Program Optimization

Program optimization

- A computer program may be optimized so that it executes more rapidly, or is capable of operating with **less memory storage** or **other resources, or less power**.

- The process of modifying a software system to make some aspect of it work more efficiently or use fewer resources.

Platform dependent/ Platform independent optimization

- Code optimization can be also broadly categorized as

- **Platform-dependent** techniques use specific properties of one platform, or rely on parameters depending on the single platform or even on the single processor. Platform dependent techniques involve ***instruction scheduling***, ***instruction-level parallelism, data-level parallelism, cache optimization techniques*** (i.e. parameters that differ among various platforms) and the optimal instruction scheduling might be different even on different processors of the same architecture.

- In **Platform independent** writing or producing different versions of the same code for different processors might be needed. In the case of ***compile-level optimization***, platform-independent techniques are ***generic techniques*** such as ***loop unrolling, reduction in function calls, memory efficient routines, reduction in conditions,*** etc. Compile level optimization impact most CPU architectures in a similar way. Compile level optimization serve to reduce the total Instruction path length required to complete the program and/or reduce total memory usage during the process

Levels of Optimization

- **Design level:**
  - Architectural design of a system affects its performance. At the highest level, the design may be optimized to make best use of the available resources.
- **Source code level:**
  - avoid poor quality, coding can also improve performance.
- **Assembly level**
  - At the lowest level, writing code using an assembly language, designed for a particular hardware platform will normally produce the most efficient code.
- **Run time:** Just in time compilers and Assembler programmers may be able to perform run time optimization exceeding the capability of static compilers by dynamically adjusting parameters according to the actual input or other factors.

Levels of Optimization

- **Compile level:**
  - Compilers try to generate good code. I.e. Fast
  - Code improvement is challenging
  - Code improvement may slow down the compilation process. In some domains, such as just-in-time compilation, compilation speed is critical.

Themes behind Optimization Techniques

- **Avoid redundancy:** something already computed need not be computed again
- **Smaller code:** less work for CPU, cache, and memory!
- **Less jumps:** jumps interfere with code pre-fetch
- **Code locality:** codes executed close together in time is generated close together in memory – increase locality of reference
- **Extract more information about code**: More info – better code generation

Criterion of code optimization

- Must preserve the semantic equivalence of the programs.
- The algorithm should not be modified.
- Transformation, on average should speed up the execution of the program.
- Worth the effort: Intellectual and compilation effort spend on insignificant improvement..
- Transformations are simple enough to have a good effect

Peephole Optimization

- Simple compiler do not perform machine-independent code improvement
  - They generates *naïve* code

- It is possible to take the target hole and optimize it
  - Sub-optimal sequences of instructions that match an optimization pattern are transformed into optimal sequences of instructions
  - This technique is known as ***peephole optimization***
  - Peephole optimization usually works by sliding a window of several instructions (a *peephole*)

Optimizing Transformations
(Basic peephole techniques)

- Compile time evaluation
- Common sub-expression elimination
- Code motion
- Strength Reduction
- Dead code elimination
- Copy propagation
- Loop optimization
  - Induction variables and strength reduction

Compile-Time Evaluation

- Expressions whose values can be pre-computed at the compilation time
- Two ways:
  - Constant folding
  - Constant propagation

Compile-Time Evaluation

- **Constant folding**: Evaluation of an expression with constant operands to replace the expression with single value
- Example:

```
area := (22.0/7.0) * r ** 2
```

Compile-Time Evaluation

- **Constant Propagation**: Replace a variable with constant which has been assigned to it earlier.
- Example:

```
pi := 3.14286
area = pi * r ** 2
```

```
                    area = 3.14286 * r
** 2
```

Constant Propagation

- What does it mean?
  - Given an assignment x = c, where c is a constant, replace later uses of x with uses of c, provided there are no intervening assignments to x.
    - Similar to copy propagation
    - Extra feature: It can analyze constant-value conditionals to determine whether a branch should be executed or not.
- When is it performed?
  - Early in the optimization process.
- What is the result?
  - Smaller code
  - Fewer registers

Common Sub-expression Evaluation

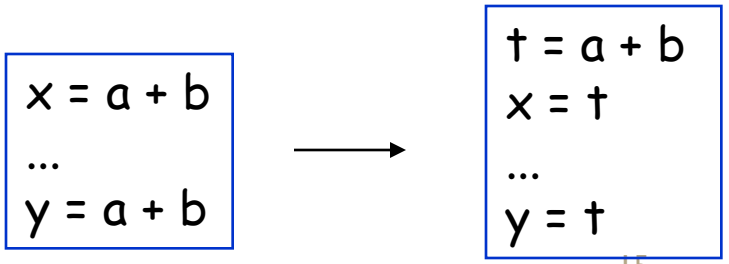
- Identify common sub-expression present in different expression, compute once, and use the result in all the places.
  - The *definition* of the variables involved should not change

Example:

```
a := b * c                    temp := b * c
…                             a := temp
…                             …
x := b * c + 5                x := temp + 5
```

Common Subexpression Elimination

- Local common sub expression elimination
  - Performed within basic blocks
  - Algorithm sketch:
    - Traverse BB from top to bottom
    - Maintain table of expressions evaluated so far
      - if any operand of the expression is redefined, remove it from the table
    - Modify applicable instructions as you go
      - generate temporary variable, store the expression in it and use the variable next time the expression is encountered.



Common Subexpression Elimination

```
c = a + b
d = m * n
e = b + d
f = a + b
g = - b
h = b + a
a = j + a
k = m * n
j = b + d
a = - b
if m * n go to L
```



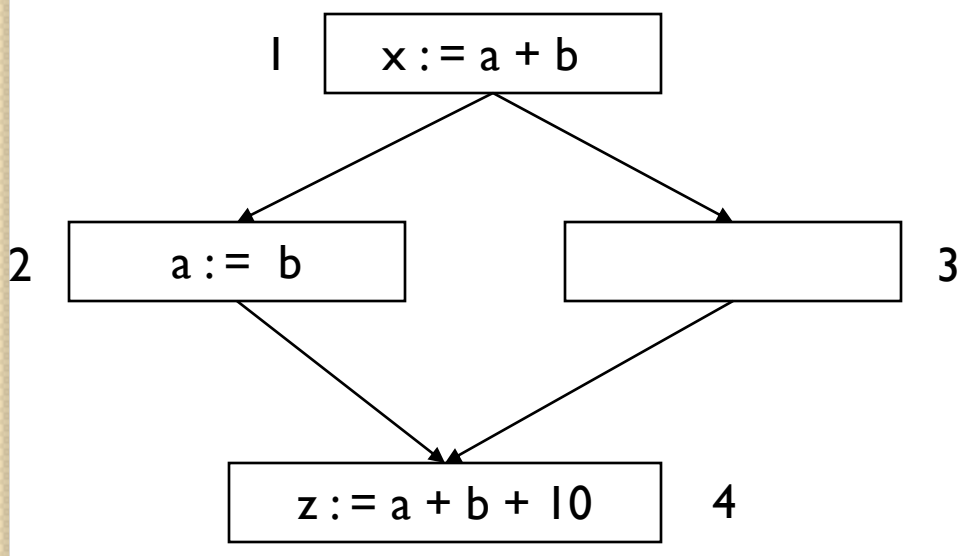
```
t1 = a + b
c = t1
t2 = m * n
d = t2
t3 = b + d
e = t3
f = t1
g = -b
h = t1 /* commutative */
a = j + a
k = t2
j = t3
a = -b
if t2 go to L
```

the table contains quintuples:
(pos, opd1, opr, opd2, tmp)

Common Subexpression Elimination

- Global common subexpression elimination
  - Performed on flow graph
  - Requires available expression information
    - In addition to finding what expressions are available at the endpoints of basic blocks, we need to know where each of those expressions was most recently evaluated (which block and which position within that block).

Common Sub-expression Evaluation



"a + b" is not a common sub-expression in 1 and 4

None of the variable involved should be modified in any path

Code Motion

- Moving code from one part of the program to other without modifying the algorithm
  - Reduce size of the program
  - Reduce execution frequency of the code subjected to movement

Code Motion

1. *Code Space reduction*: Similar to common sub-expression elimination but with the objective to reduce code size.

Example: Code hoisting

```
if (a< b) then
    z := x ** 2
else
    y := x ** 2 + 10
```

⟶

```
temp : = x ** 2
if (a< b) then
    z := temp
else
    y := temp + 10
```

"x ** 2" is computed once in both cases, but the code size in the second case reduces.

Code Motion

2 *Execution frequency reduction*: reduce execution frequency of partially available expressions (expressions available atleast in one path)

Example:

```
if (a<b) then
      z = x * 2
else
y = 10
g = x * 2
```

```
 temp = x * 2
if (a<b) then
 z = temp
else
   y = 10
g = temp;
```

Code Motion

- Move expression out of a loop if the evaluation does not change inside the loop.

Example:

```
while ( i < (max-2) ) ...
```

Equivalent to:

```
t :=  max - 2
while ( i < t ) ...
```

Code Motion

- Safety of Code movement

  Movement of an expression e from a basic block b_i to another block b_j, is safe if it does not introduce any new occurrence of e along any path.

