第九章　机器无关的优化

主要内容

- 引言
- 优化的来源
- 数据流分析
- 循环的识别、分析和优化

引言

- 代码优化或者代码改进
  - 在目标代码中消除不必要的指令
  - 把一个指令序列替换为一个完成相同功能的更快的指令序列
- 全局优化
- 有哪些可能的优化或改进机会？
- 具体的优化实现基于数据流分析技术
  - 用以收集程序相关信息的算法。

优化的来源

- 程序的冗余
  - 冗余是使用高级程序设计语言编程的副产品
  - 通过抽象的易于书写的方式编程和程序的高效之间存在矛盾。
  - 给编译器提供了优化的机会。

优化的示例

```
void quicksort(int m, int n)
    /* 递归地对 a[m]和a[n]之间的元素排序 */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* 片断由此开始 */
    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; /* 对换a[i]和a[j]*/
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* 对换a[i]和a[n] */
    /* 片断在此结束 */
    quicksort(m,j); quicksort(i+1,n);
}
```

图 9-1　快速排序算法的 C 代码

```
(1)    i = m-1              (16)    t7 = 4*i
(2)    j = n                (17)    t8 = 4*j
(3)    t1 = 4*n             (18)    t9 = a[t8]
(4)    v = a[t1]            (19)    a[t7] = t9
(5)    i = i+1              (20)    t10 = 4*j
(6)    t2 = 4*i             (21)    a[t10] = x
(7)    t3 = a[t2]           (22)    goto (5)
(8)    if t3<v goto (5)     (23)    t11 = 4*i
(9)    j = j-1              (24)    x  = a[t11]
(10)   t4 = 4*j             (25)    t12 = 4*i
(11)   t5 = a[t4]           (26)    t13 = 4*n
(12)   if t5>v goto (9)     (27)    t14 = a[t13]
(13)   if i>=j goto (23)    (28)    a[t12] = t14
(14)   t6 = 4*i             (29)    t15 = 4*n
(15)   x = a[t6]            (30)    a[t15] = x
```

图 9-2　图 9-1 中程序片断的三地址代码

优化的示例（续

```
(1)    i = m-1          (16)   t7 = 4*i
(2)    j = n            (17)   t8 = 4*j
(3)    t1 = 4*n         (18)   t9 = a[t8]
(4)    v = a[t1]        (19)   a[t7] = t9
(5)    i = i+1          (20)   t10 = 4*j
(6)    t2 = 4*i         (21)   a[t10] = x
(7)    t3 = a[t2]       (22)   goto (5)
(8)    if t3<v goto (5) (23)   t11 = 4*i
(9)    j = j-1          (24)   x  = a[t11]
(10)   t4 = 4*j         (25)   t12 = 4*i
(11)   t5 = a[t4]       (26)   t13 = 4*n
(12)   if t5>v goto (9) (27)   t14 = a[t13]
(13)   if i>=j goto (23)(28)   a[t12] = t14
(14)   t6 = 4*i         (29)   t15 = 4*n
(15)   x = a[t6]        (30)   a[t15] = x
```

