CODE OPTIMIZATION

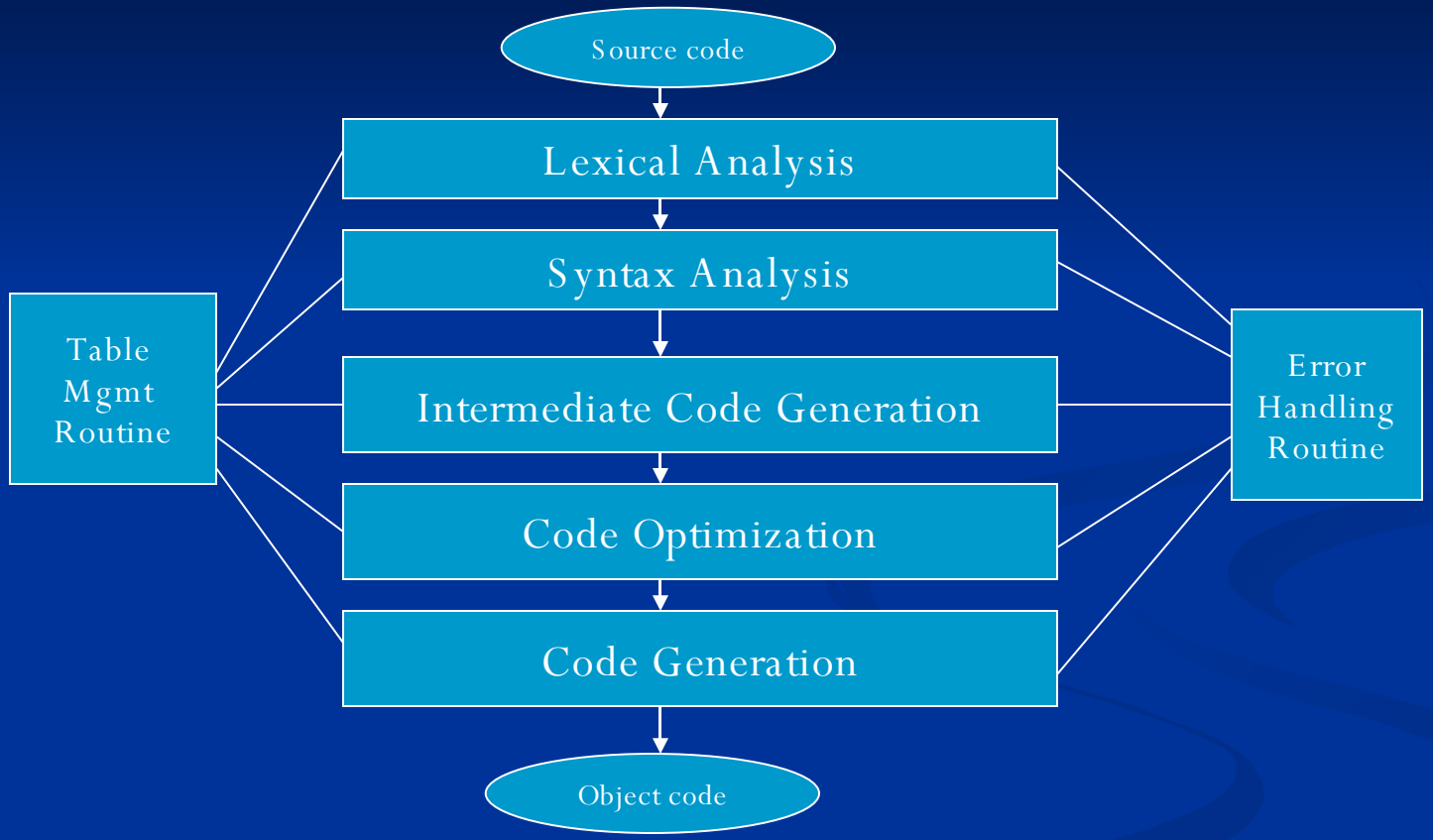
Presented By:

Amita das

Jayanti bhattacharya

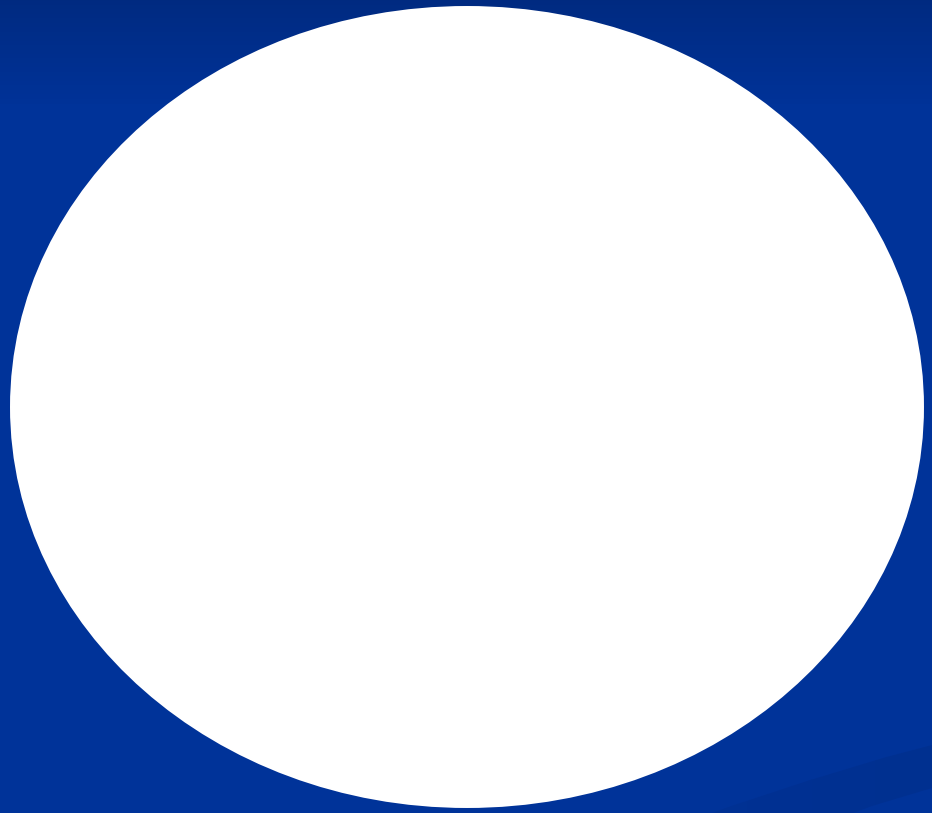
Jaistha Upadhyay

Design Of a Compiler



What is optimization?

- In computing, **optimization** is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources. For instance, 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 draw less power. The system may be a single computer program, a collection of computers or even an entire network such as the internet.



Levels' of optimization

Optimization can occur at a number of 'levels':

- **Design level**

  At the highest level, the design may be optimized to make best use of the available resources. The implementation of this design will benefit from the use of suitable efficient algorithms and the implementation of these algorithms will benefit from writing good quality code. The architectural design of a system overwhelmingly affects its performance. The choice of algorithm affects efficiency more than any other item of the design.

- **Compile level**

  Use of an optimizing compiler tends to ensure that the executable program is optimized at least as much as the compiler can predict.

- **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 since the programmer can take advantage of the full repertoire of machine instructions. The operating systems of most machines has been traditionally written in Assembler code for this reason.

- Runtime

  Just In Time Compiler and assembler programmers are able to perform runtime optimization.

When to optimize ?

Optimization is often performed at the end of the development stage_since it

- reduces readability
- adds code that is used to improve the performance.

Criteria For optimization

✓An optimization must preserve the meaning of a program :

-Cannot change the output produced for any input

-Can not introduce an error

✓optimization should, on average, speed up programs

✓Transformation should be worth the effort

Improvements can be made at various phases:

Source Code:

-Algorithms transformations can produce spectacular improvements

-Profiling can be helpful to focus a programmer's attention on important code.

Intermediate Code:

-Compiler can improve loops, procedure calls and address calculations

-Typically only optimizing compilers include this phase

Target Code:

-Compilers can use registers efficiently

-Peephole transformation can be applied

Types of Code optimization

- Common Sub-expression Removal
- Dead Code Optimization
- Loop Optimization

Common Sub expression elimination

Common Sub expression elimination is a optimization that searches for instances of identical expressions (i.e they all evaluate the same value), and analyses whether it is worthwhile replacing with a single variable holding the computed value.

```
a=b * c + g

d=b * c * d
```

```
temp=b * c

a=temp + g

d=temp * d
```

Dead code Optimization:

Dead Code elimination is a compiler optimization that removes code that does not affect a program. Removing such code has two benefits It shrinks program size, an important consideration in some contexts. It lets the running program avoid executing irrelevant operations, which reduces its running time.

Dead Code elimination is of two types

Unreachable Code

Redundant statement

Unreachable Code

In Computer Programming, Unreachable Code or dead code is code that exists in the source code of a program but can never be executed.

Program Code

```
If (a>b)
m=a
elseif (a<b)
m=b
elseif (a==b)
m=0
else
m=-1
```

Optimized Code

```
If (a>b)
m=a
elseif (a<b)
m=b
else
m=0
```

Redundant Code

Redundant Code is code that is executed but has no effect on the output from a program

```
main(){
int a,b,c,r;
a=5;
b=6;
c=a +b;
r=2;
r++;
printf("%d",c);
}
```

Adding time & space complexity

Loop optimization

Loop optimization plays an important role in improving the performance of the source code by reducing overheads associated with executing loops.

Loop Optimization can be done by removing:

- Loop invariant
- Induction variables

Loop Invariant

```
i = 1
s = 0
do {
s = s + i
a = 5
i = i + 1
{
while (i <= n)
```

```
i = 1
s = 0
a = 5
do {
s = s + i
i = i + 1
{
while (i <= n)
```

Bringing a=5 outside the do while loop, is called code motion.