Example: Unsafe code movement

```
if (a<b) then                    temp = x * 2
   z = x * 2                          if (a<b) then
else                      ⟶       z = temp
   y = 10                                else
                                    y = 10
```

Strength Reduction

- Replacement of an operator with a less costly one.

Example:

```
for i=1 to 10 do
…
x = i * 5
…

end
```

⟶

```
temp = 5;
for i=1 to 10 do
…
  x = temp
…
  temp = temp + 5
end
```

- Typical cases of strength reduction occurs in address calculation of array references.
- Applies to integer expressions involving induction variables (loop optimization)

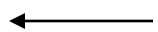
Dead Code Elimination

- Dead Code are portion of the program which will not be executed in any path of the program.
  - Can be removed
- Examples:
  - No control flows into a basic block
  - A variable is dead at a point -> its value is not used anywhere in the program
  - An assignment is dead -> assignment assigns a value to a dead variable

Dead Code Elimination

- Examples:

```
DEBUG:=0
if (DEBUG) print
```



Can be eliminated

Copy Propagation

- What does it mean?
  - Given an assignment x = y, replace later uses of x with uses of y, provided there are no intervening assignments to x or y.
- When is it performed?
  - At any level, but usually early in the optimization process.
- What is the result?
  - Smaller code

Copy Propagation

- `f := g` are called copy statements or copies
- Use of `g` for `f`, whenever possible after copy statement

Example:

x[i] = a;	x[i] = a;
sum = x[i] + a;	sum = a + a;

- May not appear to be code improvement, but opens up scope for other optimizations.

Loop Optimization

- Decrease the number if instruction in the inner loop
- Even if we increase no of instructions in the outer loop
- Techniques:
  - Code motion
  - Induction variable elimination
  - Strength reduction

Loop Optimization

consider the following C code snippet whose intention is to obtain the sum of all integers from 1 to N

```
int i,
sum = 0;
for (i = 1; i <= N; i++)
sum += i;
printf ("sum: %d\n", sum);
```

```
int sum = (N * (N+1)) >> 1;
printf ("sum: %d\n", sum);
```

Peephole Optimization

- Peephole optimization is very fast
  - Small overhead per instruction since they use a small, fixed-size window

- It is often easier to generate inexperienced code and run peephole optimization than generating good code!

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]

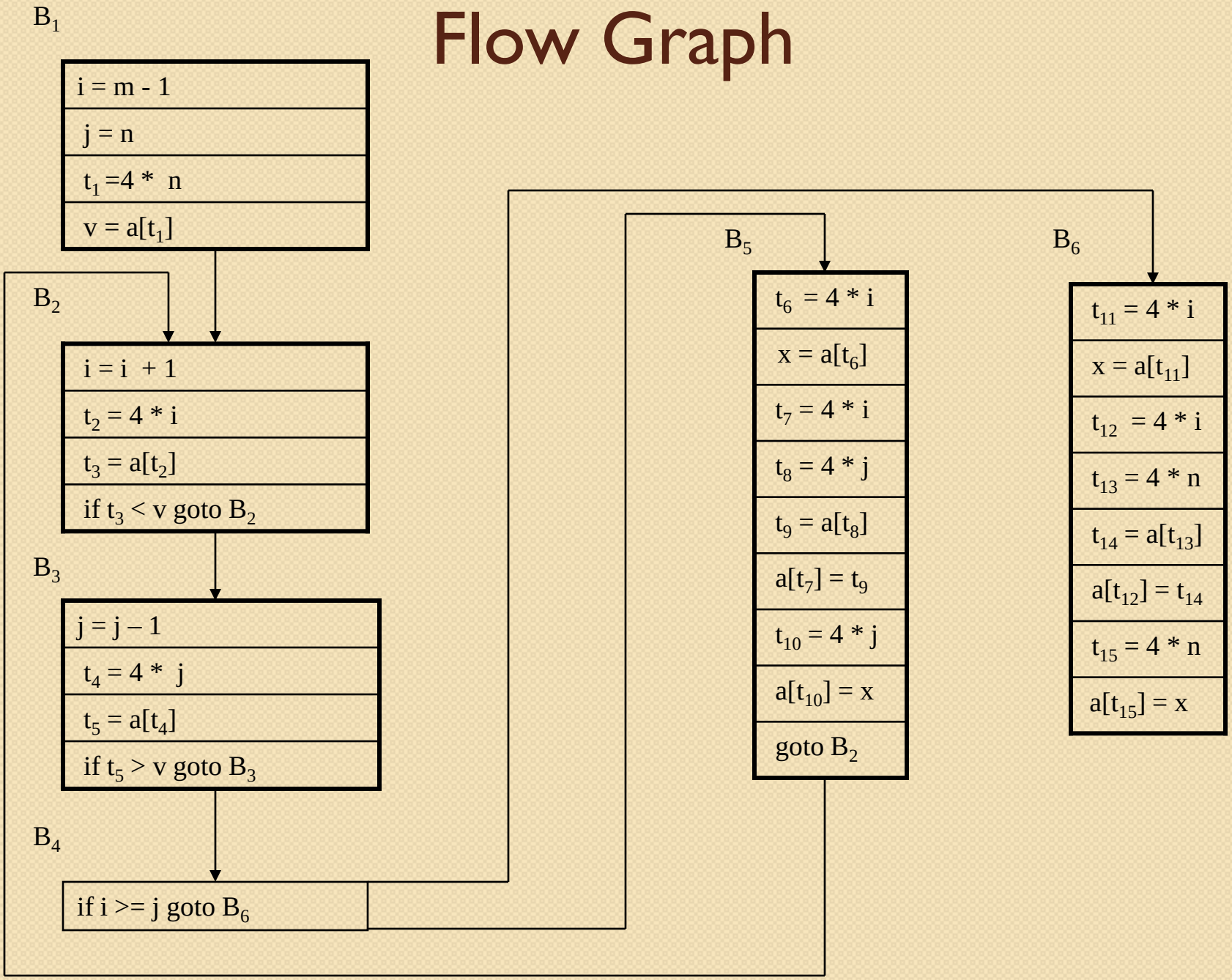
16	t_7 = 4 * I
17	t_8 = 4 * j
18	t_9 = a[t_8]
19	a[t_7] = t_9
20	t_{10} = 4 * j
21	a[t_{10}] = x
22	goto (5)
23	t_{11} = 4 * I
24	x = a[t_{11}]
25	t_{12} = 4 * i
26	t_{13} = 4 * n
27	t_{14} = a[t_{13}]
28	a[t_{12}] = t_{14}
29	t_{15} = 4 * n
30	a[t_{15}] = x

Find The Basic Block

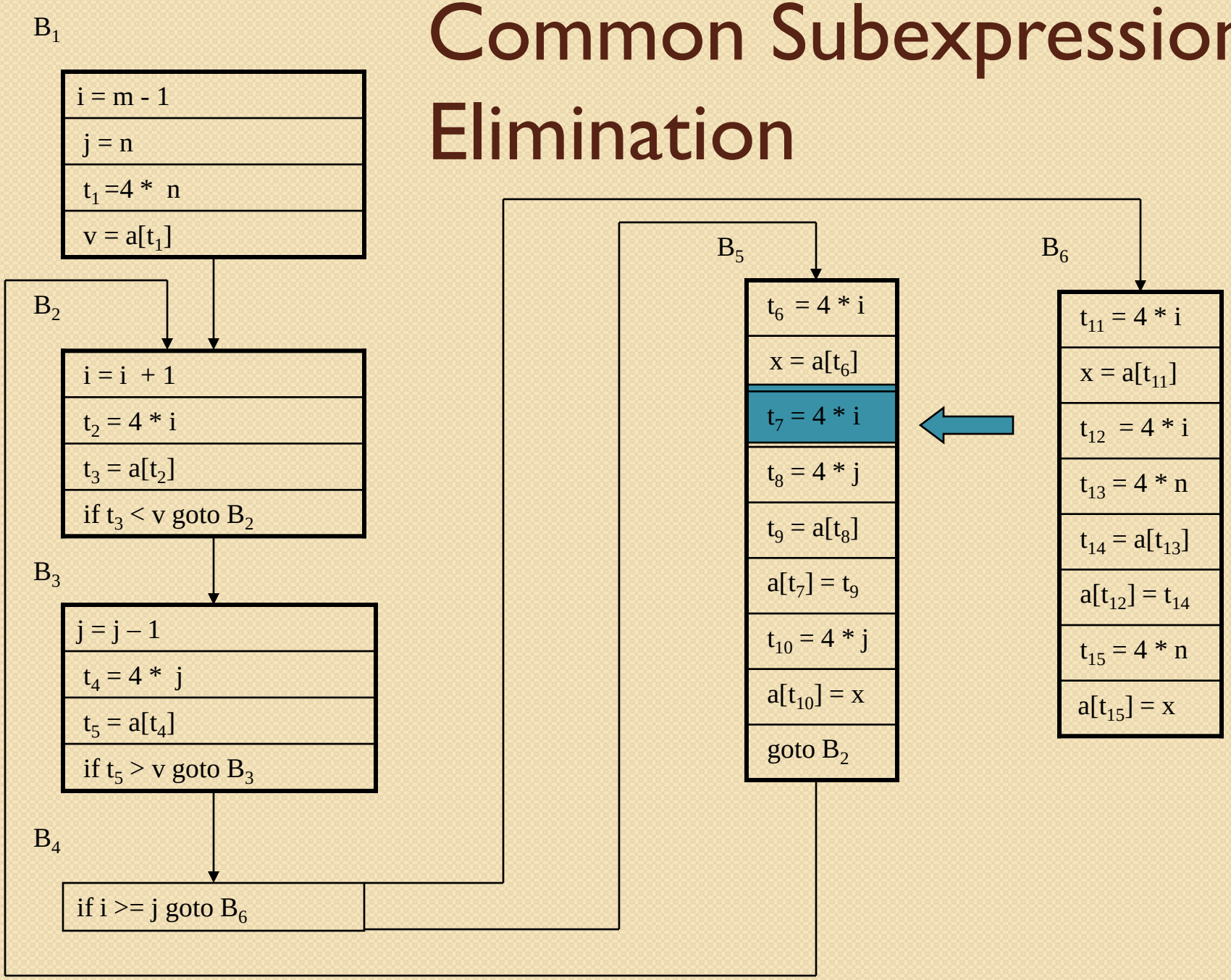
1	i = m - 1
2	j = n
3	t_1 =4 * n
4	v = a[t_1]
5	i = i + 1
6	t_2 = 4 * i
7	t_3 = a[t_2]
8	if t_3 < v goto (5)
9	j = j – 1
10	t_4 = 4 * j
11	t_5 = a[t_4]
12	if t_5 > v goto (9)
13	if i >= j goto (23)
14	t_6 = 4 * i
15	x = a[t_6]

