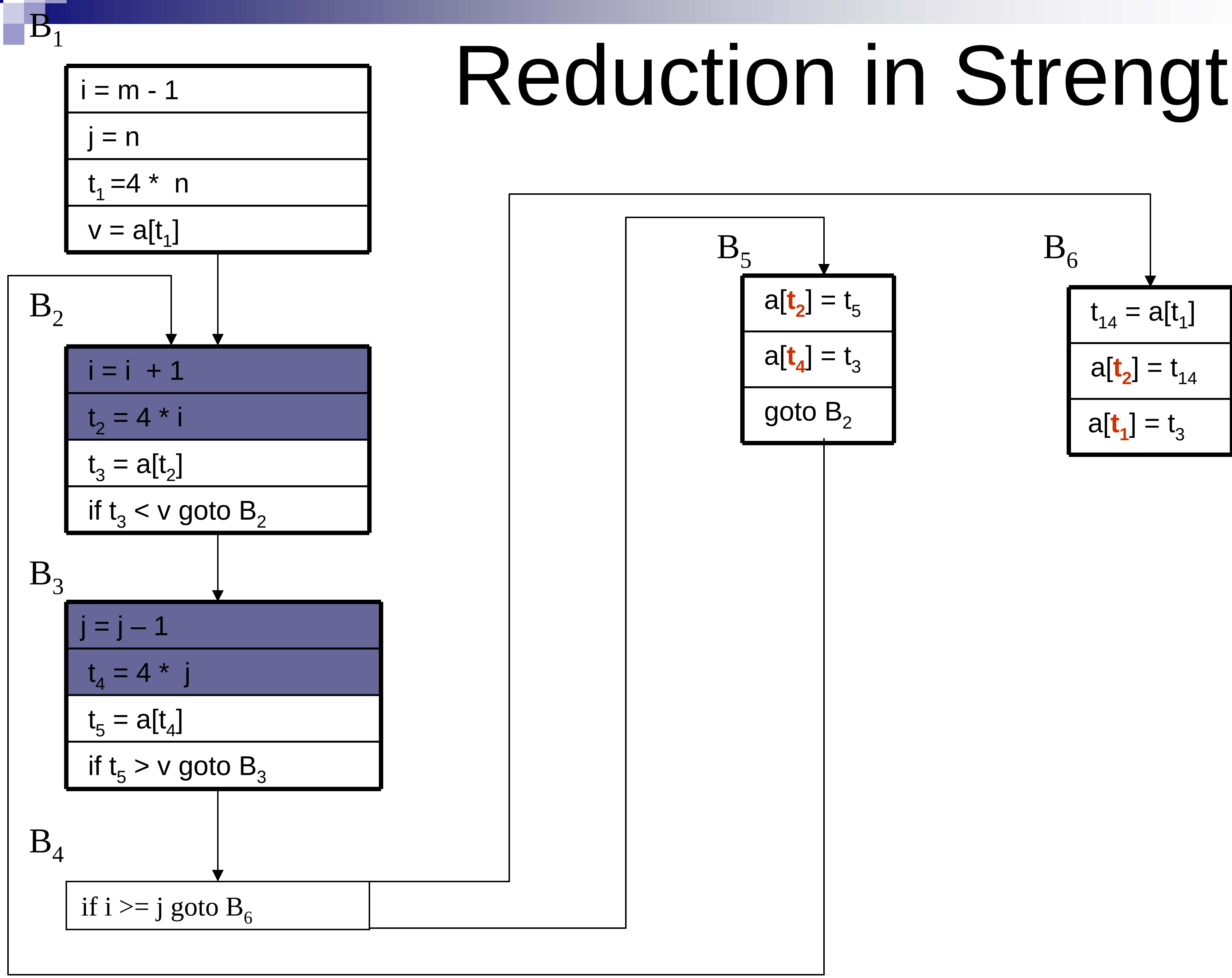
Dead Code Elimination



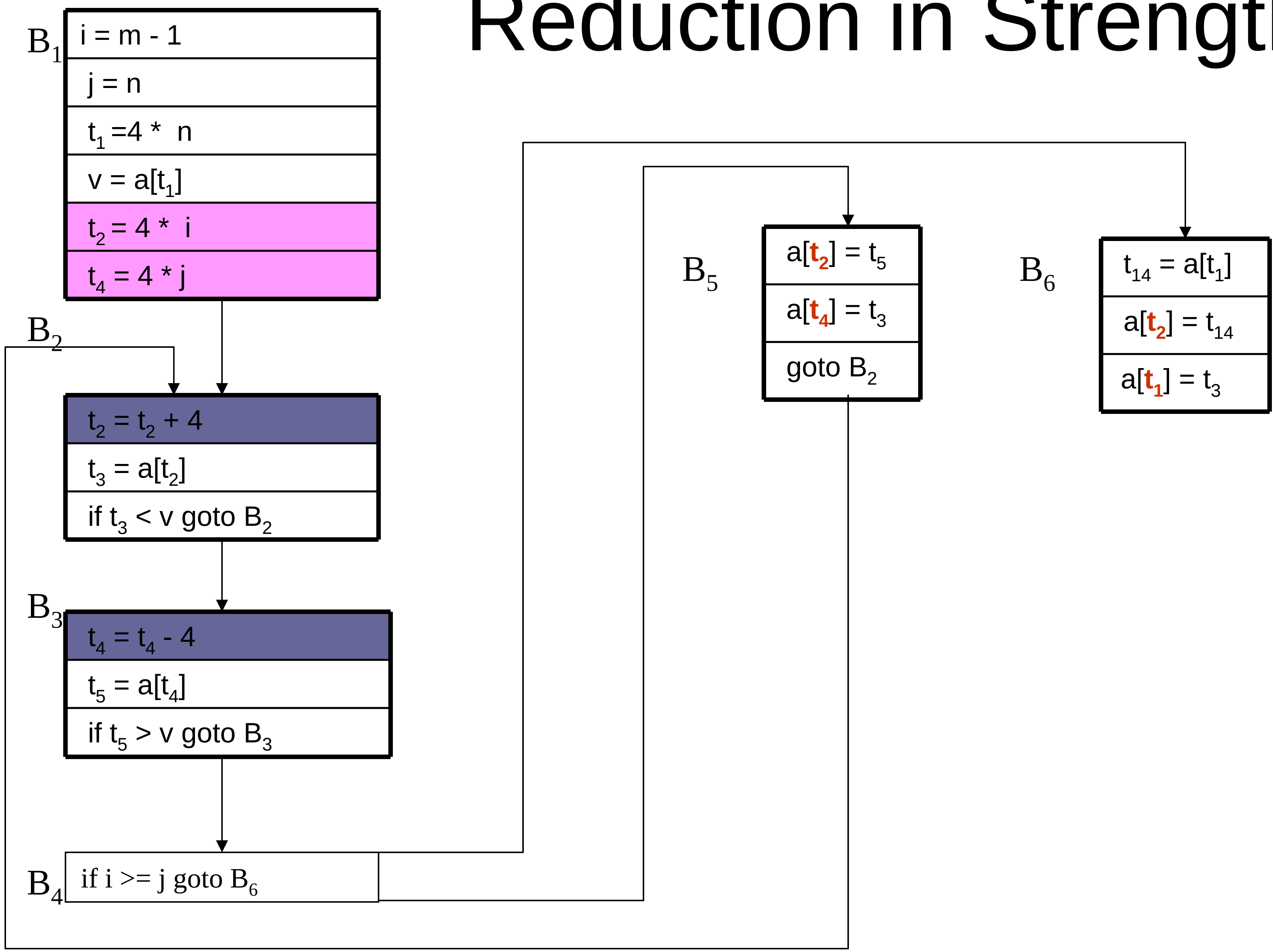
Dead Code Elimination



Reduction in Strength



Reduction in Strength



A Code Generator

- Generates target code for a sequence of three-address statements using next-use information
- Uses new function *getreg* to assign registers to variables
- Computed results are kept in registers as long as possible, which means:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Checks if operands to three-address code are available in registers

The Code Generation Algorithm

- For each statement x := y op z
  1. Set location $L = getreg(y, z)$
  2. If $y \notin L$ then generate
     `MOV` y',L
     where y' denotes one of the locations where the value of y is available (choose register if possible)
  3. Generate
     `OP` z',L
     where z' is one of the locations of z;
     Update register/address descriptor of x to include L
  4. If y and/or z has no next use and is stored in register, update register descriptors to remove y and/or z

Register and Address Descriptors

- A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.
  `MOV a,R0` “**`R0`** contains **`a`**”
- An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.
  `MOV a,R0`
  `MOV R0,R1` “**`a`** in **`R0`** and **`R1`**”

The *getreg* Algorithm

- To compute *getreg*(*y*,*z*)
  1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
     Update address descriptor: value *y* no longer in *R*
  2. Else, return a new empty register if available
  3. Else, find an occupied register *R*;