图 9-2　图 9-1 中程序片断的三地址代码

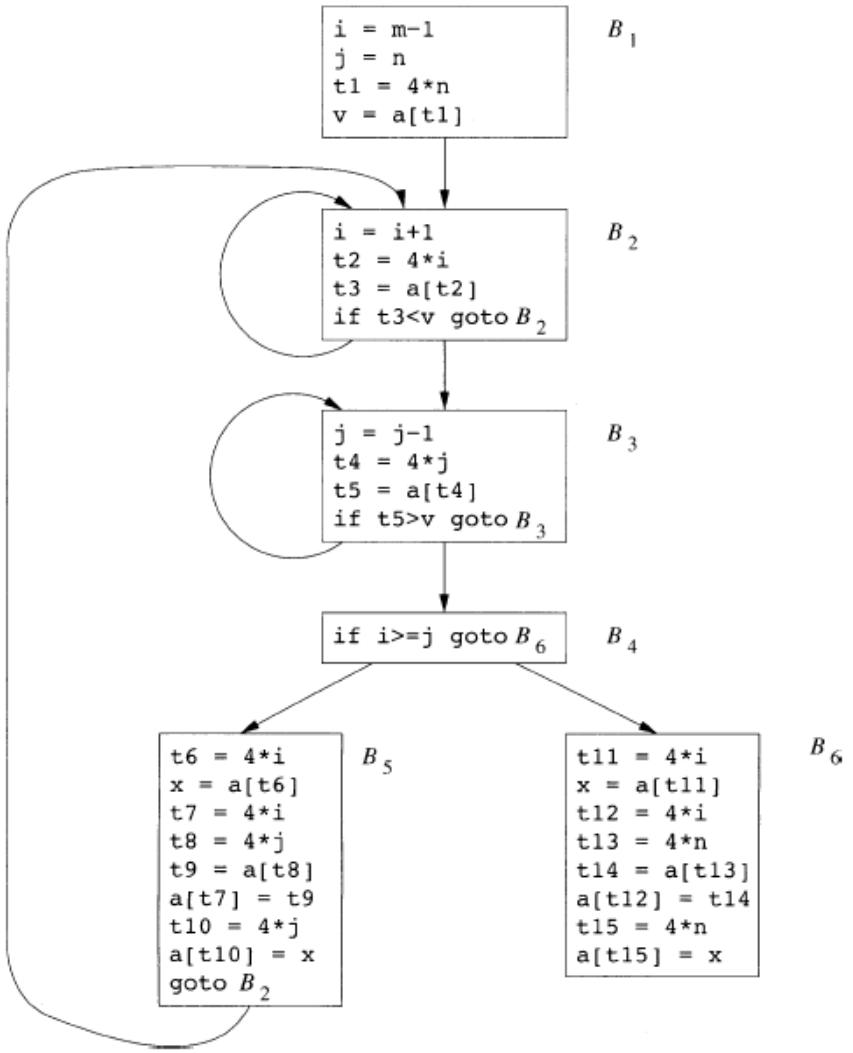


图 9-3　快速排序代码片断的流图

优1 例（续

- 全局
  - 表 其中的变量值都
  - 共
  - 如 赋予变量x，且x
  - 免
  - 例
  - 例 B5、B6中

```
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
```

B_5

a) 消除之前

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

B_5

b) 消除之后

图 9-4　局部公共子表达式消除

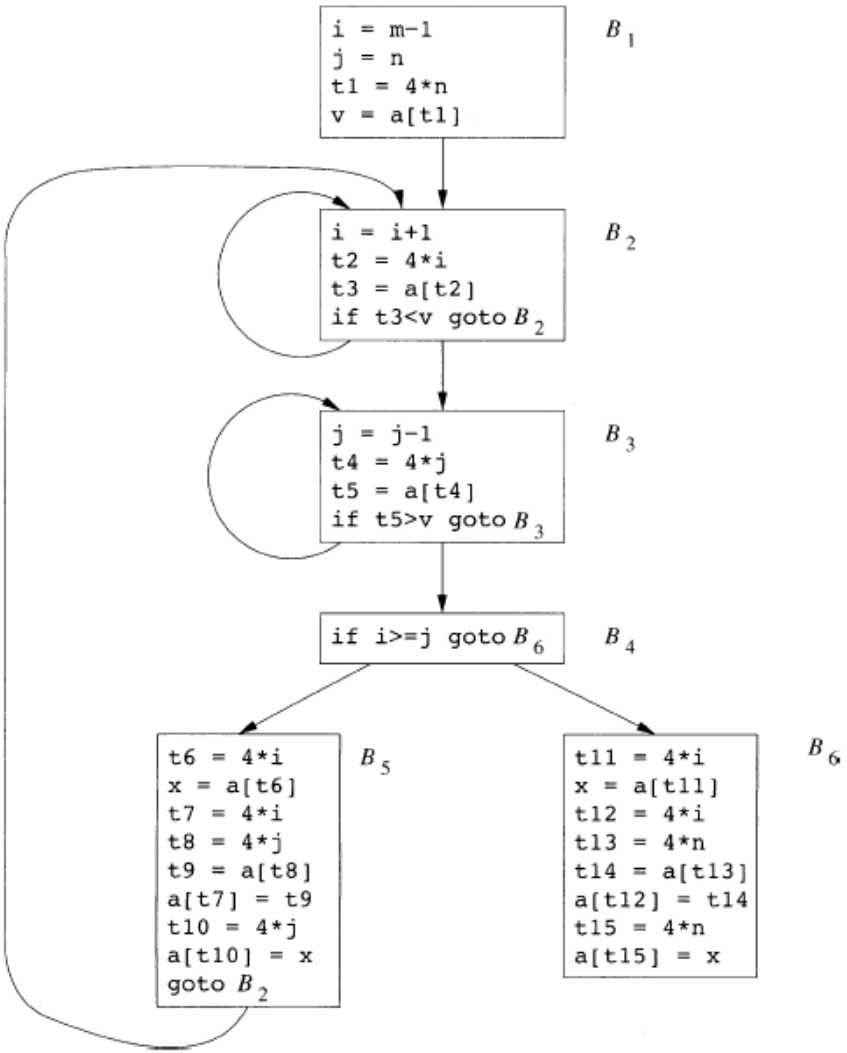
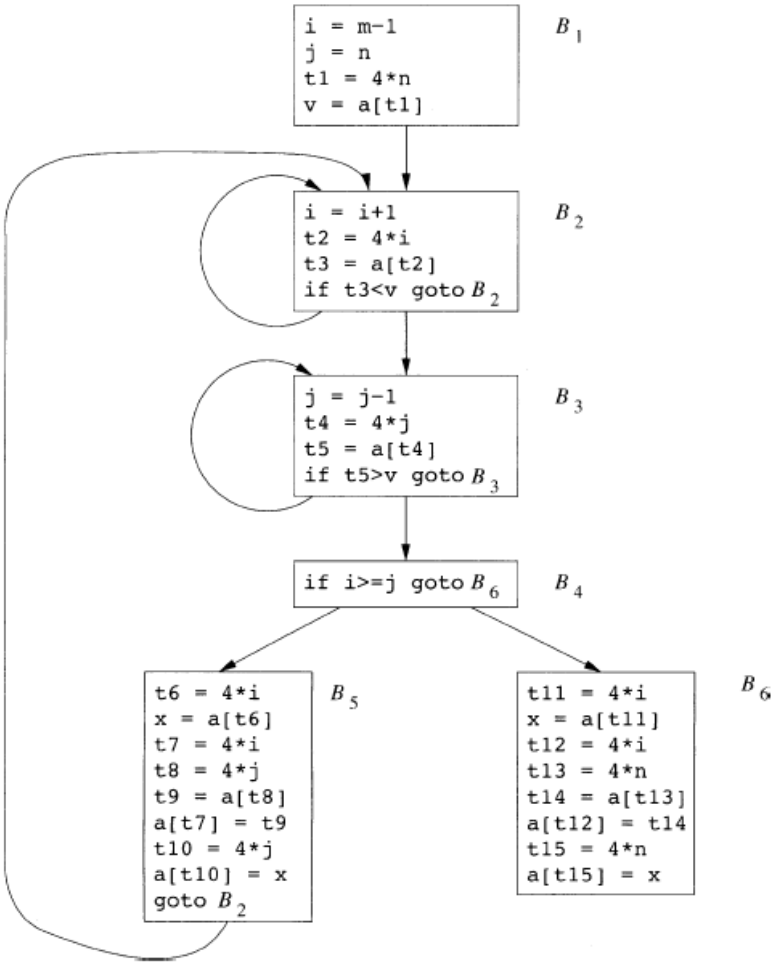
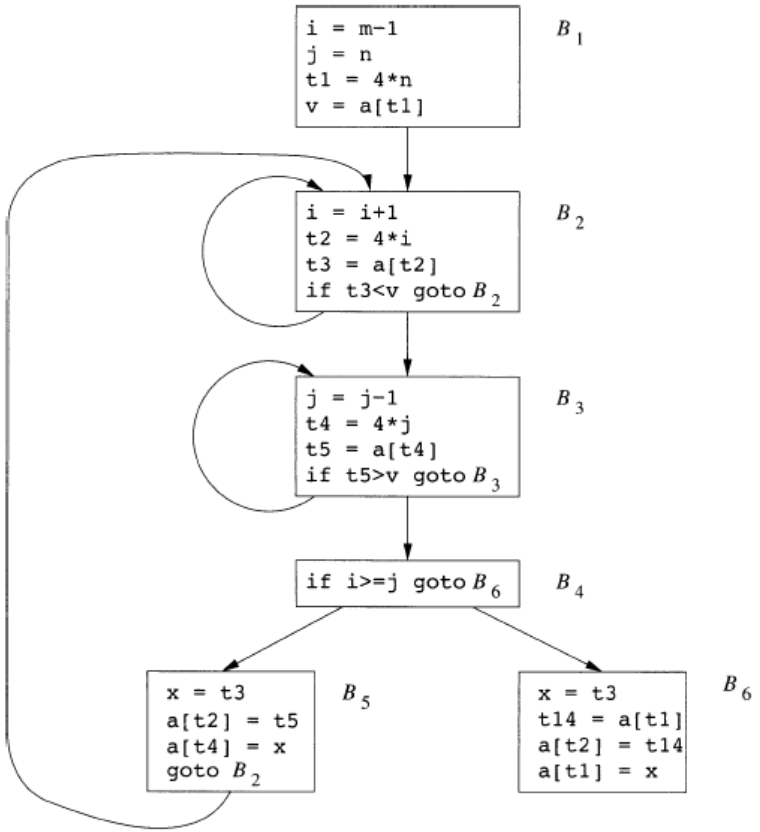