Induction variables

```
i = 1
s= 0
S1=0
S2=0
while (i < =n)
{
s= s + a[ i ]
t1 = i * 4
s= s + b[ t1 ]
t2 = t1 +2
s2= s2 + c[ t2 ]
i = i + 1
}
```

```
i = 1
s= 0
S1=0
S2=0
t2=0
while (i < =n)
{
s= s + a[ i ]
t1 = t1+ 4
s= s + b[ t1 ]
s2= s2 + c[t1 +2 ]
i = i + 1
}
```

“+” replaced “ * ”, t1 was made independent of i

t1,t2 are induction variables. i is inducing t1 and t1 is inducing t2

Quicksort in C

```
void quicksort(int m, int n)
{  int i, j, v, x;

   if (n <= m) return;
   /* Start of partition code */
   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;
   /* End of partition code */
   quicksort(m, j); quicksort(i+1, n);
} /* quicksort */
```

Three-Address Code

- Find the basic blocks
- Draw the control flow graph

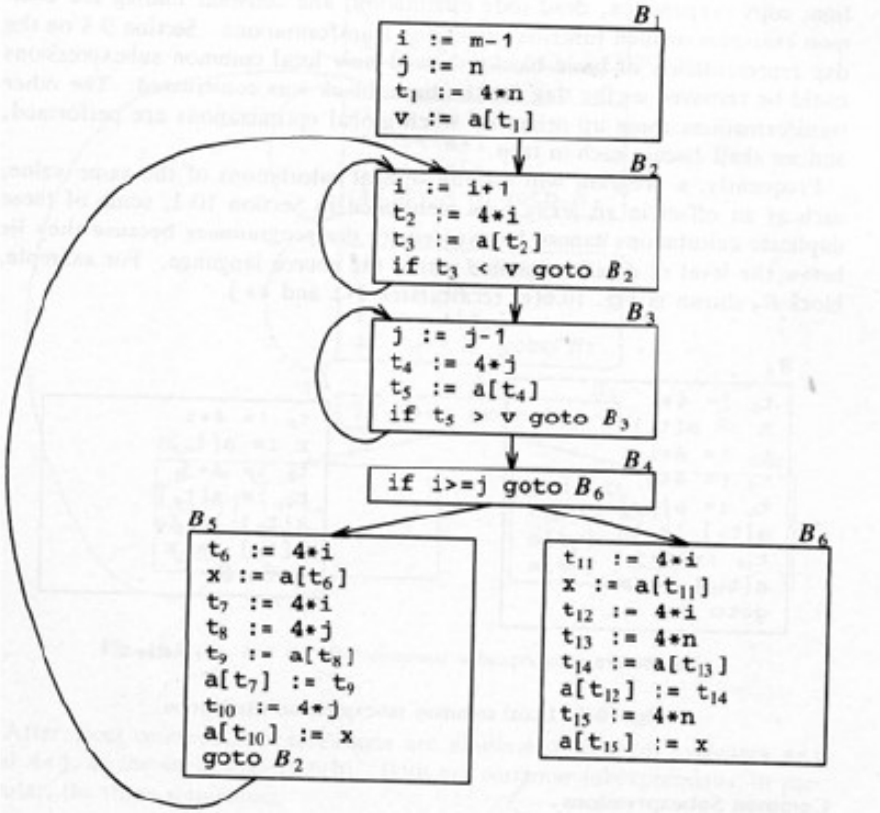
```
 (1) i := m-1
 (2) j := n
 (3) t1 := 4*n
 (4) v := a[t1]
 (5) i := i+1
 (6) t2 := 4*i
 (7) t3 := a[t2]
 (8) if t3 < v goto (5)
 (9) j := j-1
(10) t4 := 4*j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4*i
(15) x := a[t6]
```

```
(16) t7 := 4*i
(17) t8 := 4*j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4*j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4*i
(24) x := a[t11]
(25) t12 := 4*i
(26) t13 := 4*n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4*n
(30) a[t15] := x
```

Control Flow Graph

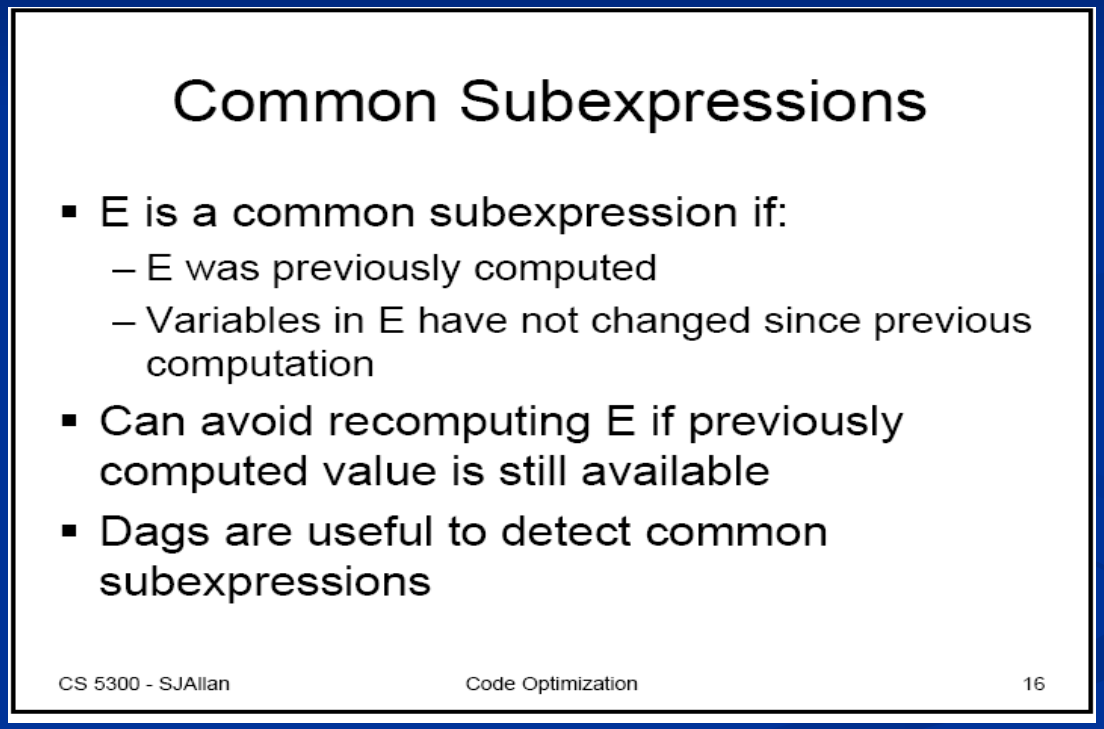
- Nodes represent basic blocks
- The initial node is the block whose leader is the first statement
- There exists an edge from B_i to B_j if:
  - There is a conditional or unconditional jump from the last statement in B_i to the first statement in B_j
  - B_j immediately follows B_i in the order of the program, and B_i does not end in an unconditional jump

Flow Graph

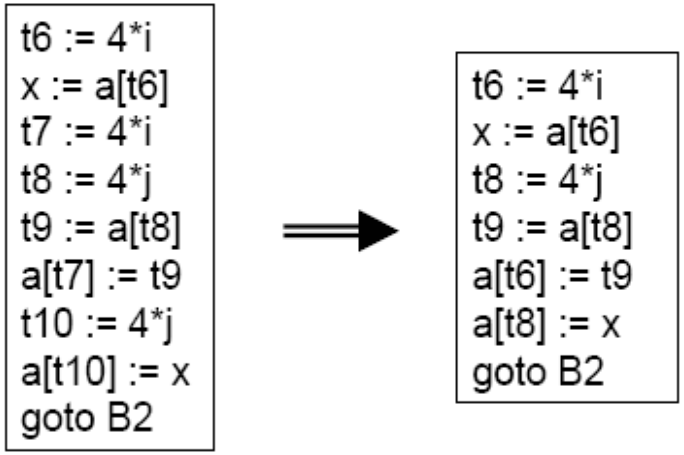

B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
B2
i := i+1
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2
B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
B4
if i>=j goto B6
B5
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
B6
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x

Common Sub-expression Removal

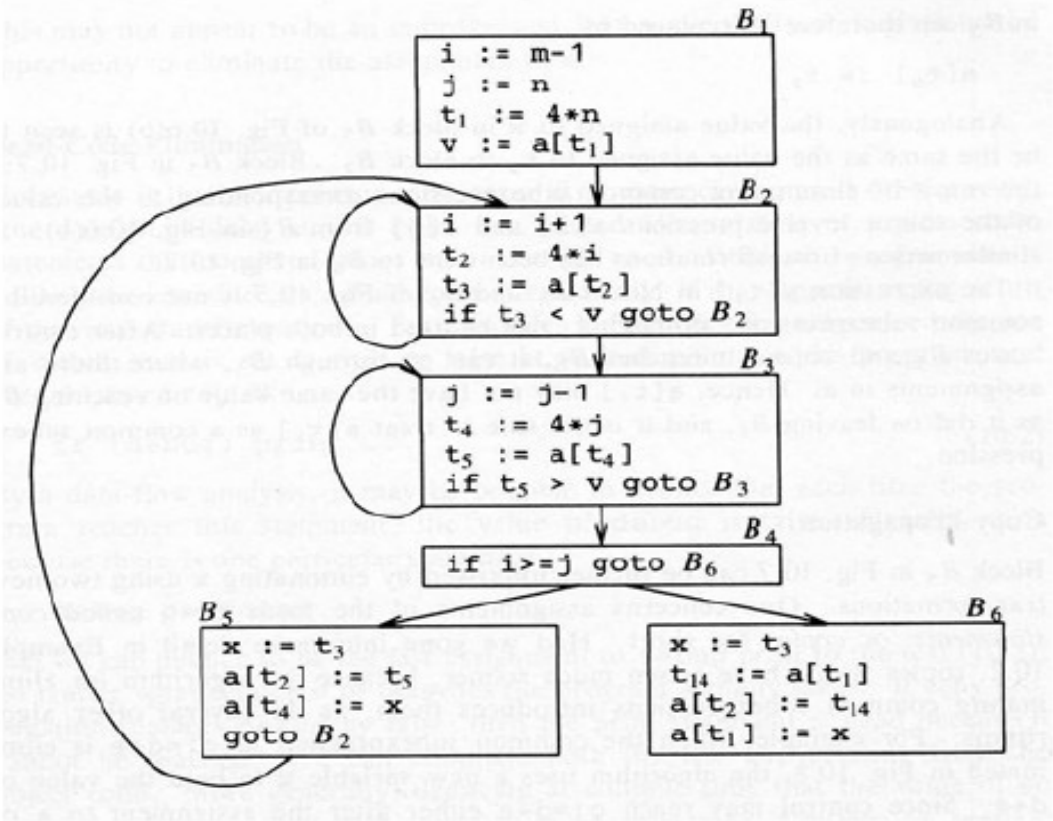
- It is used to remove redundant computations which usually improves the execution time of a program.