     Store contents (register spill) by generating
     `MOV` *R*,*M*
     for every *M* in address descriptor of *y*;
     Return register *R*
  4. Return a memory location

Code Generation Example

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*
`t := a - b`	**`MOV a,R0`** **`SUB b,R0`**	Registers empty **`R0`** contains **`t`**	**`t`** in **`R0`**
`u := a - c`	**`MOV a,R1`** **`SUB c,R1`**	**`R0`** contains **`t`** **`R1`** contains **`u`**	**`t`** in **`R0`** **`u`** in **`R1`**
`v := t + u`	**`ADD R1,R0`**	**`R0`** contains **`v`** **`R1`** contains **`u`**	**`u`** in **`R1`** **`v`** in **`R0`**
`d := v + u`	**`ADD R1,R0`** **`MOV R0,d`**	**`R0`** contains **`d`**	**`d`** in **`R0`** **`d`** in **`R0`** and memory

Register Allocation and Assignment

- The *getreg* algorithm is simple but sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block
- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings

Allocating Registers in Loops

- Suppose loading a variable *x* has a cost of 2
- Suppose storing a variable *x* has a cost of 2
- Benefit of allocating a register to a variable *x* within a loop *L* is

  $$\sum_{B \in L} (\ use(x, B) + 2\ live(x, B)\ )$$

  where *use*(*x*, *B*) is the number of times *x* is used in *B* and *live*(*x*, *B*) = true if *x* is live on exit from *B*

Global Register Allocation Using Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (spilled) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*)
- Find a *k*-coloring for the graph, with *k* the number of registers