图 9-3　快速排序代码片断的流图

优化的示例（续）

- 消除公共子表达式后的流图





优化 - 复制传播

- 复制语句：形如u=v这样的复制表达式
- 复制语句的产生，有时是由其公共子表达式消除优化引入的。
- 复制传播转换的基本思想
  是在复制语句u=v之后尽可能的用v来替代u。

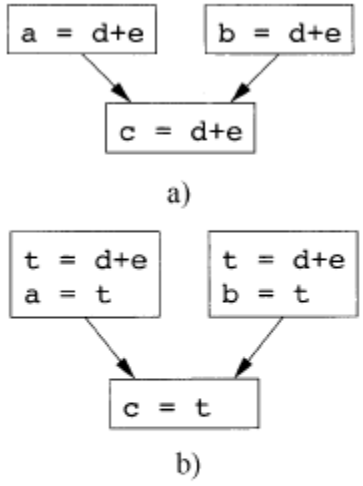


图 9-6　在公共子表达式消除过程中引入的复制语句

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

图 9-7　进行复制传播转换后的基本块 B_5

优化 - 死代码的消除

- 活跃：一个变量在一个程序点上的值可能会在以后被使用，那么这个变量在该点活跃。
- 死：不活跃
- 死代码：其计算结果用于不会被使用的语句。
- *常量折叠*会产生一些死代码。
- *复制传播*也会把一些复制语句变成死代码。