Local Common Subexpressions



Global Common Subexpressions



Copy Propagation

- Assignments of the form f := g are called copy statements (or copies)
- The idea behind copy propagation is to use g for f whenever possible after such a statement
- For example, applied to block B5 of the previous flow graph, we obtain:

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

- Copy propagation often turns the copy statement into "dead code"

Loop Optimizations

- The running time of a program may be improved if we do both of the following:
  - Decrease the number of statements in an inner loop
  - Increase the number of statements in the outer loop
- Code motion moves code outside of a loop
  - For example, consider:

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

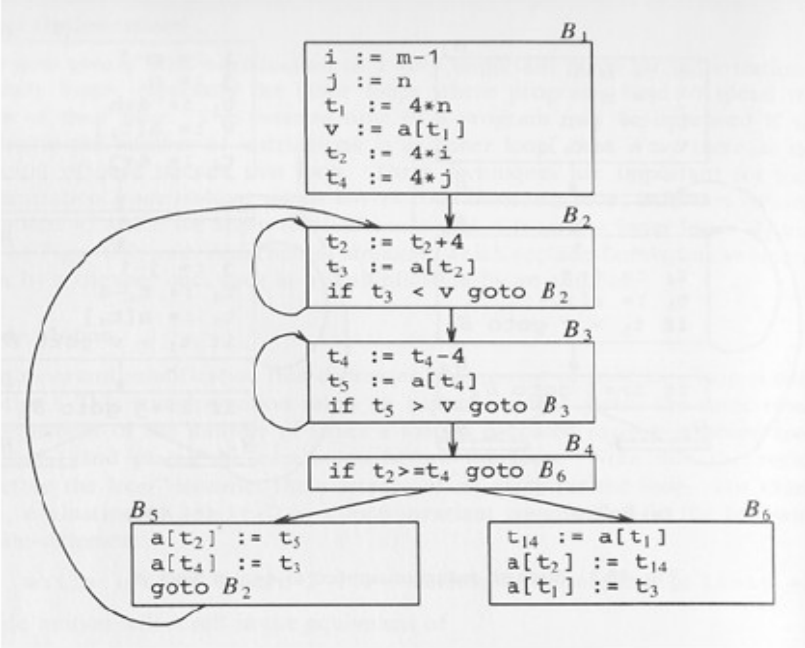
  - The result of code motion would be:

    ```
    t = limit – 2
    while (i <= t) …
    ```

Loop Optimizations

- Induction variables: variables that remain in "lock-step"
  - For example, in block B3 of previous flow graph, j and t4 are induction variables
  - Induction-variable elimination can sometimes eliminate all but one of a set of induction variables
- Reduction in strength replaces a more expensive operation with a less expensive one
  - For example, in block B3, t4 decreases by four with every iteration
  - If initialized correctly, can replace multiplication with subtraction
  - Often application of reduction in strength leads to induction-variable elimination
- Methods exist to recognize induction variables and apply appropriate transformations automatically

Loop Optimization Example



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	$x = a[t_6]$

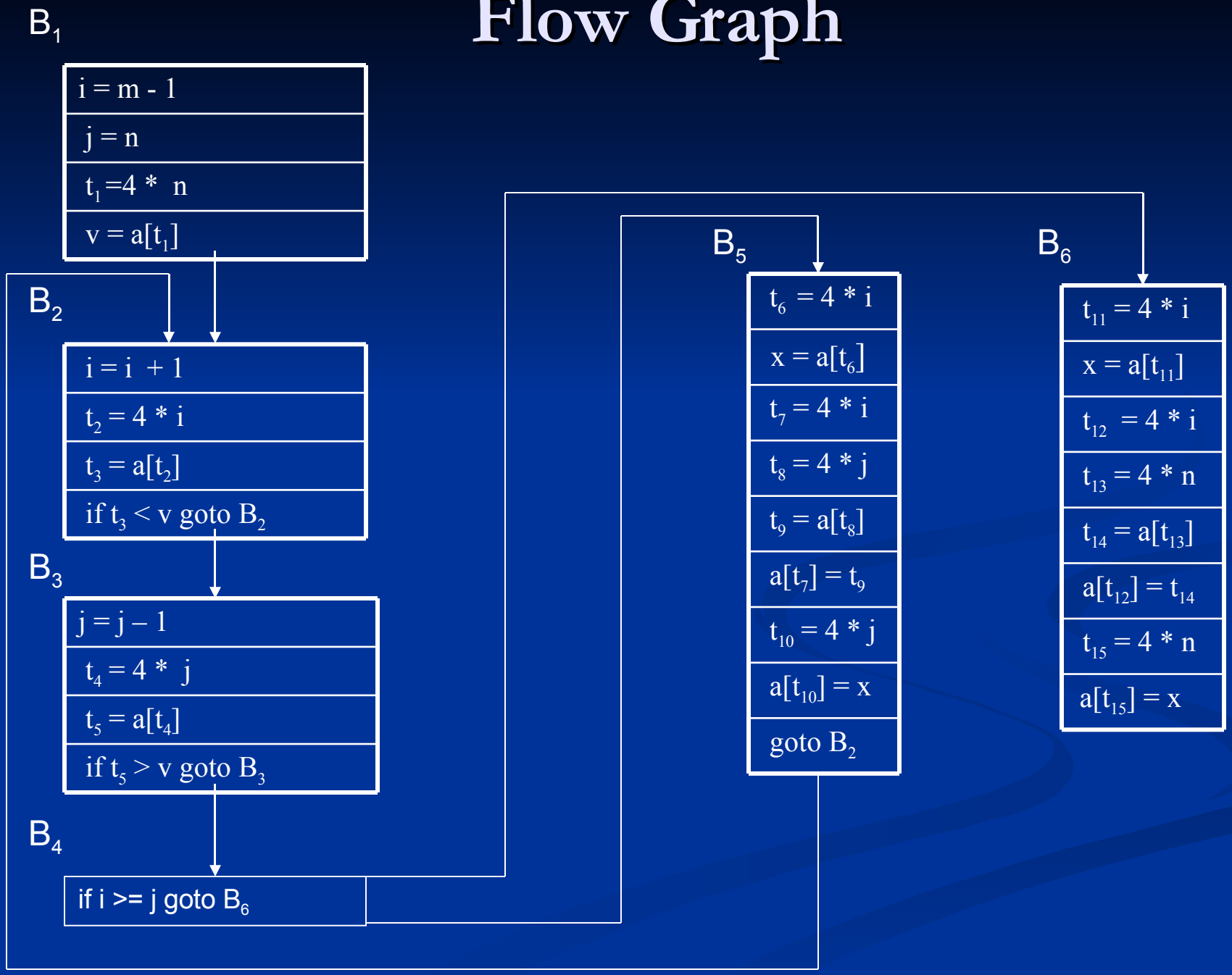
16	$t_7 = 4 * I$
17	$t_8 = 4 * j$
18	$t_9 = a[t_8]$
19	$a[t_7] = t_9$
20	$t_{10} = 4 * j$
21	$a[t_{10}] = x$
22	goto (5)
23	$t_{11} = 4 * I$
24	$x = a[t_{11}]$
25	$t_{12} = 4 * i$
26	$t_{13} = 4 * n$
27	$t_{14} = a[t_{13}]$
28	$a[t_{12}] = t_{14}$
29	$t_{15} = 4 * n$
30	$a[t_{15}] = x$

Find The Basic Block

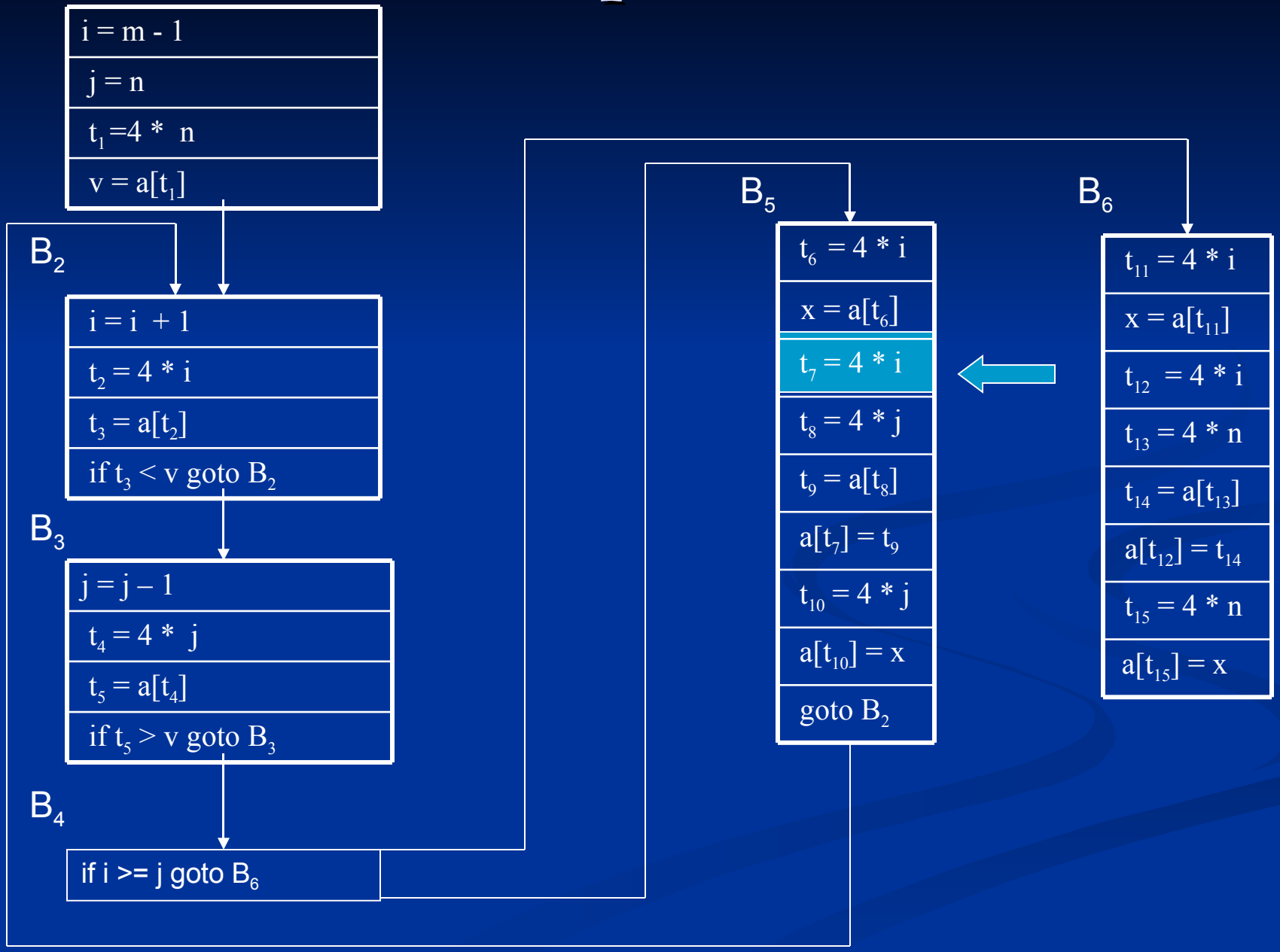
1	i = m - 1
2	j = n
3	t_1 =4 * n
4	v = a[t_1]
5	i = i + 1
6	t_2 = 4 * i
7	t_3 = a[t_2]
8	if t_3 < v goto (5)
9	j = j – 1
10	t_4 = 4 * j
11	t_5 = a[t_4]
12	if t_5 > v goto (9)
13	if i >= j goto (23)
14	t_6 = 4 * i
15	x = a[t_6]