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

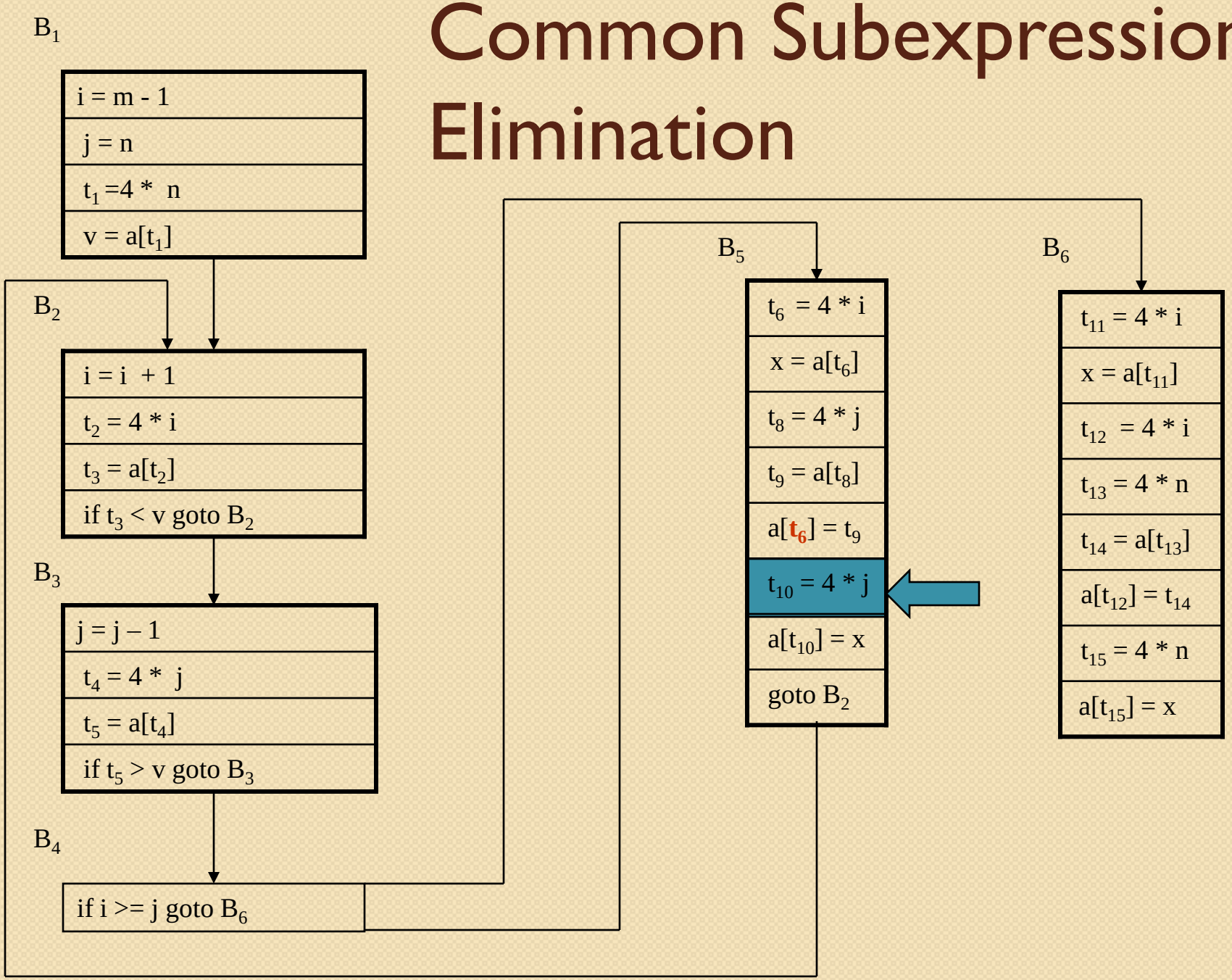
Flow Graph


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

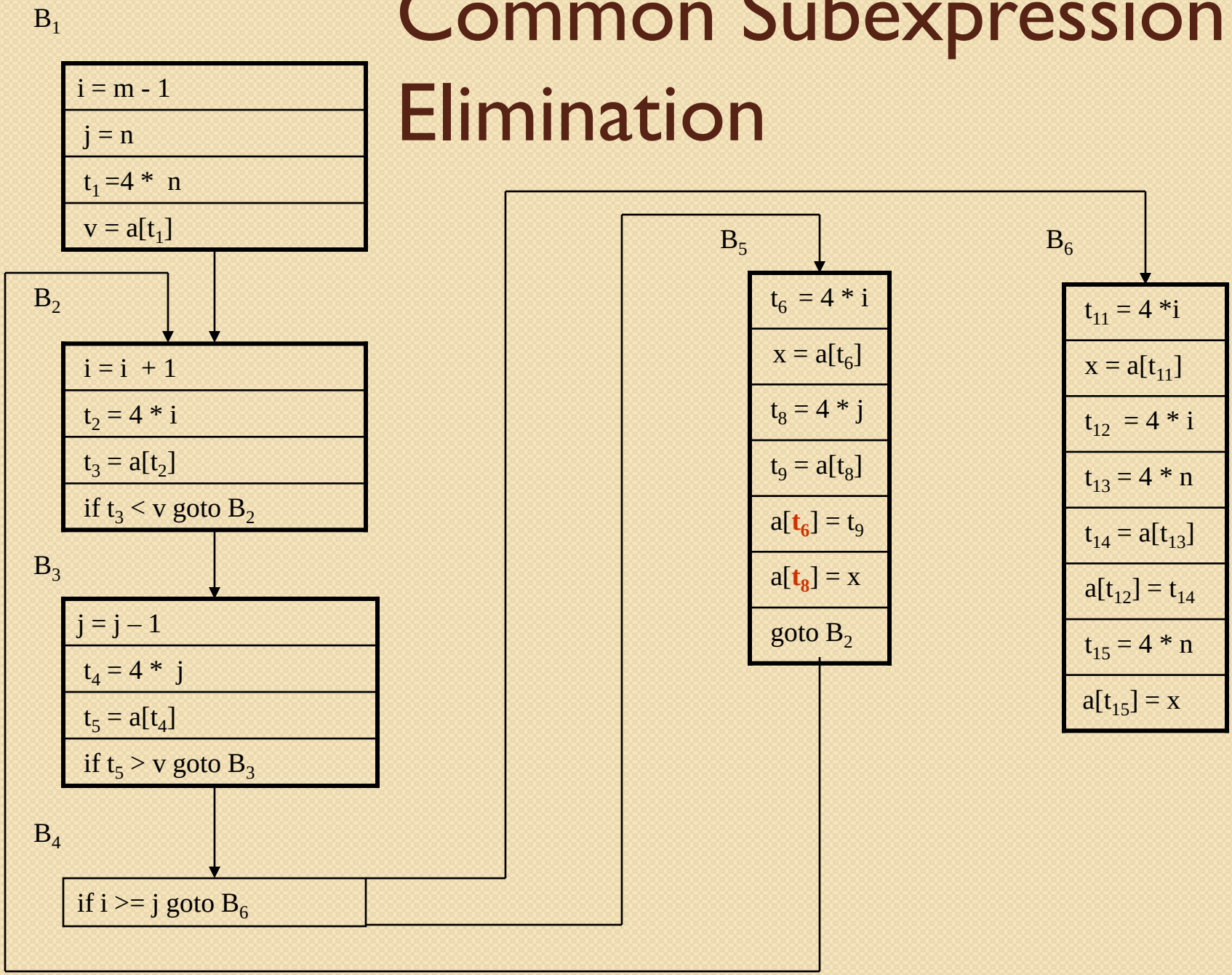