优化 - 代码移动

- 尽可能减少内部循环的指令可能，即使可能增加循环外的指令个数也是值得的。
- 循环不变表达式：无论循环多少次，表达式的值都不会变化的表达式。
- 代码移动：在进入循环前，对循环不变表达式进行求值。

```
while (i <= limit-2) /* 不改变limit值的语句*/
```

```
t = limit-2
while (i <= t)  /* 不改变limit或t值的语句 */
```

优化 - 归纳变量和强度削减

- 归纳变量：对于一个变量x，如果存在一个正的或负的常数c使得每次x被赋值时它的值总是增加c，那么x被称为归纳变量。
  - 例如图9.5 B2中的i和t2。
- 强度削减：把一个高代价的运算（比如乘法）替换为一个代价较低的运算（比如加法）的转换称为强度削减。
- 如果有一组归纳变量的值的变化保持步调一致，常常可以将这组变量删除只剩一个。

优化 - 归纳变量和强度削减(例)

- 例9.6

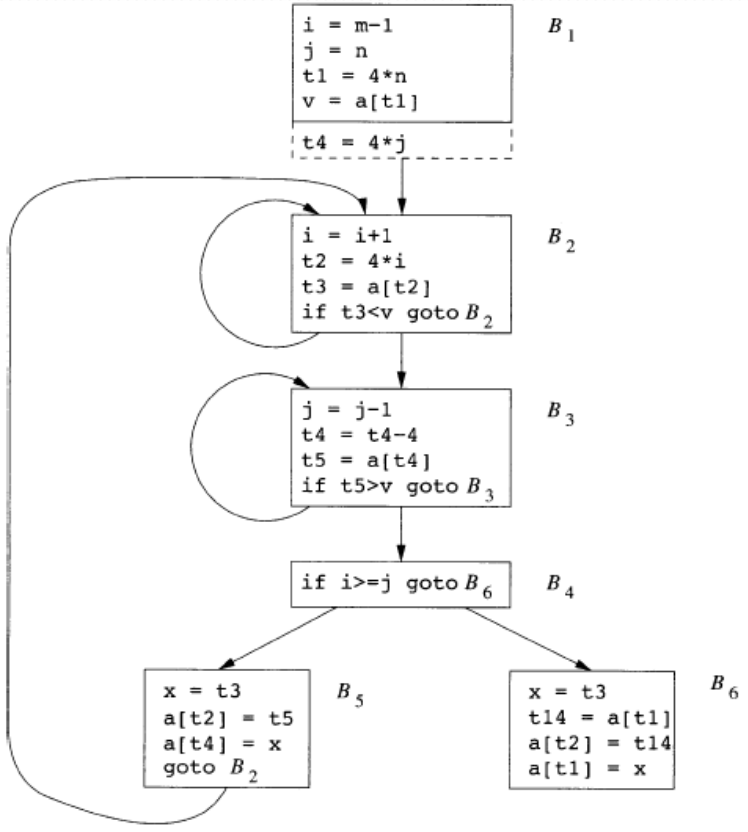


图9-8　对基本块 B_3 中的 $4*j$ 应用强度消减优化

优化 - 归纳变量和强度削减(例)

- 删除具有同步调变化的归纳变量

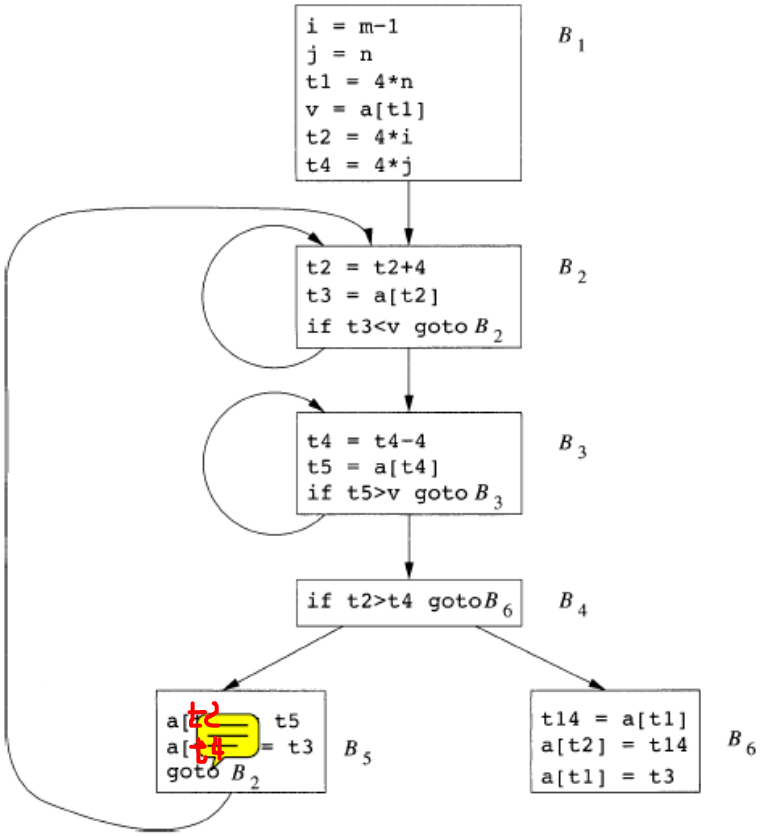


图 9-9　归纳变量消除之后的流图

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]