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

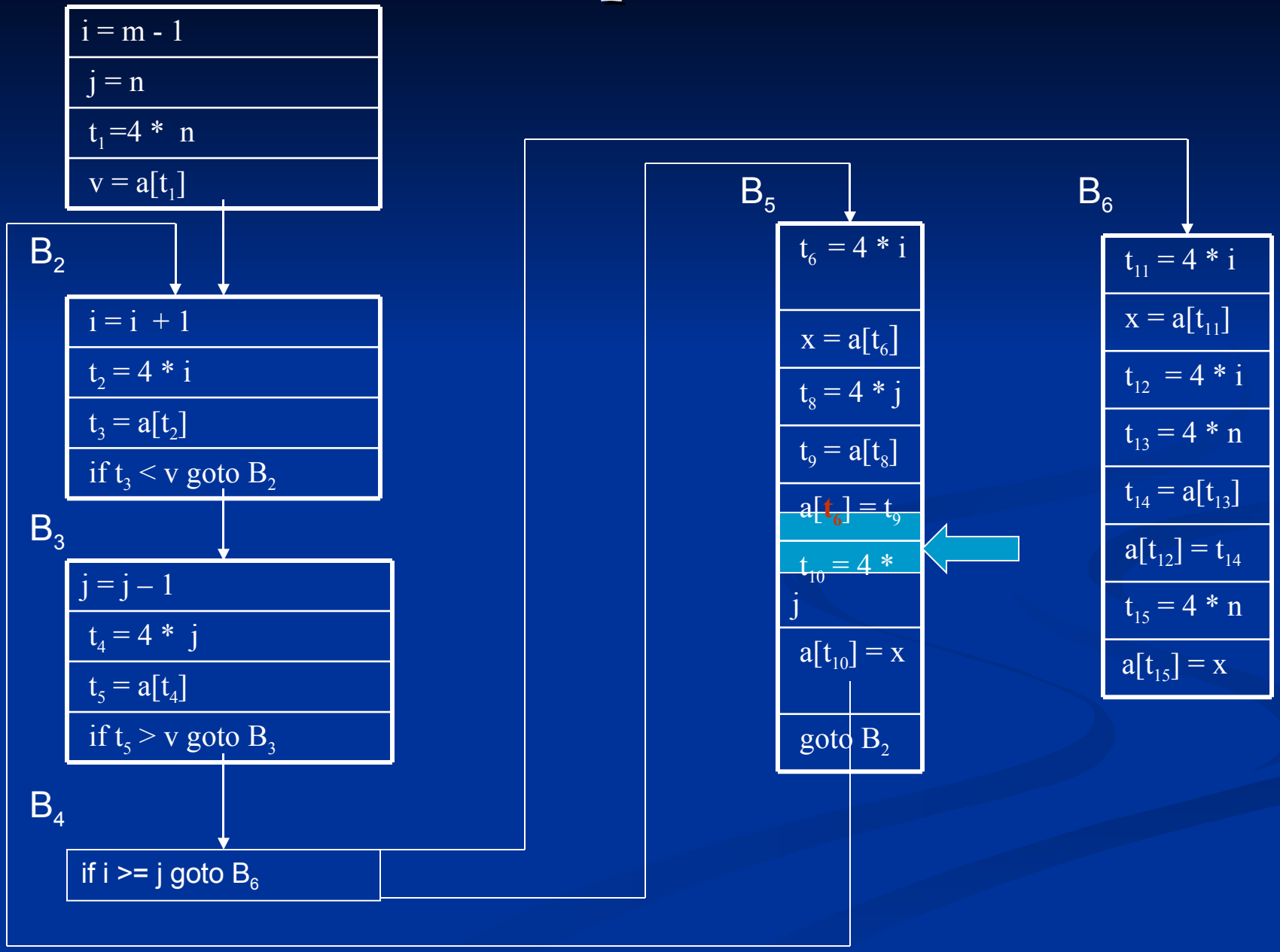
Flow Graph


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

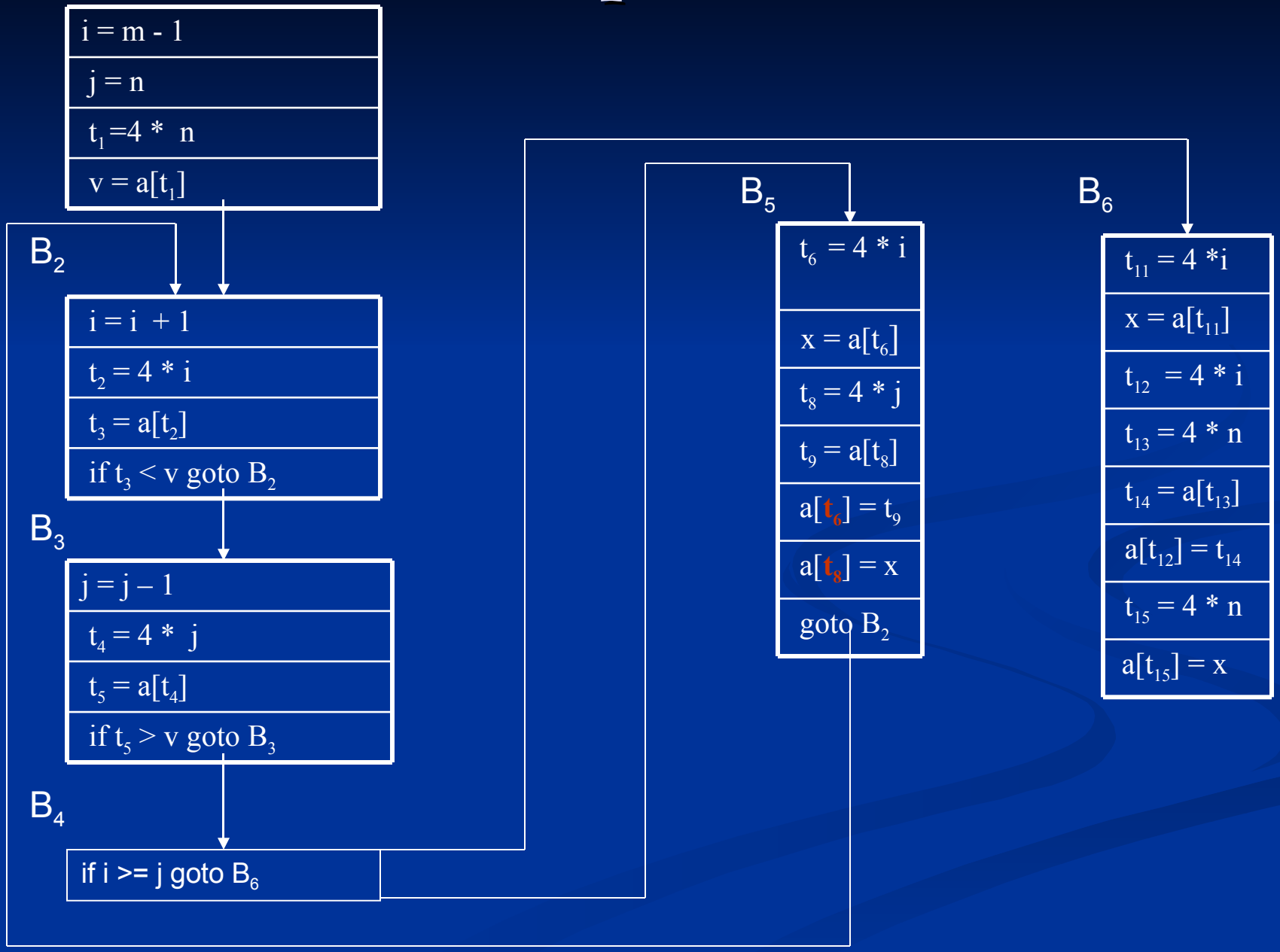
B_1 Common Subexpression Elimination