Common Subexpression Elimination



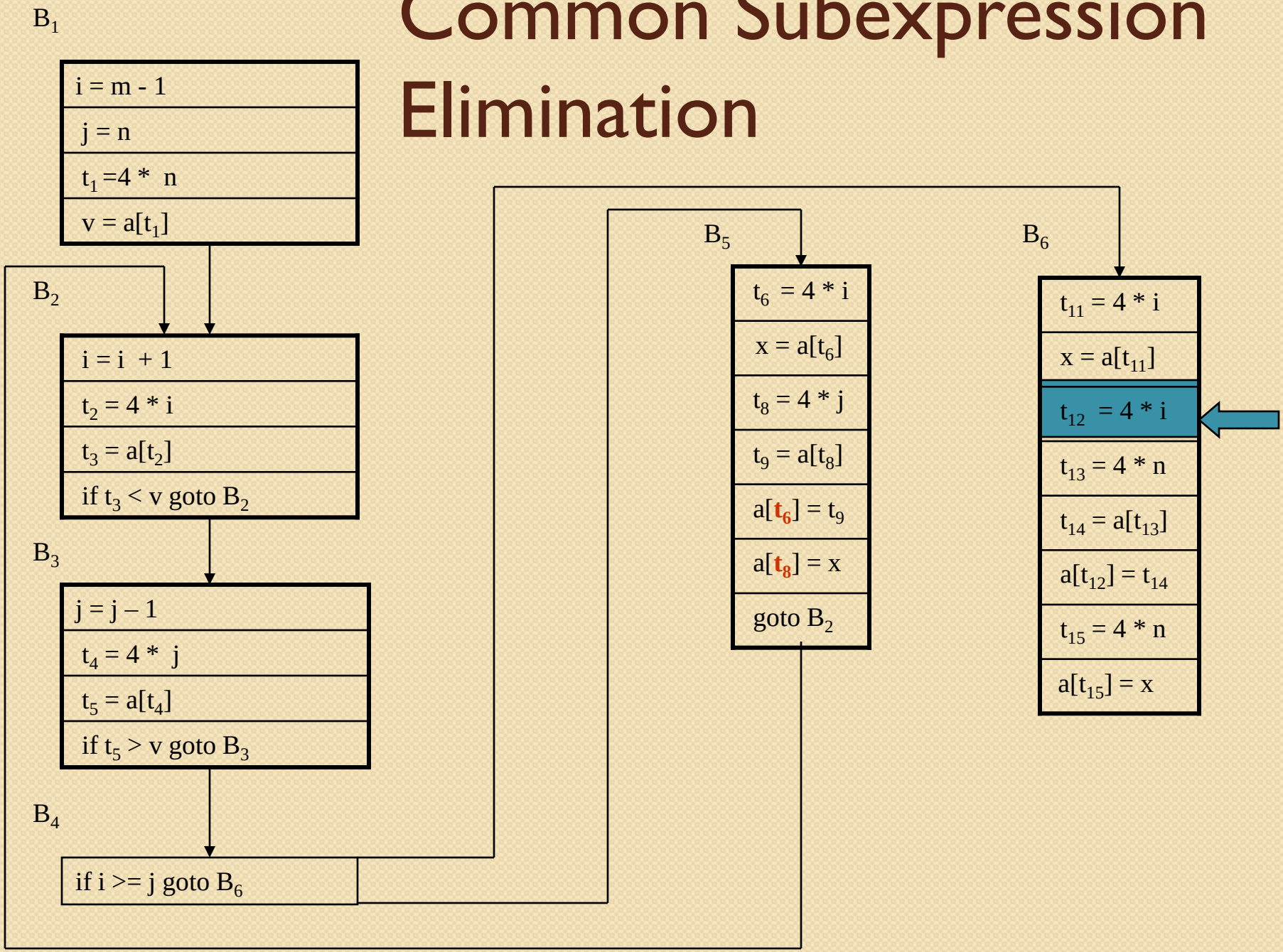
Common Subexpression Elimination



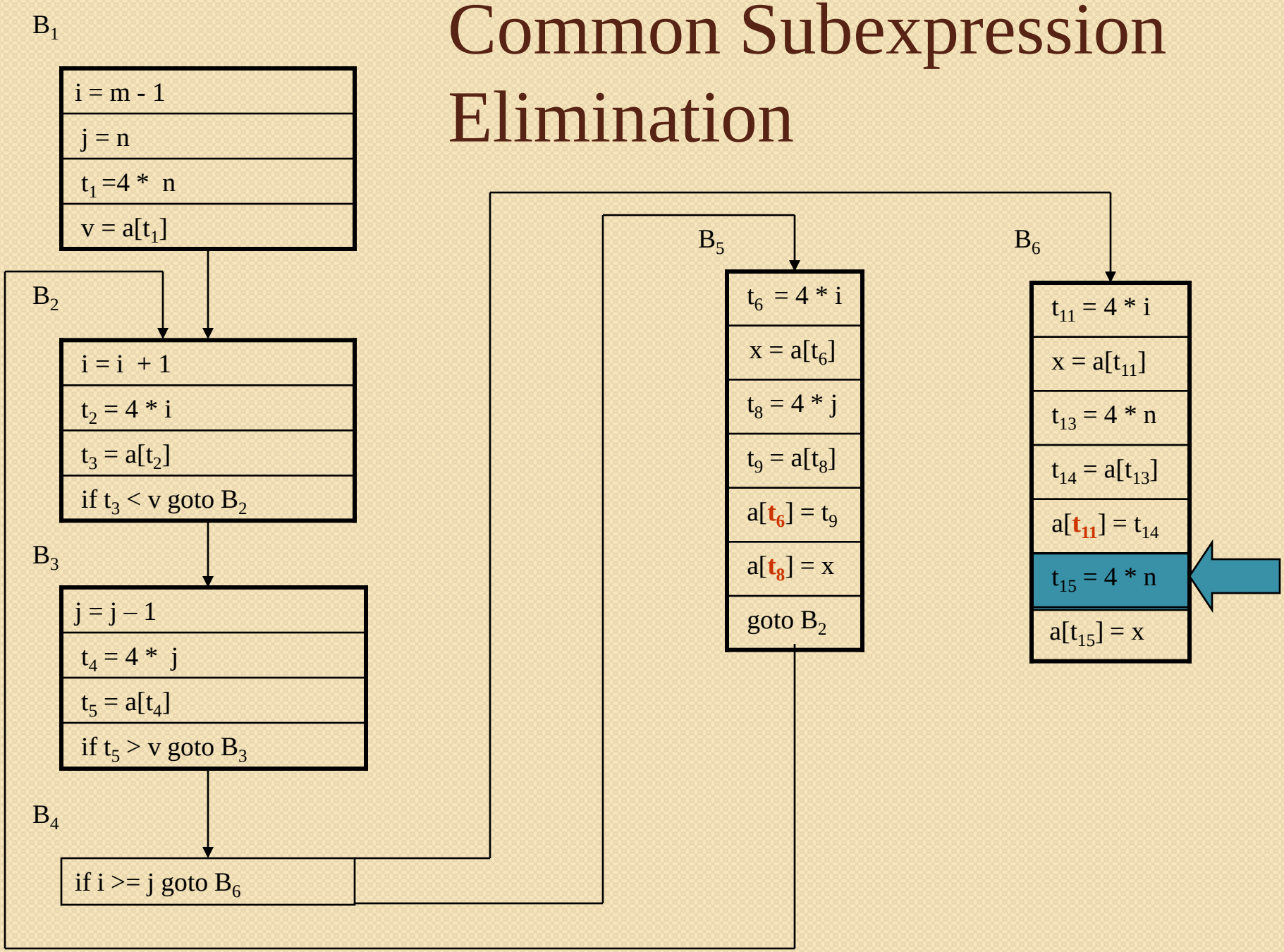
Common Subexpression Elimination