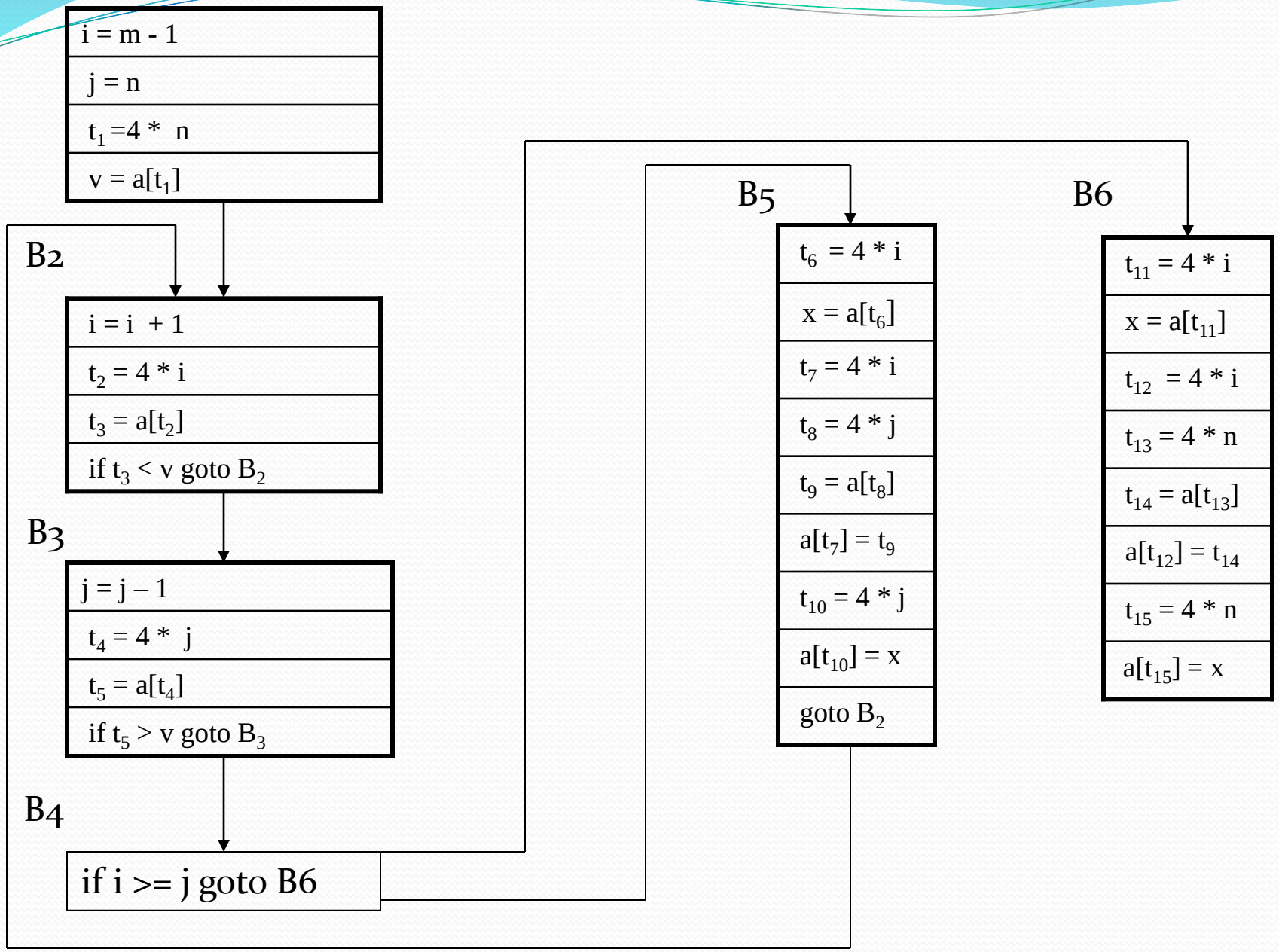
16	t_7 = 4 * I
17	t_8 = 4 * j
18	t_9 = a[t_8]
19	a[t_7] = t_9
20	t_{10} = 4 * j
21	a[t_{10}] = x
22	goto (5)
23	t_{11} = 4 * I
24	x = a[t_{11}]
25	t_{12} = 4 * i
26	t_{13} = 4 * n
27	t_{14} = a[t_{13}]
28	a[t_{12}] = t_{14}
29	t_{15} = 4 * n
30	a[t_{15}] = x