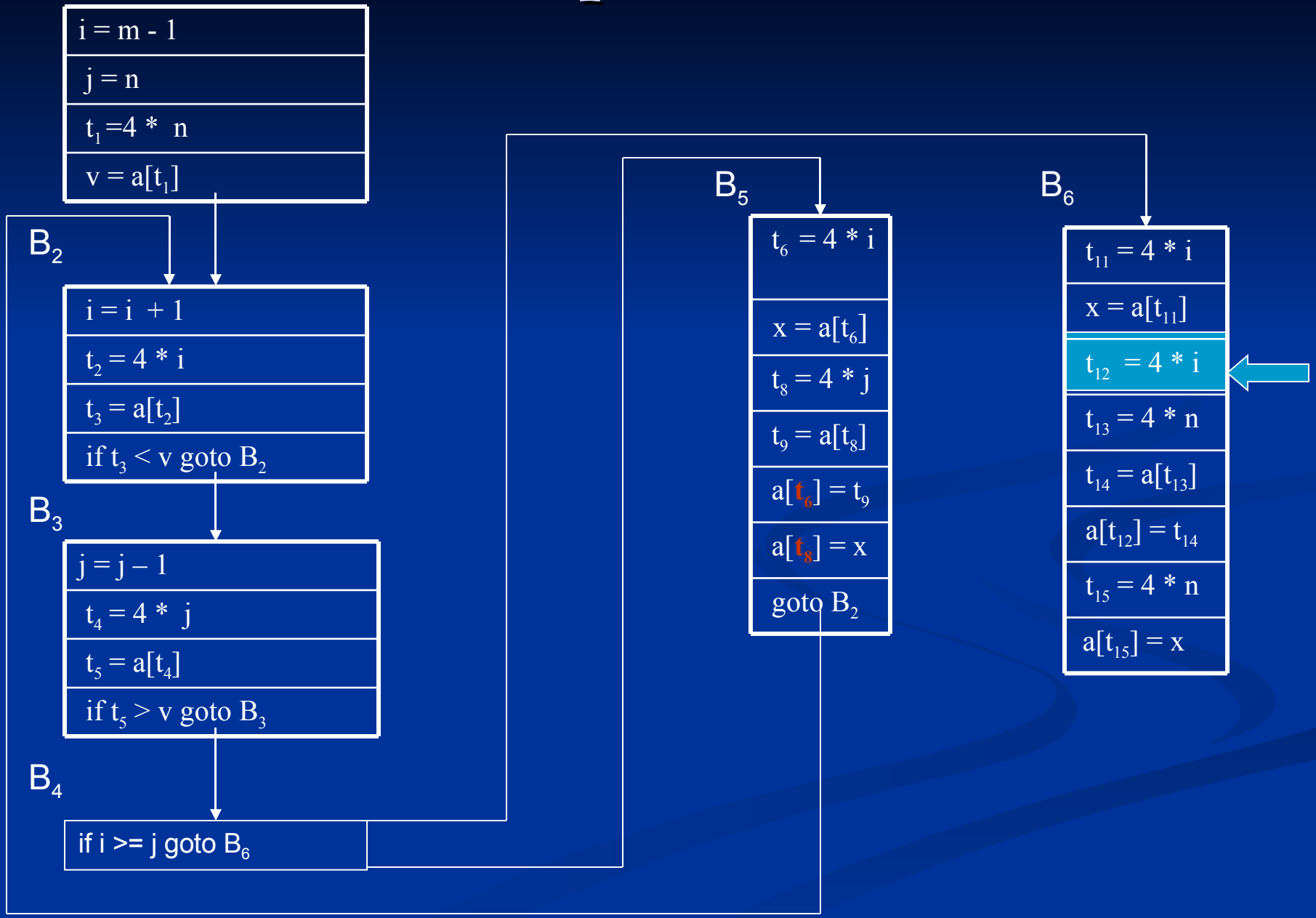
B_1 Common Subexpression Elimination



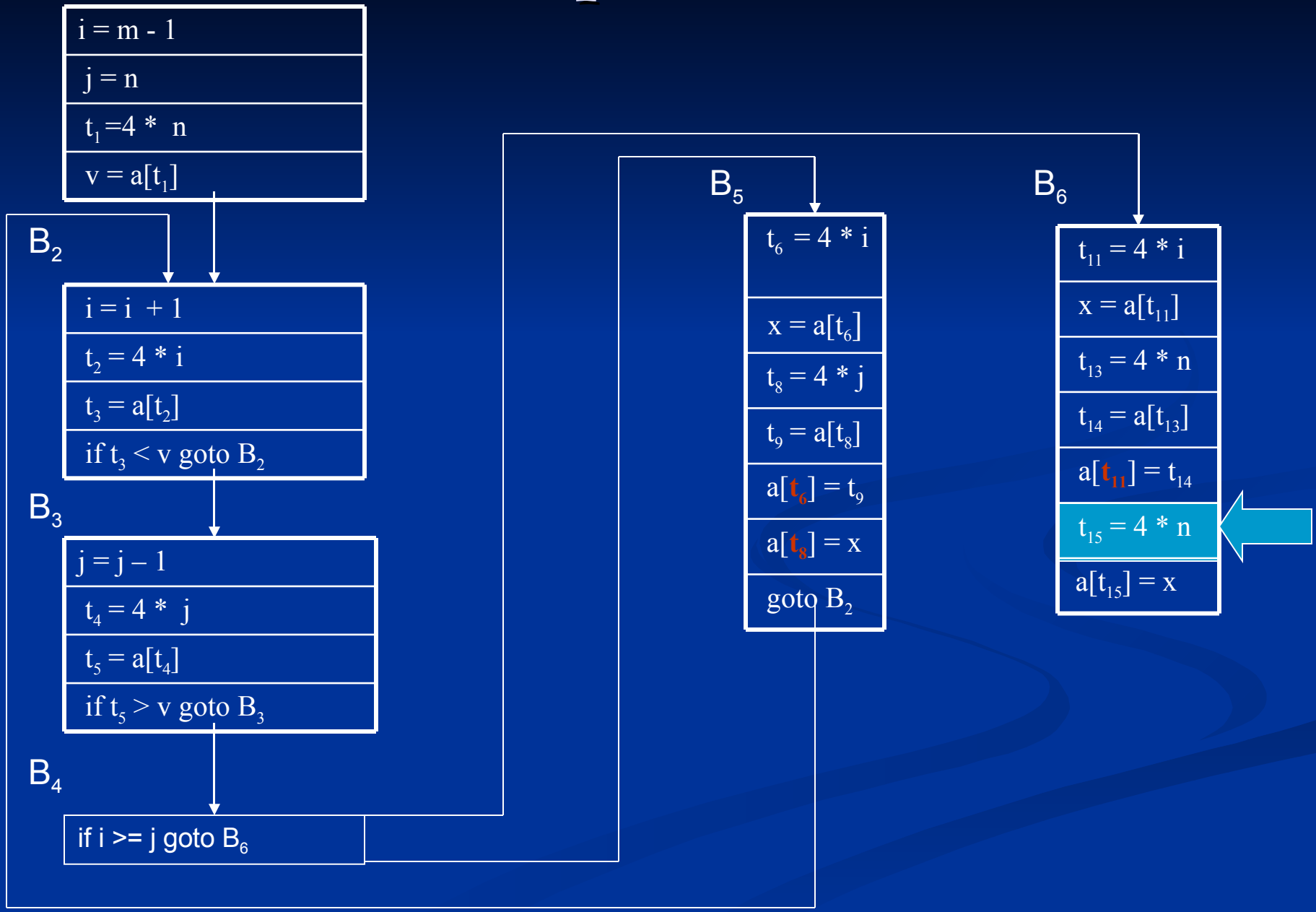
B_1 Common Subexpression Elimination



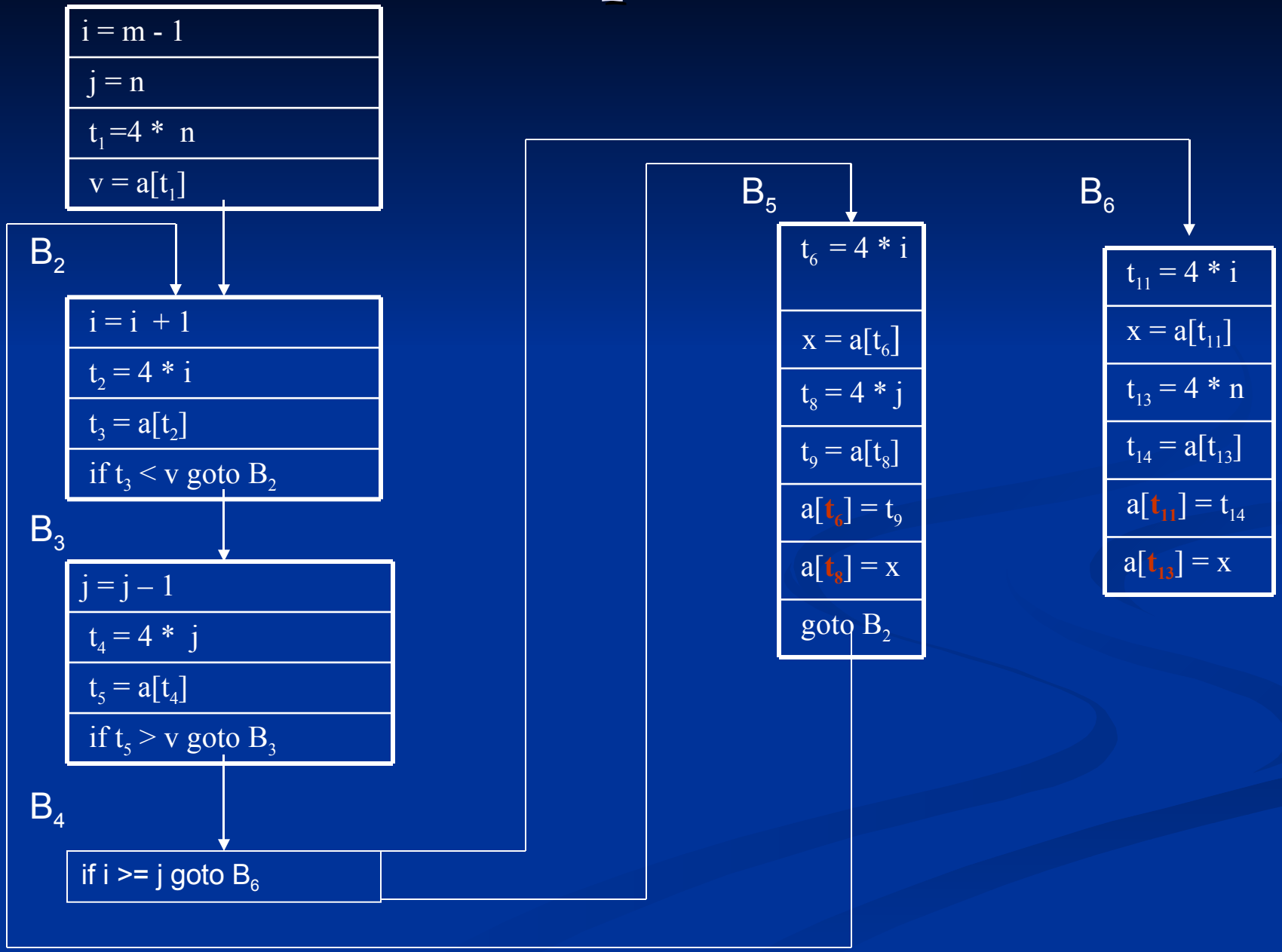
B_1 Common Subexpression Elimination