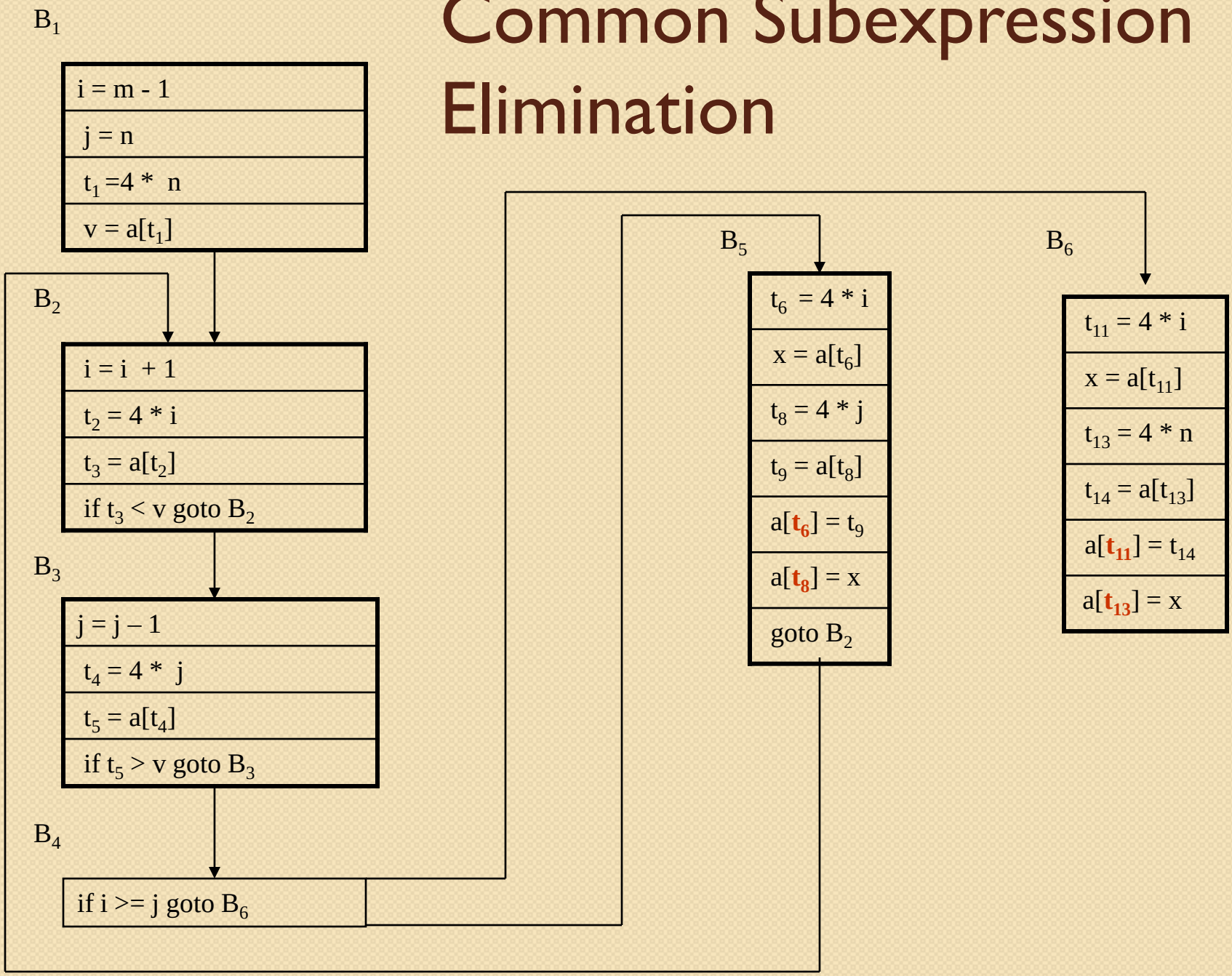
Common Subexpression Elimination



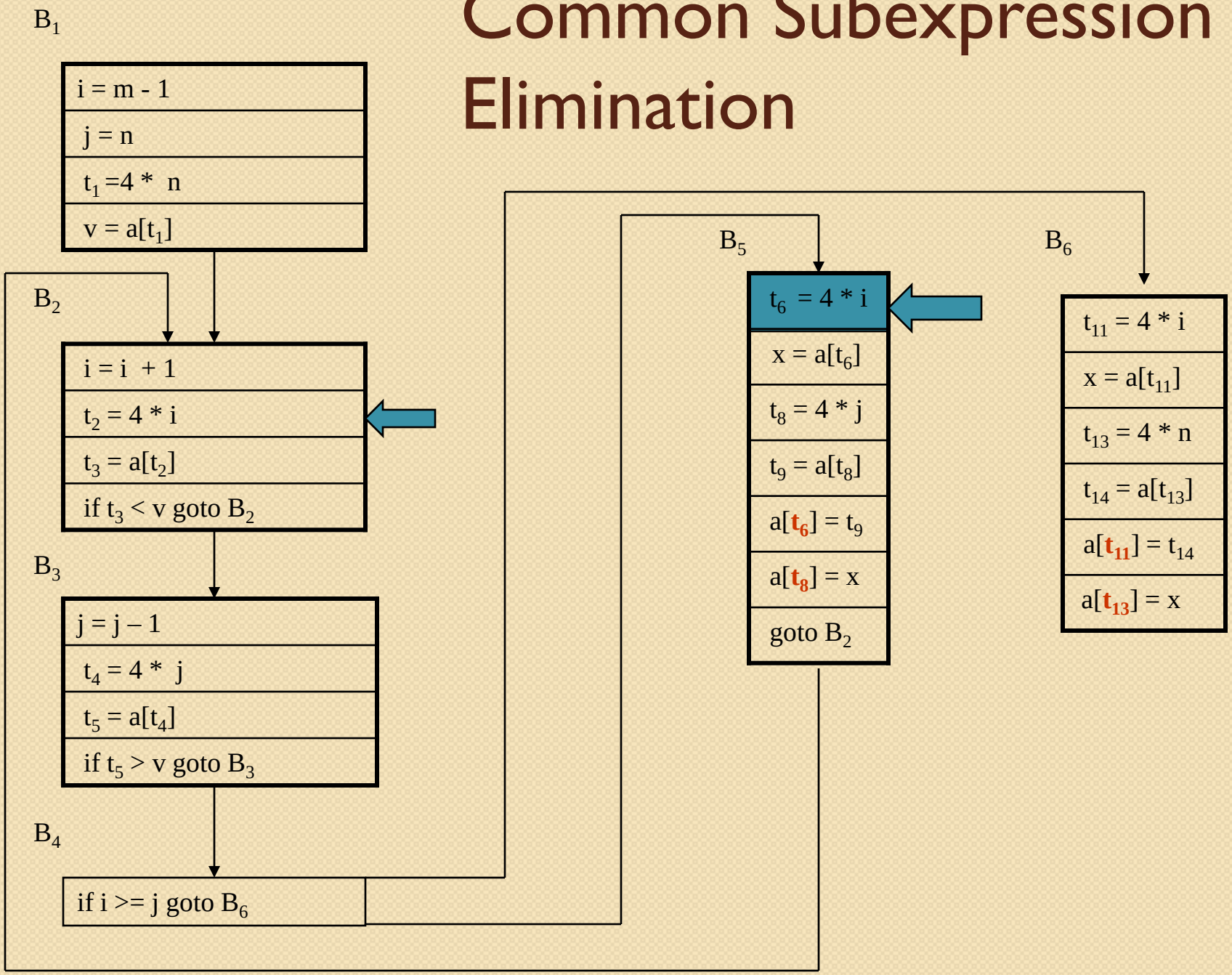
Common Subexpression Elimination



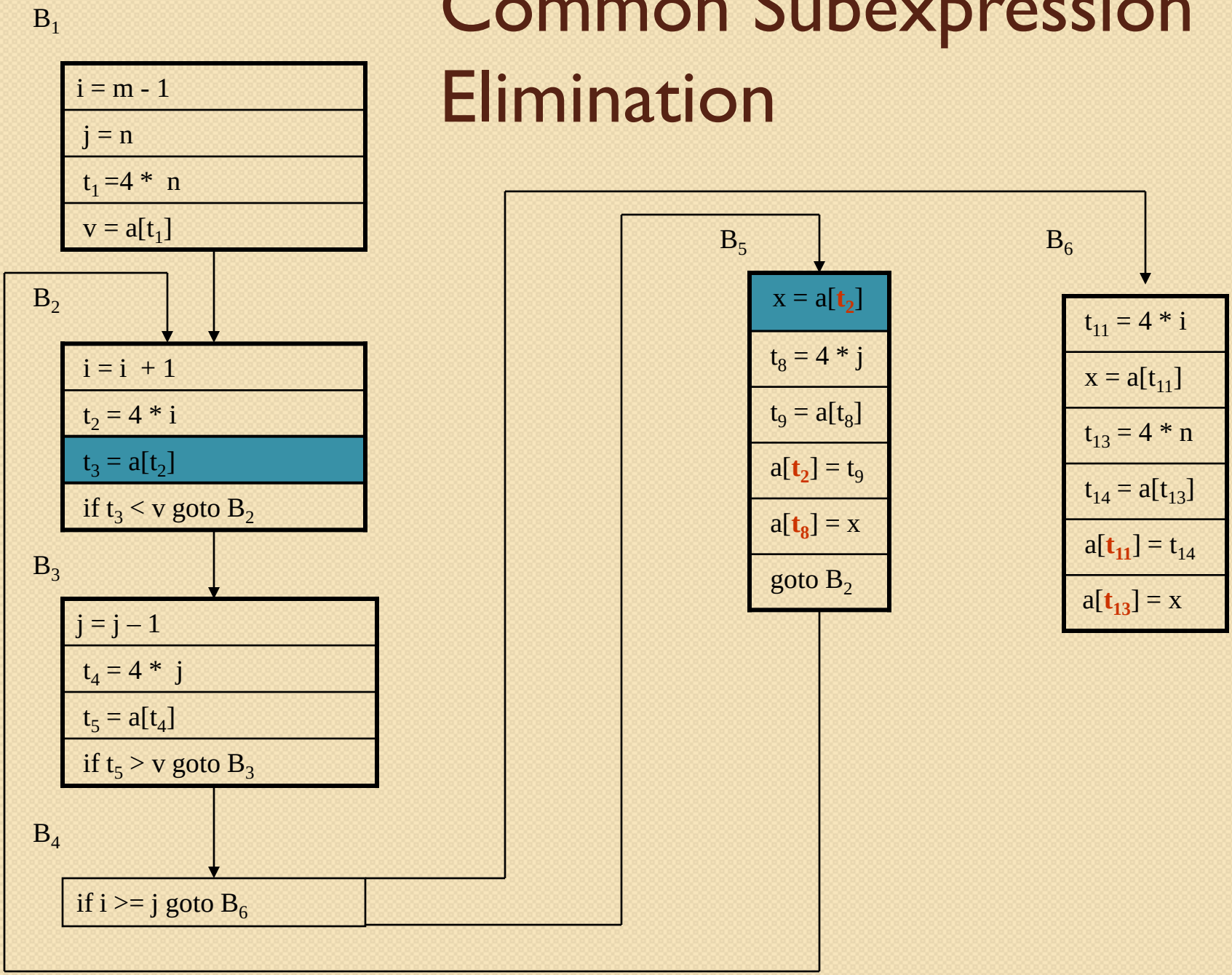
Common Subexpression Elimination