Find The Basic Block

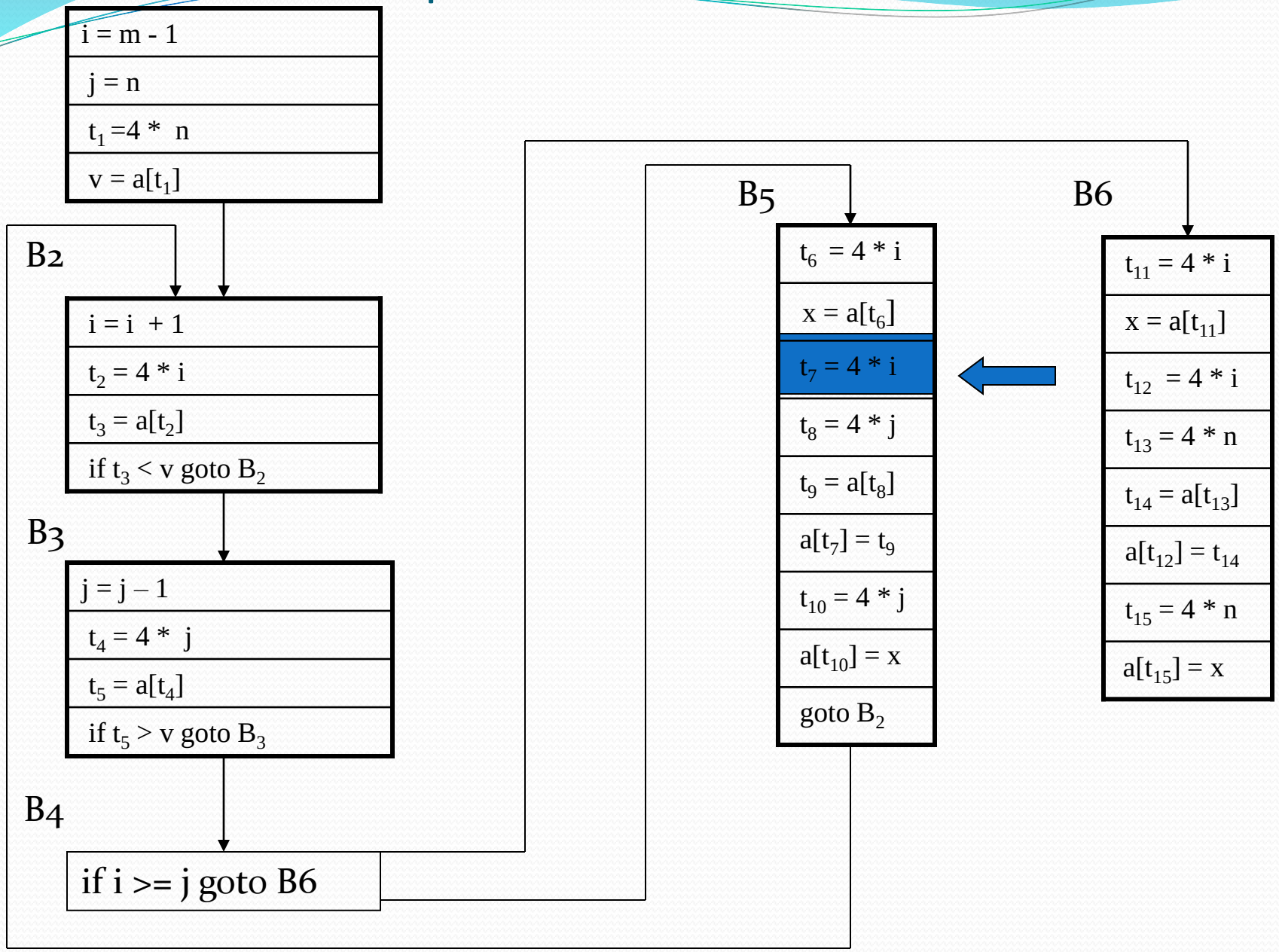
1	i = m - 1
2	j = n
3	t_1 =4 * n
4	v = a[t_1]
5	i = i + 1
6	t_2 = 4 * i
7	t_3 = a[t_2]
8	if t_3 < v goto (5)
9	j = j – 1
10	t_4 = 4 * j
11	t_5 = a[t_4]
12	if t_5 > v goto (9)
13	if i >= j goto (23)
14	t_6 = 4 * i
15	x = a[t_6]