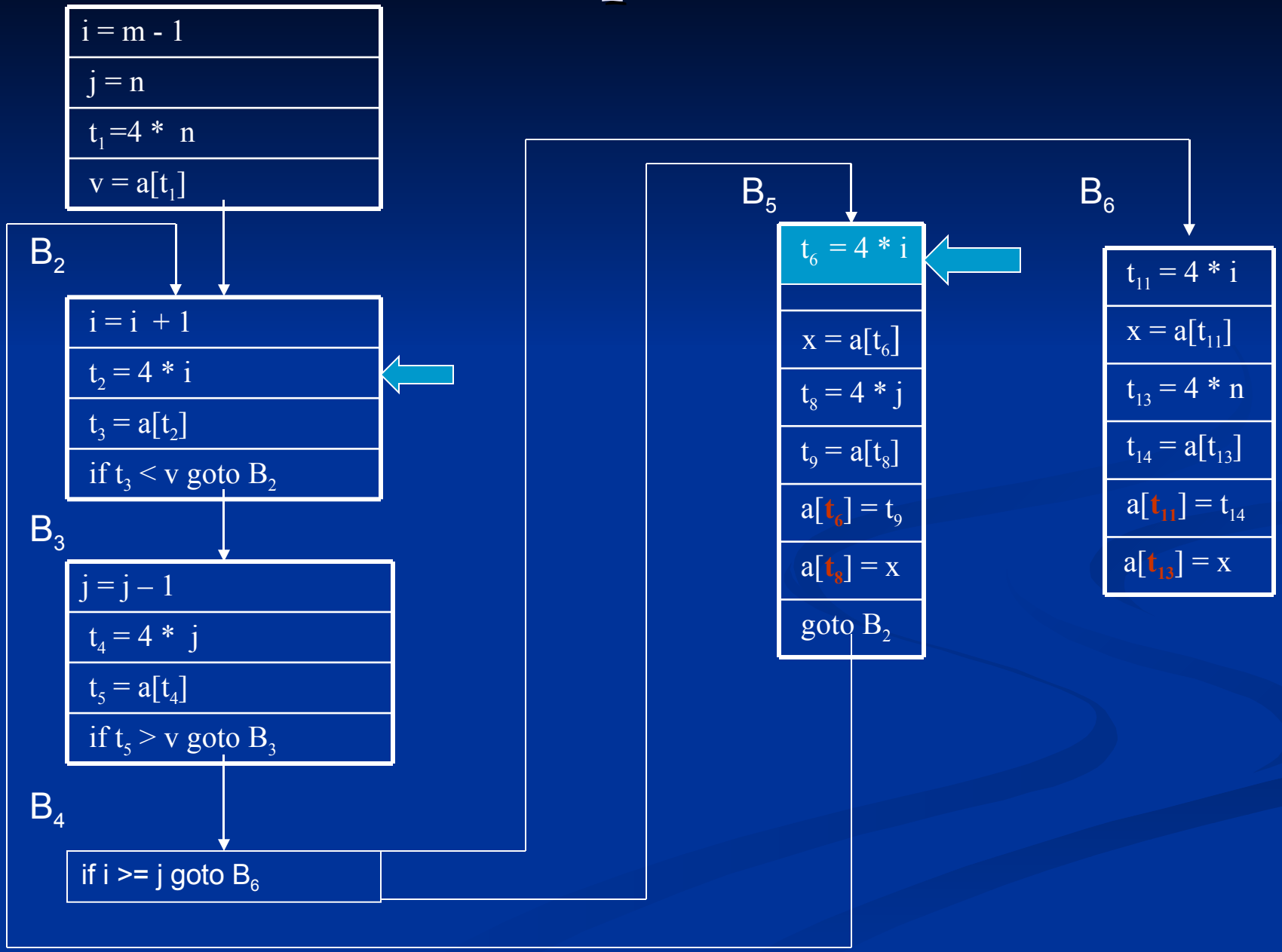
B_1 Common Subexpression Elimination



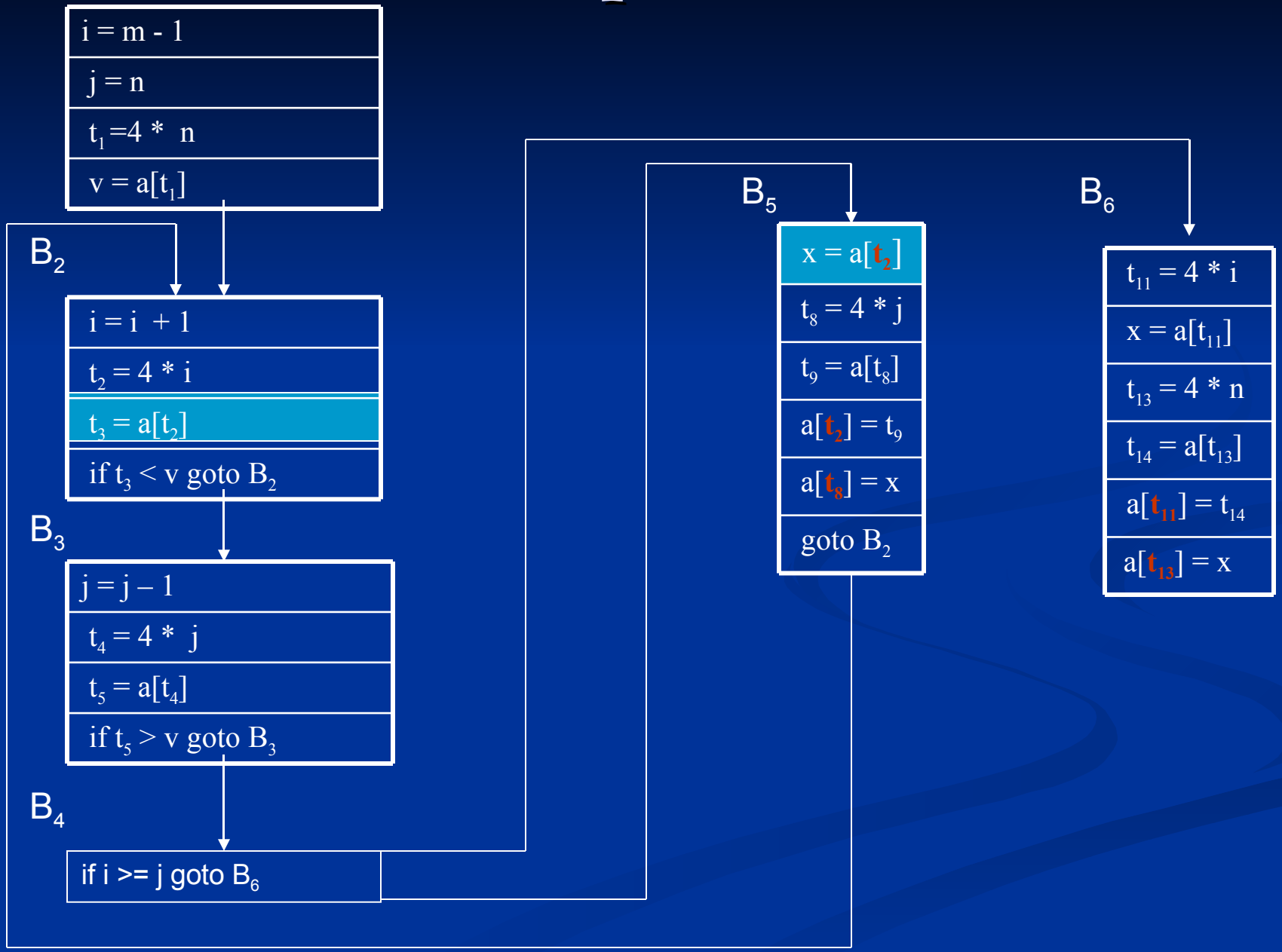
B_1 Common Subexpression Elimination



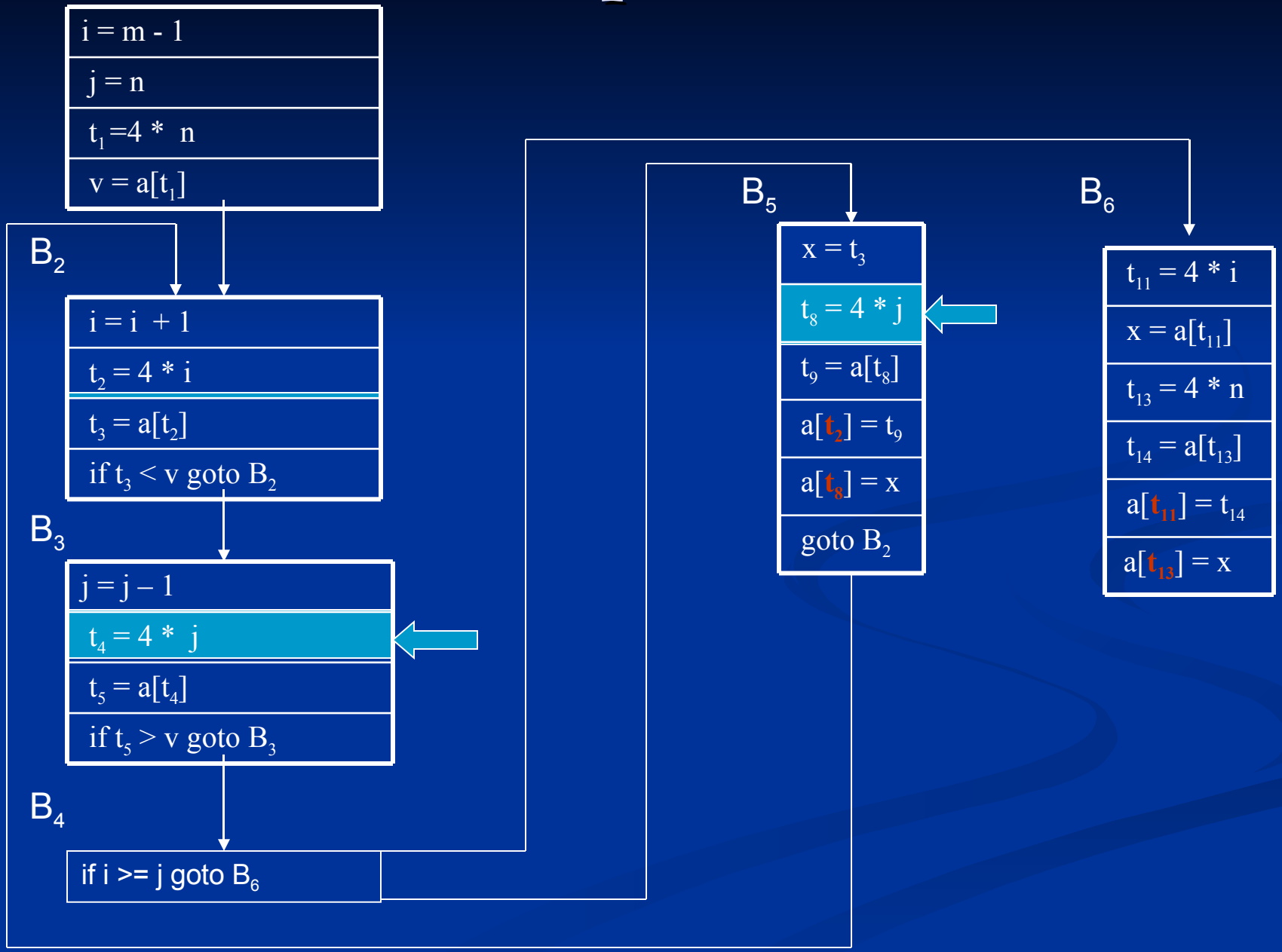
B_1 Common Subexpression Elimination