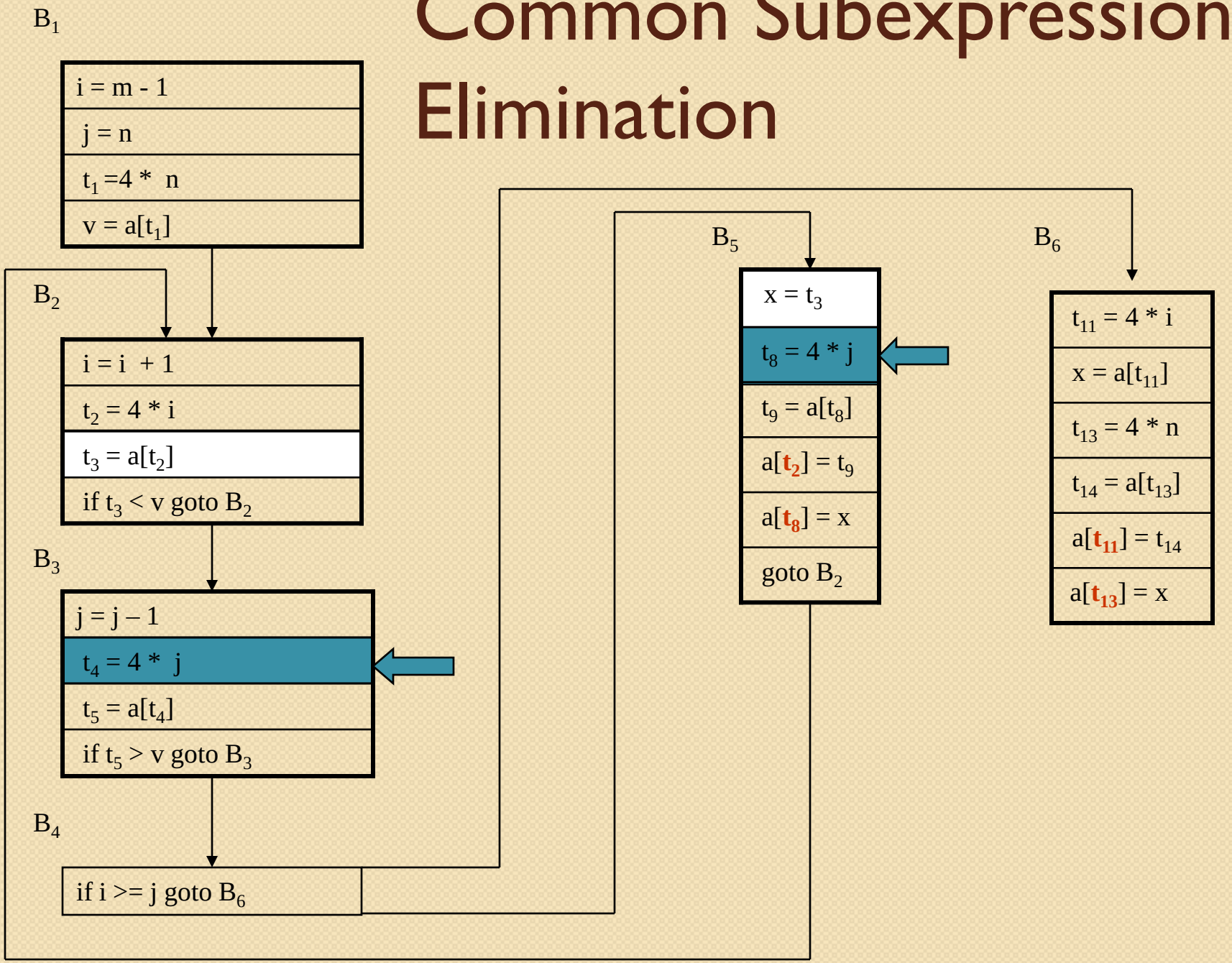
Common Subexpression Elimination



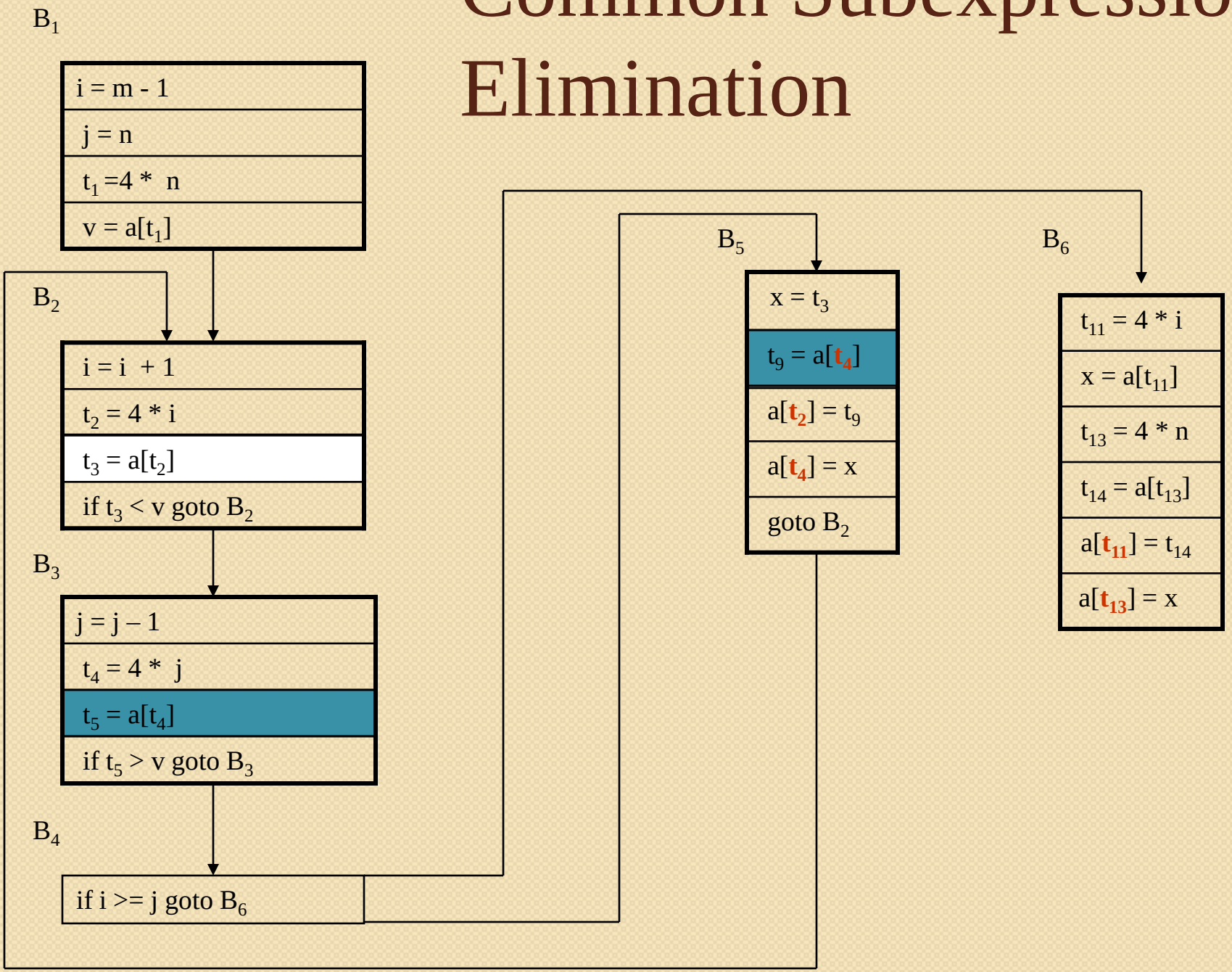
Common Subexpression Elimination



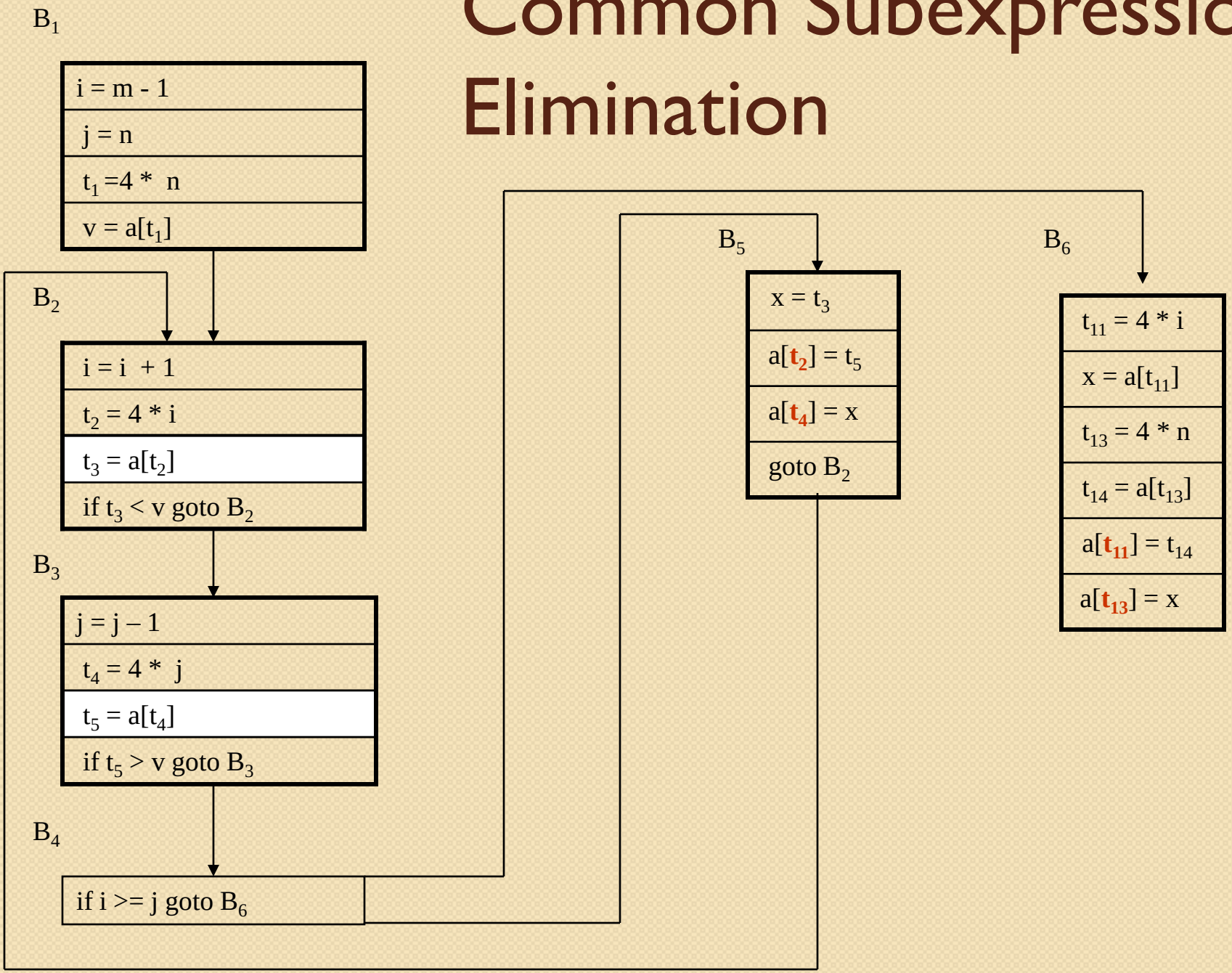