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

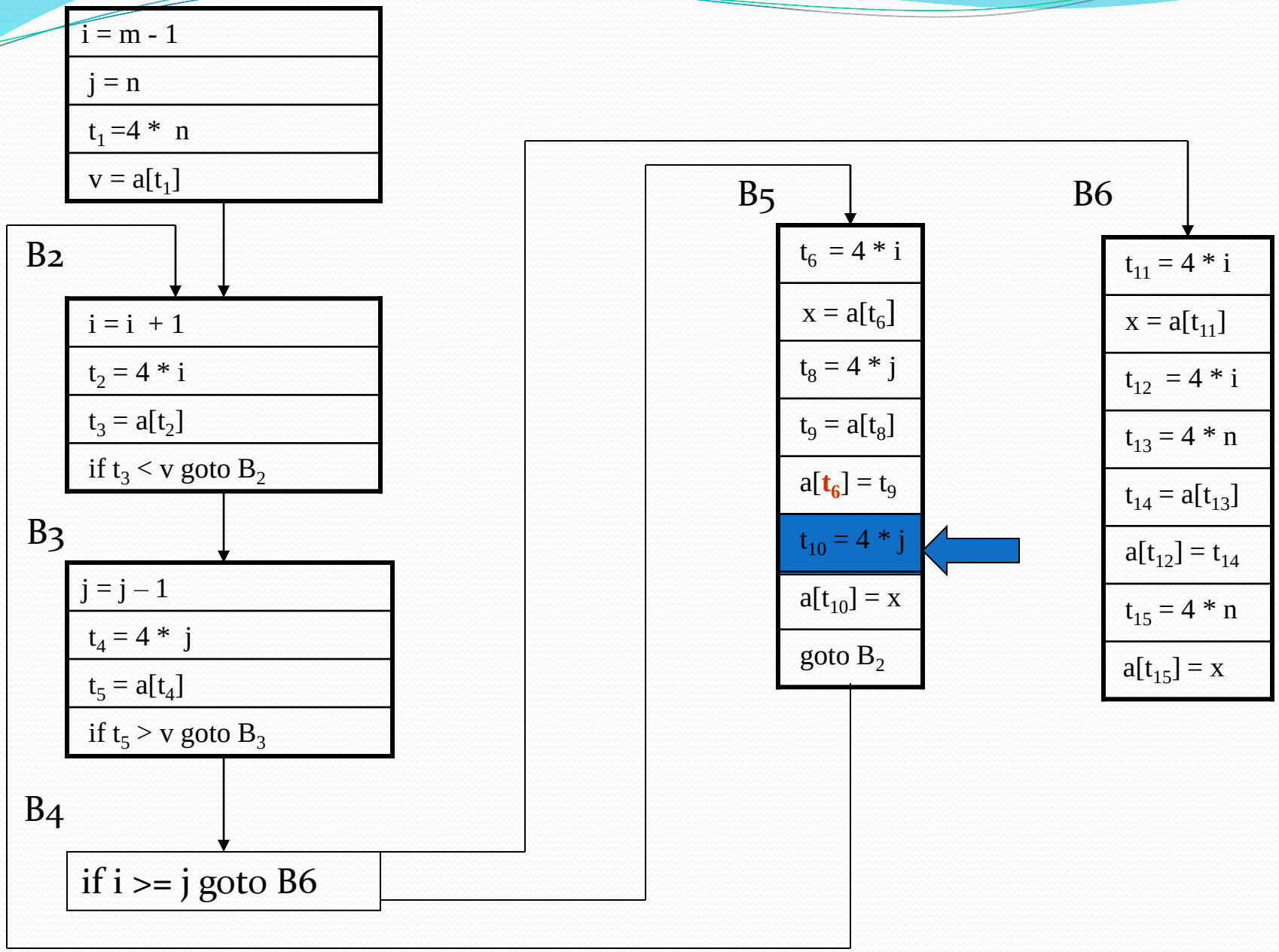
Flow Graph


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