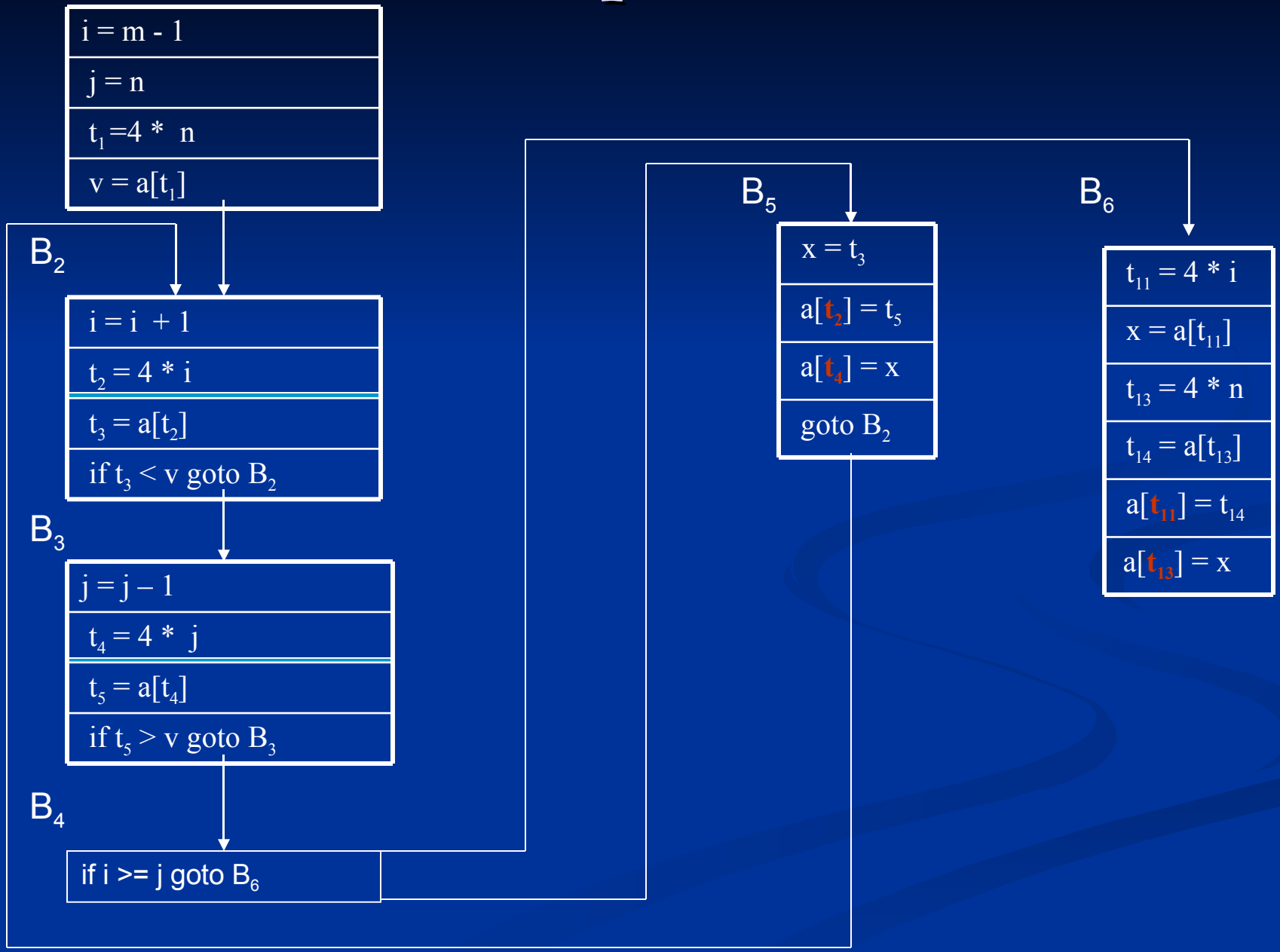
B_1 Common Subexpression Elimination



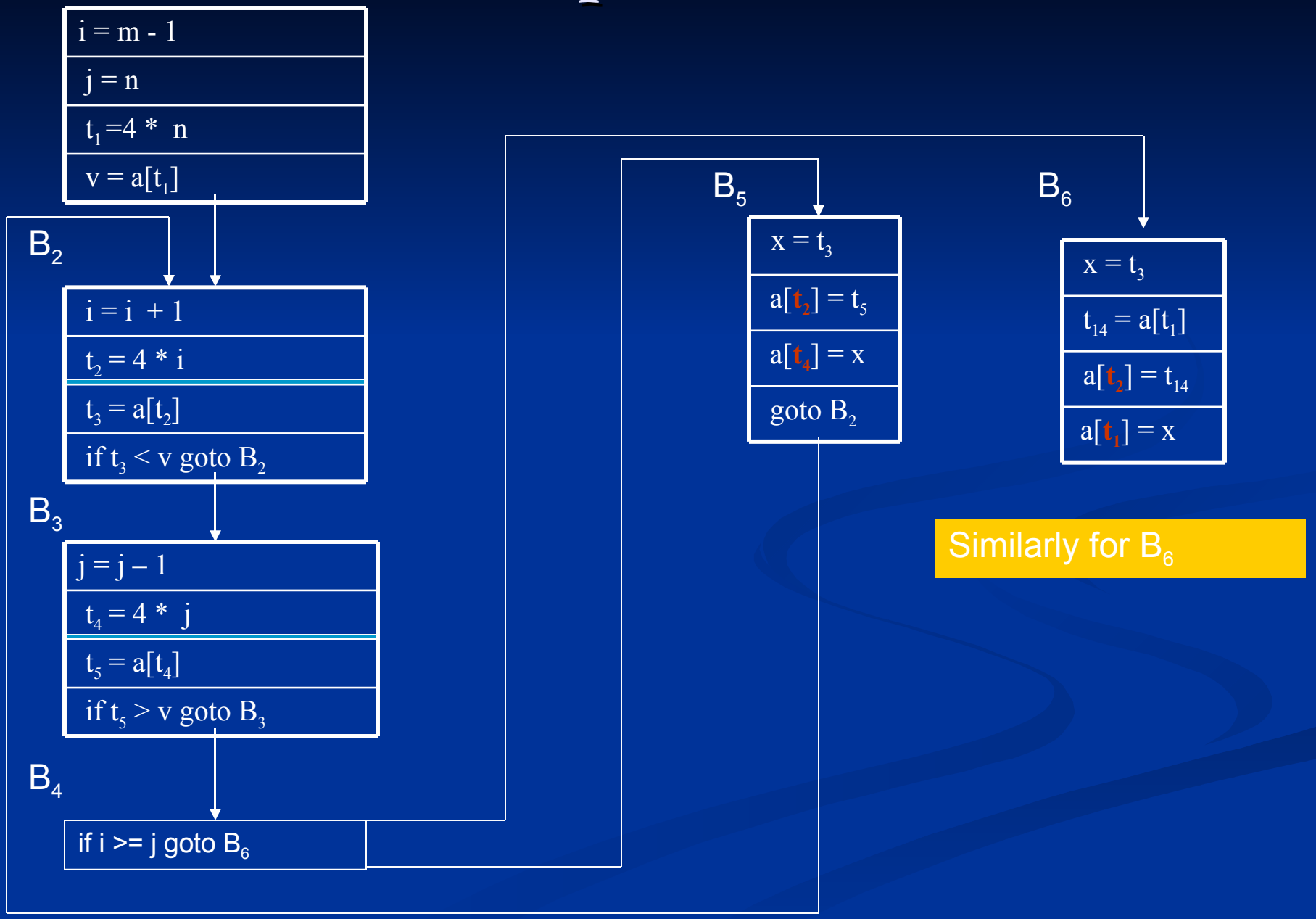
B_1 Common Subexpression Elimination



B_1 Common Subexpression Elimination

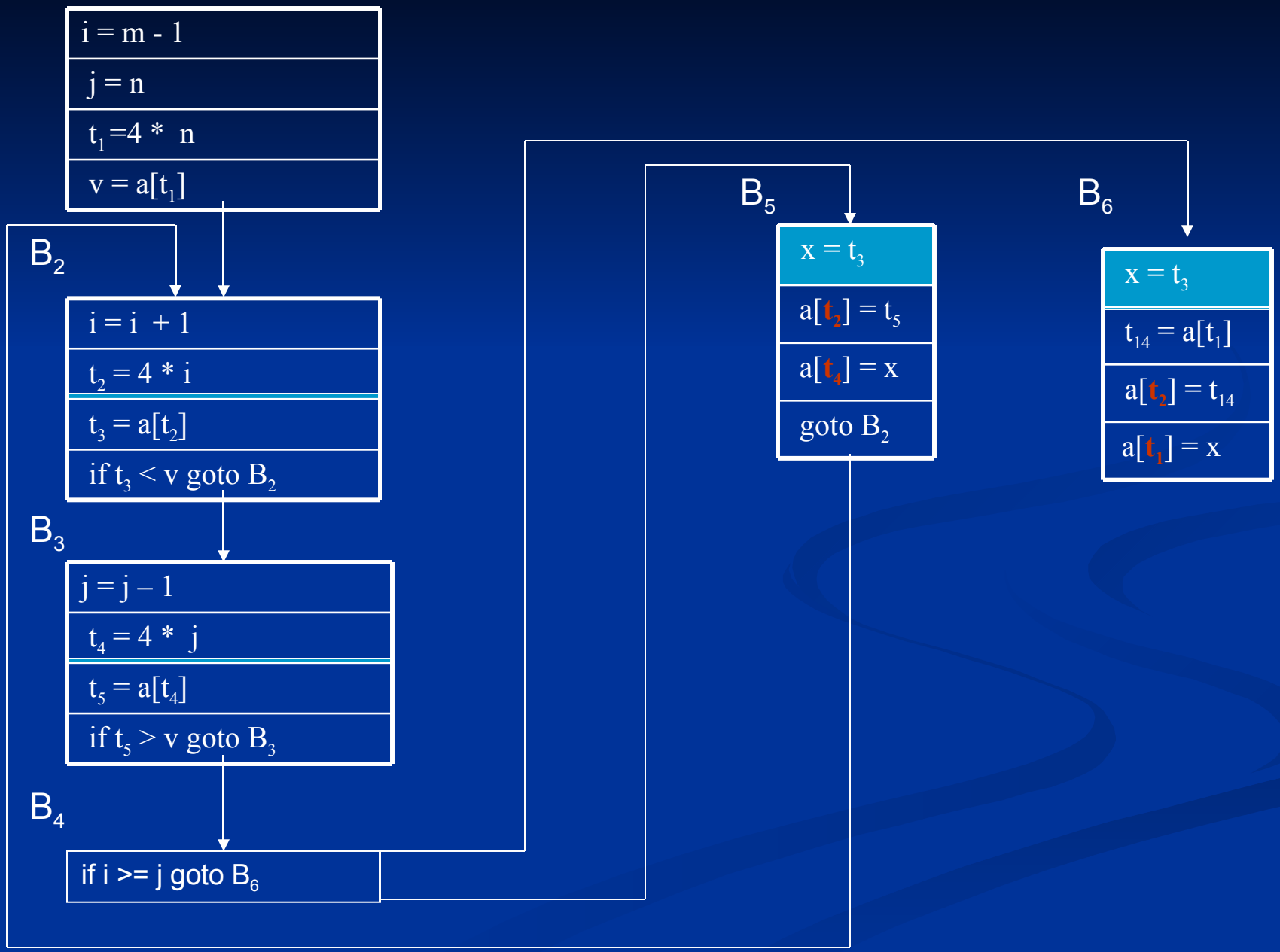