Common Subexpression Elimination



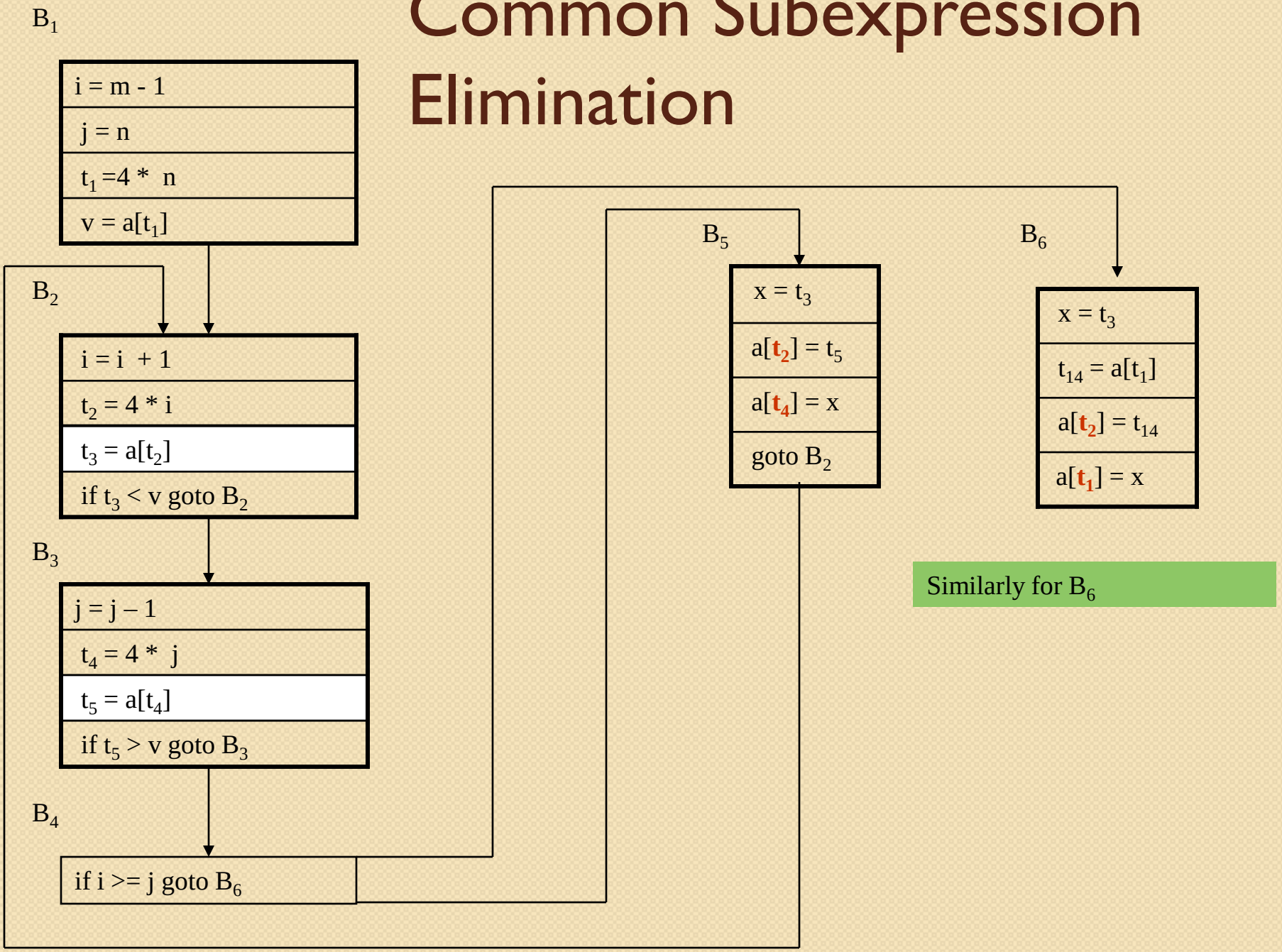
Common Subexpression Elimination



Common Subexpression Elimination

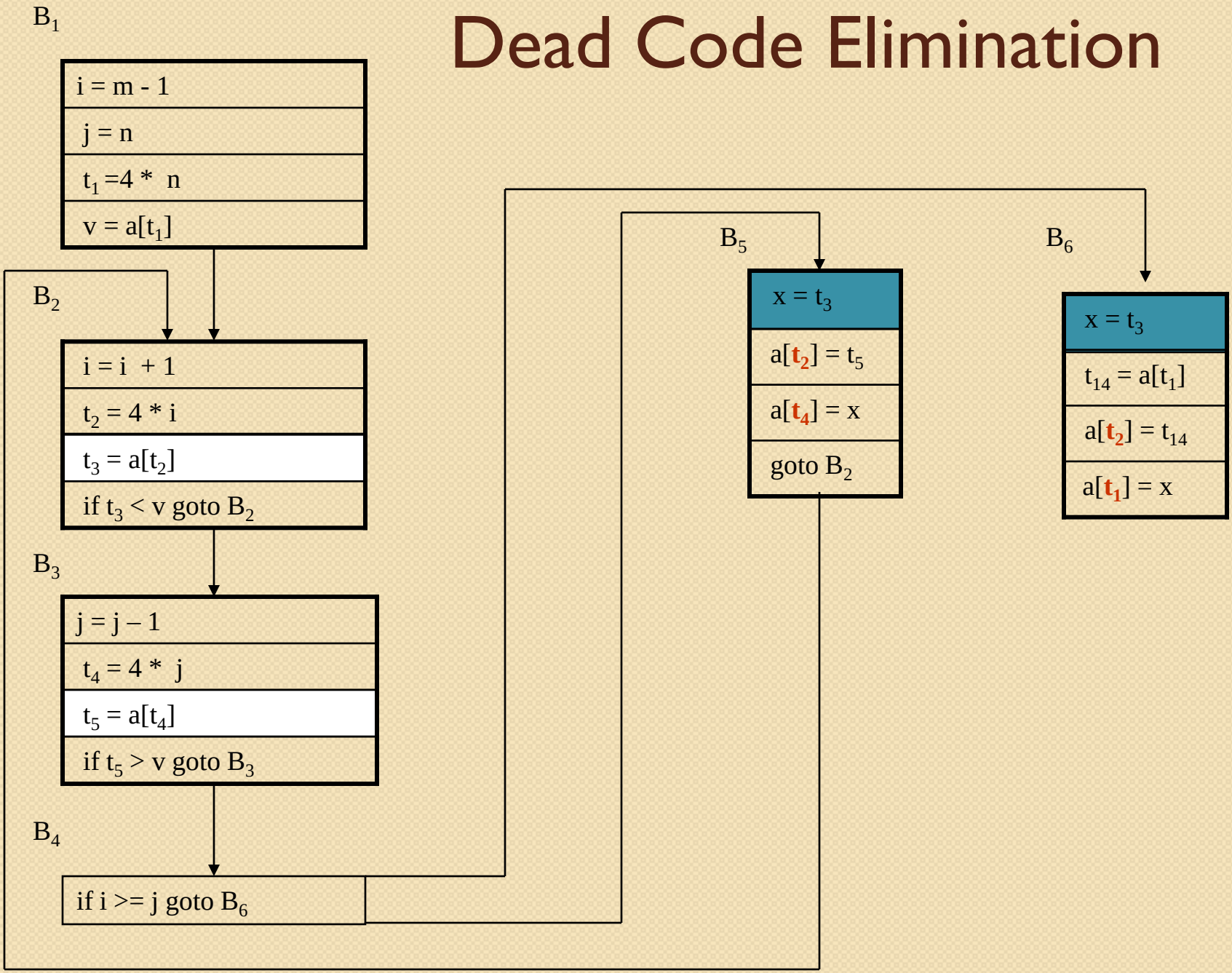


Common Subexpression Elimination