B_1 Common Subexpression Elimination

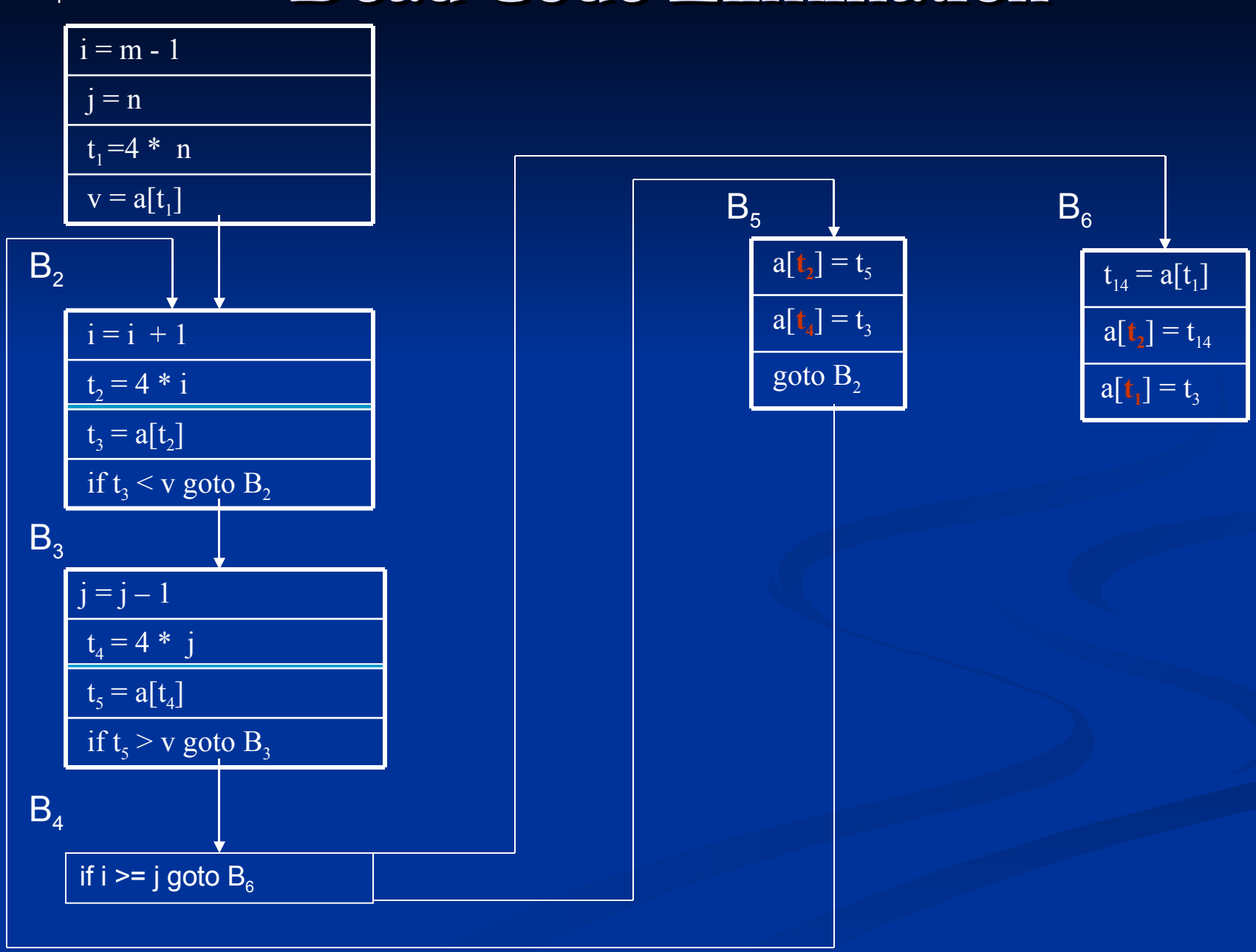


Similarly for B_6

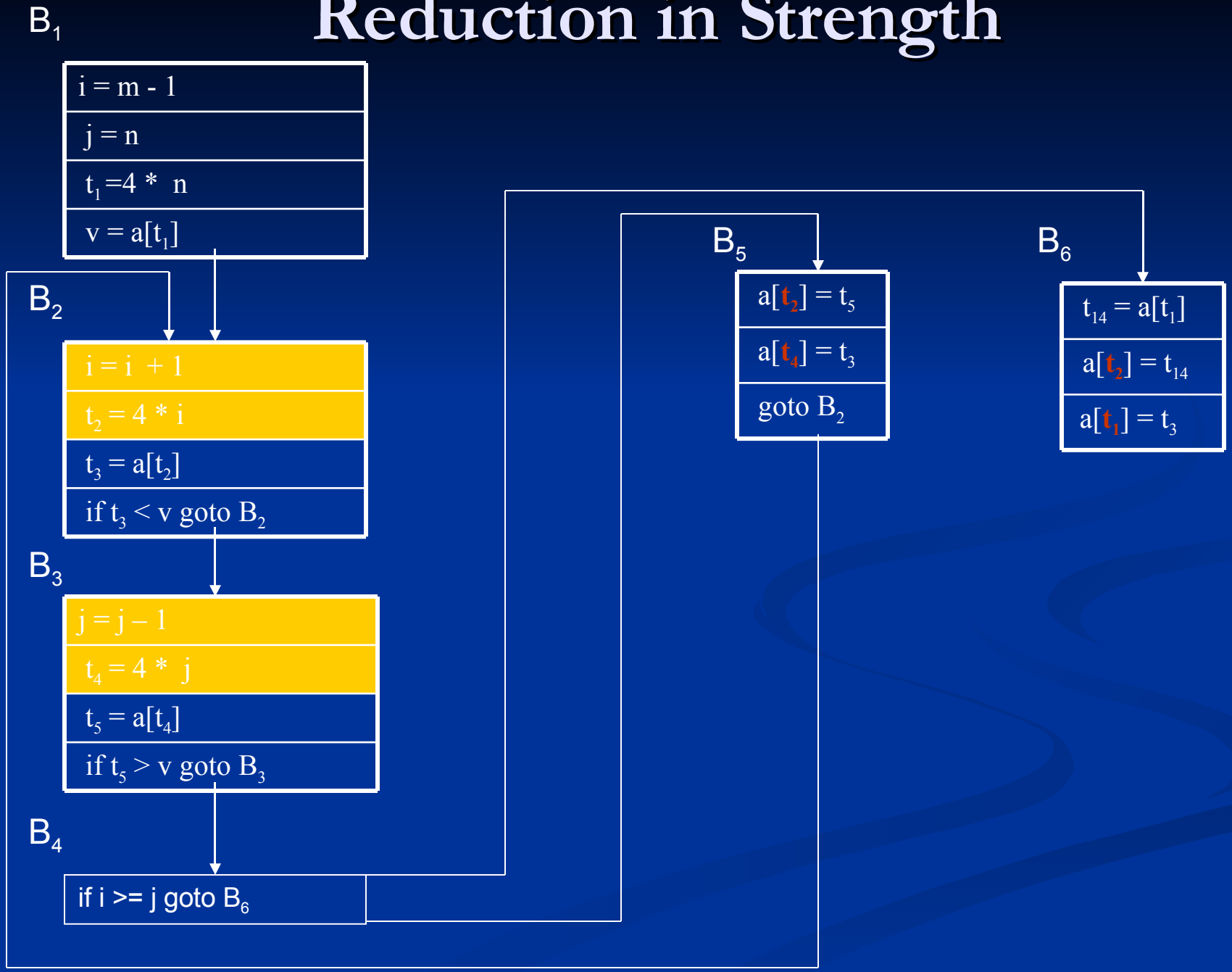
Dead Code Elimination



Dead Code Elimination



Reduction in Strength



Reduction in Strength