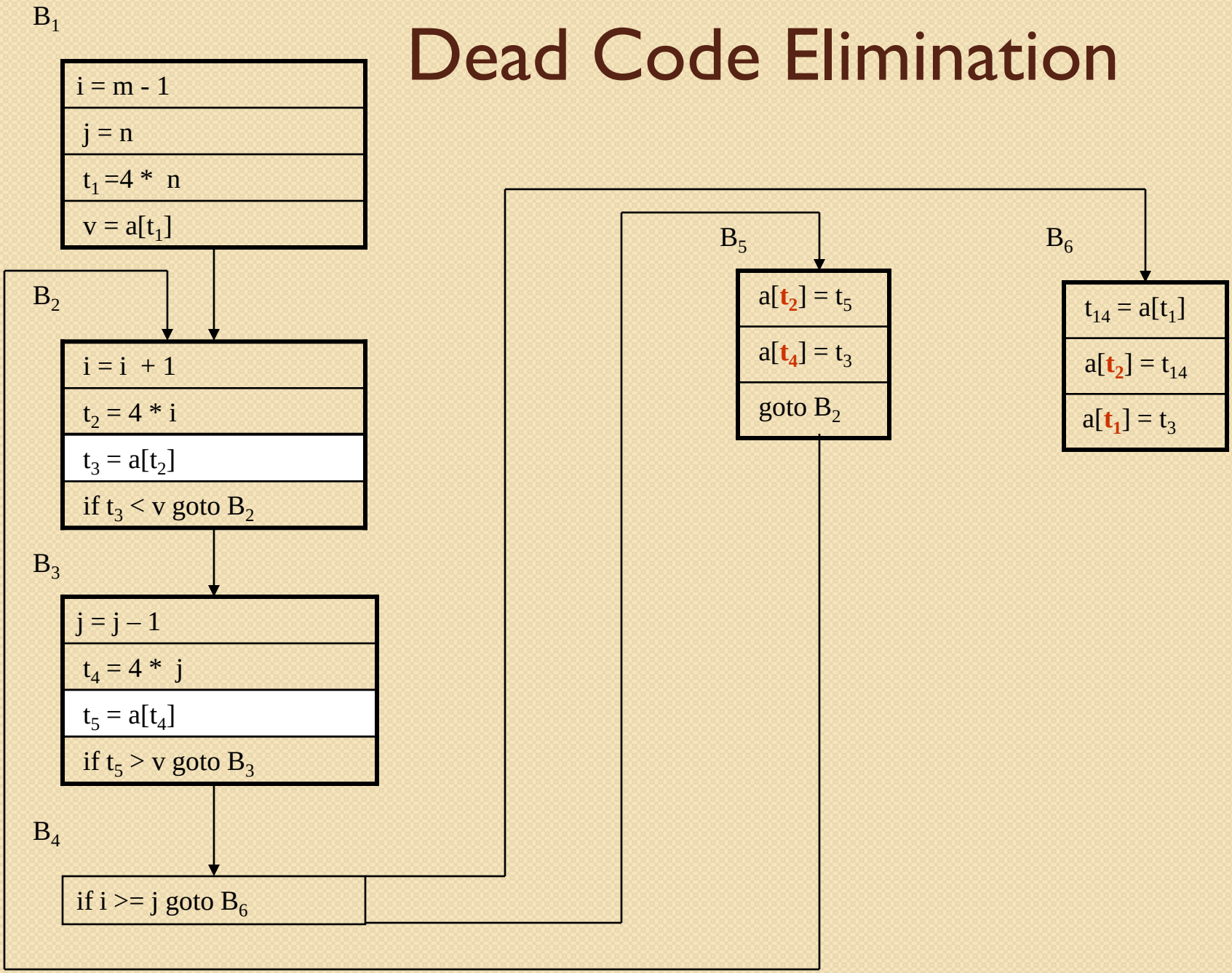


Similarly for B_6

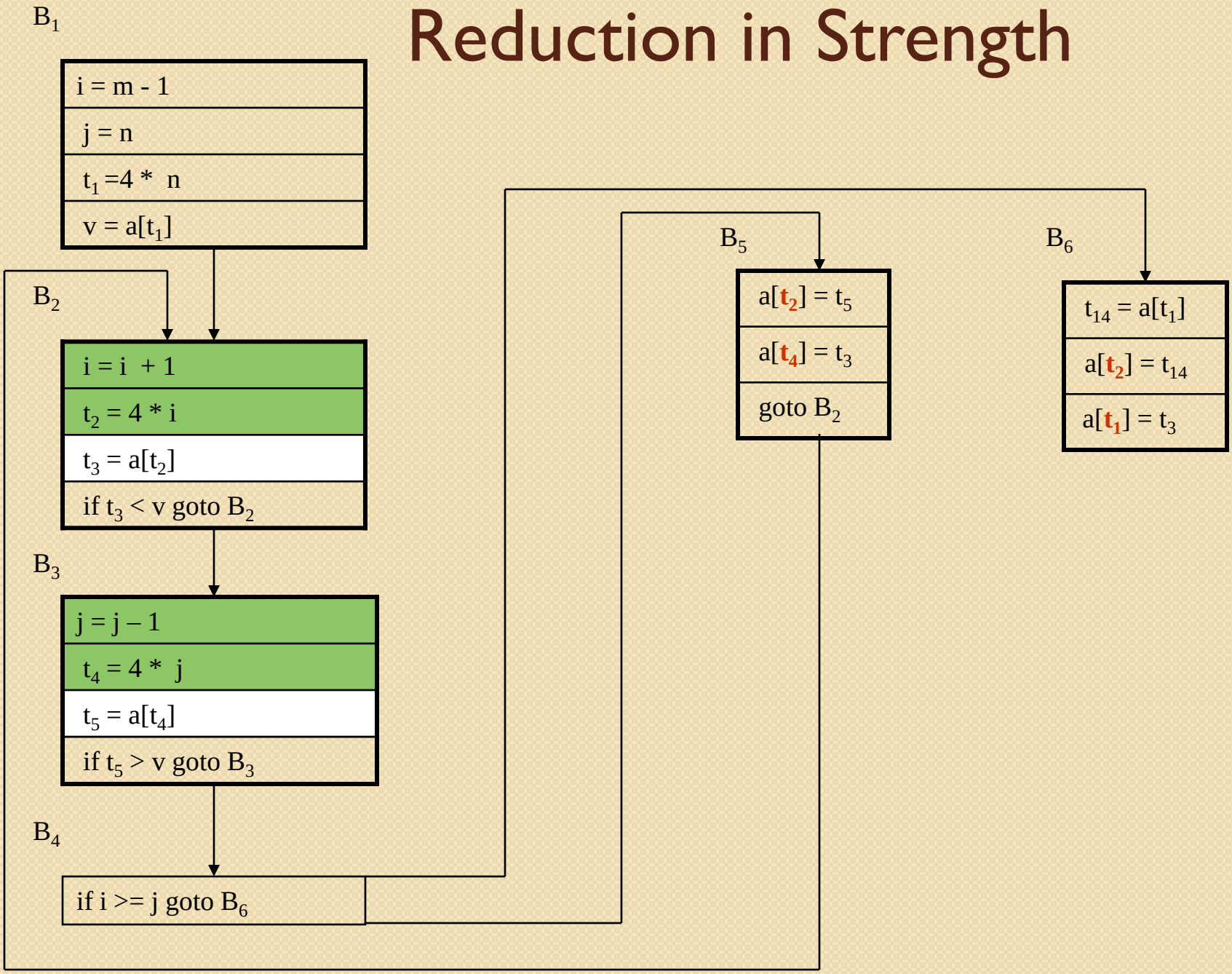
Dead Code Elimination



Dead Code Elimination



Reduction in Strength



Reduction in Strength