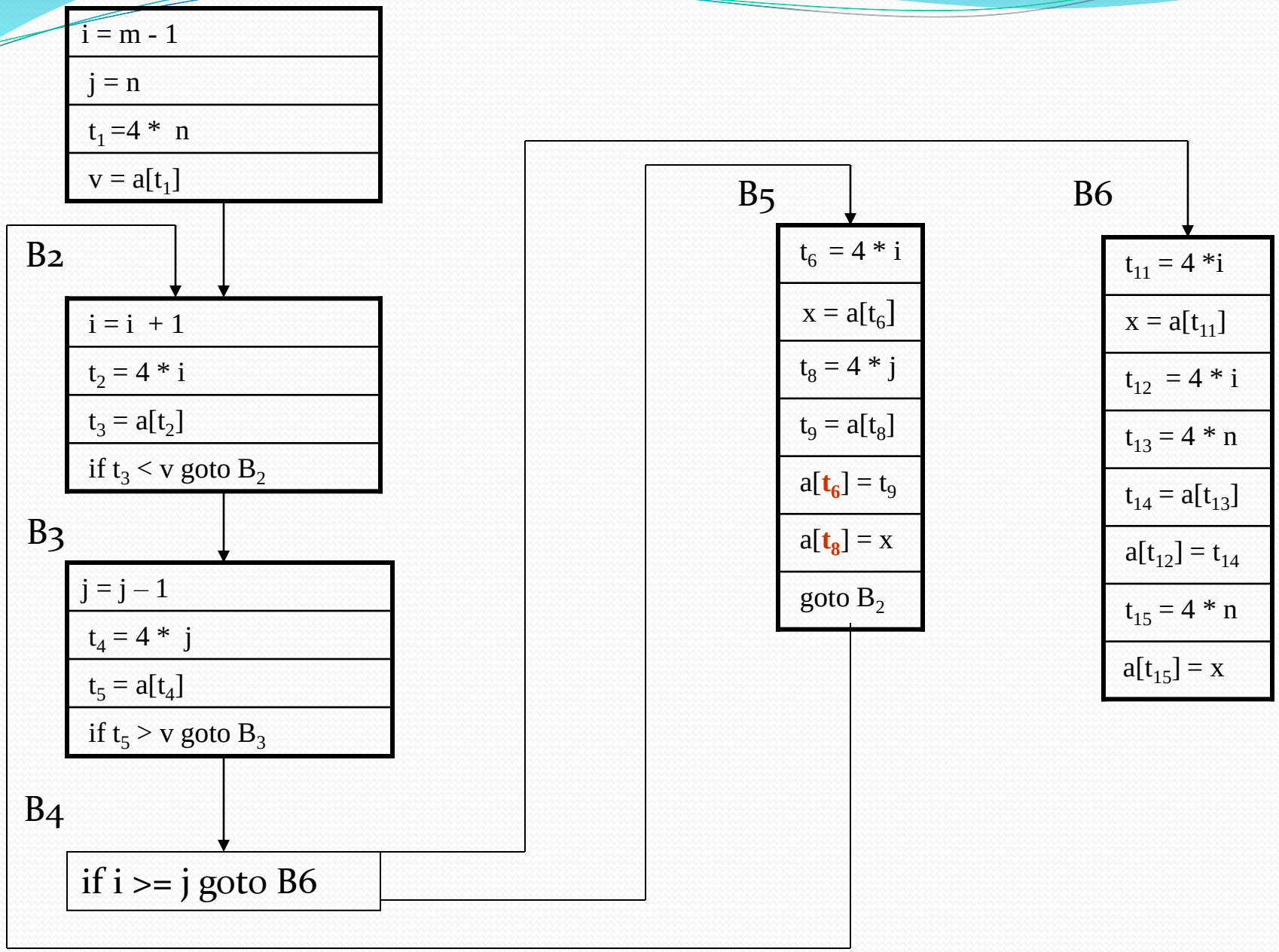
Common Subexpression Elimination



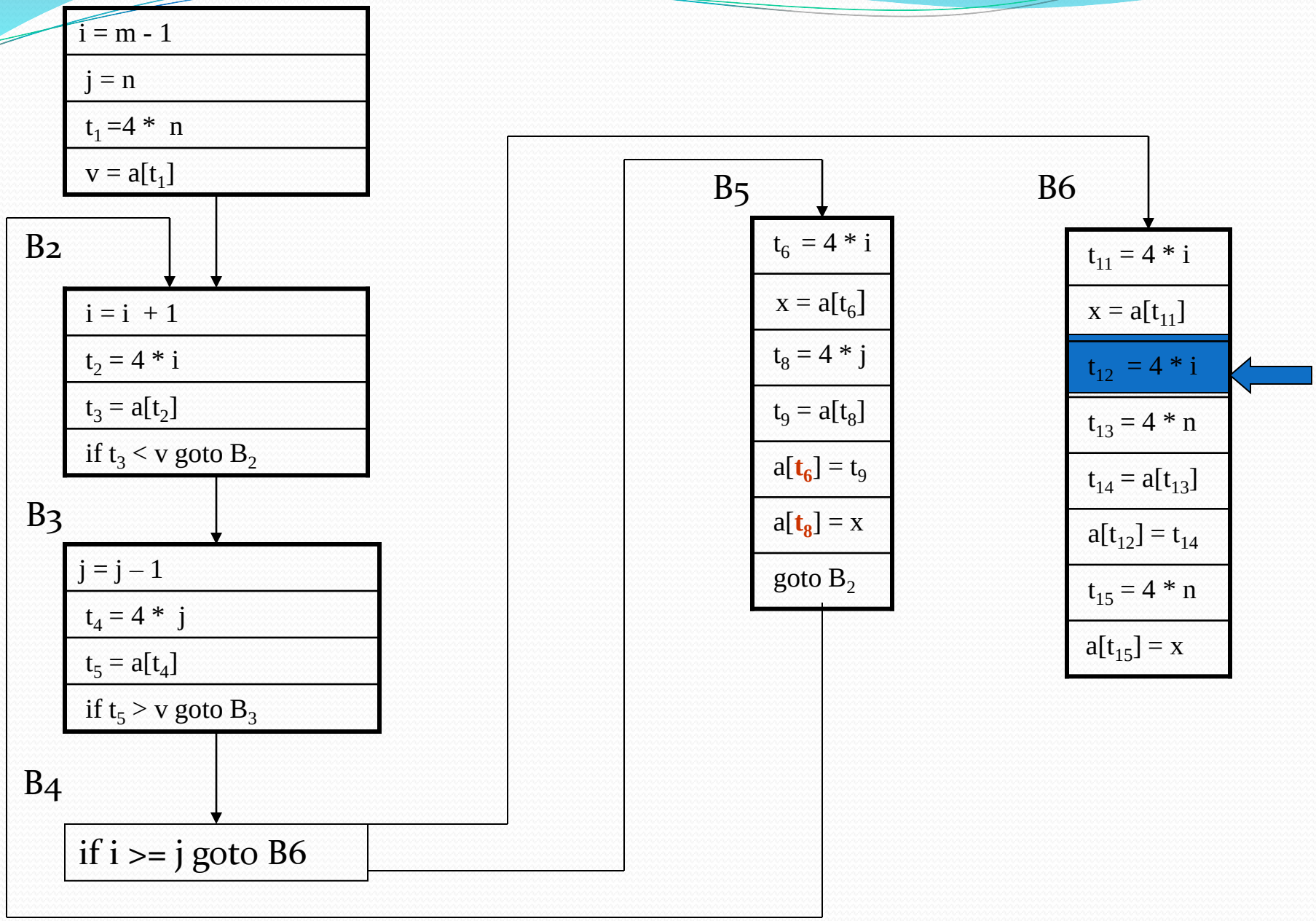
B1 Common Subexpression Elimination



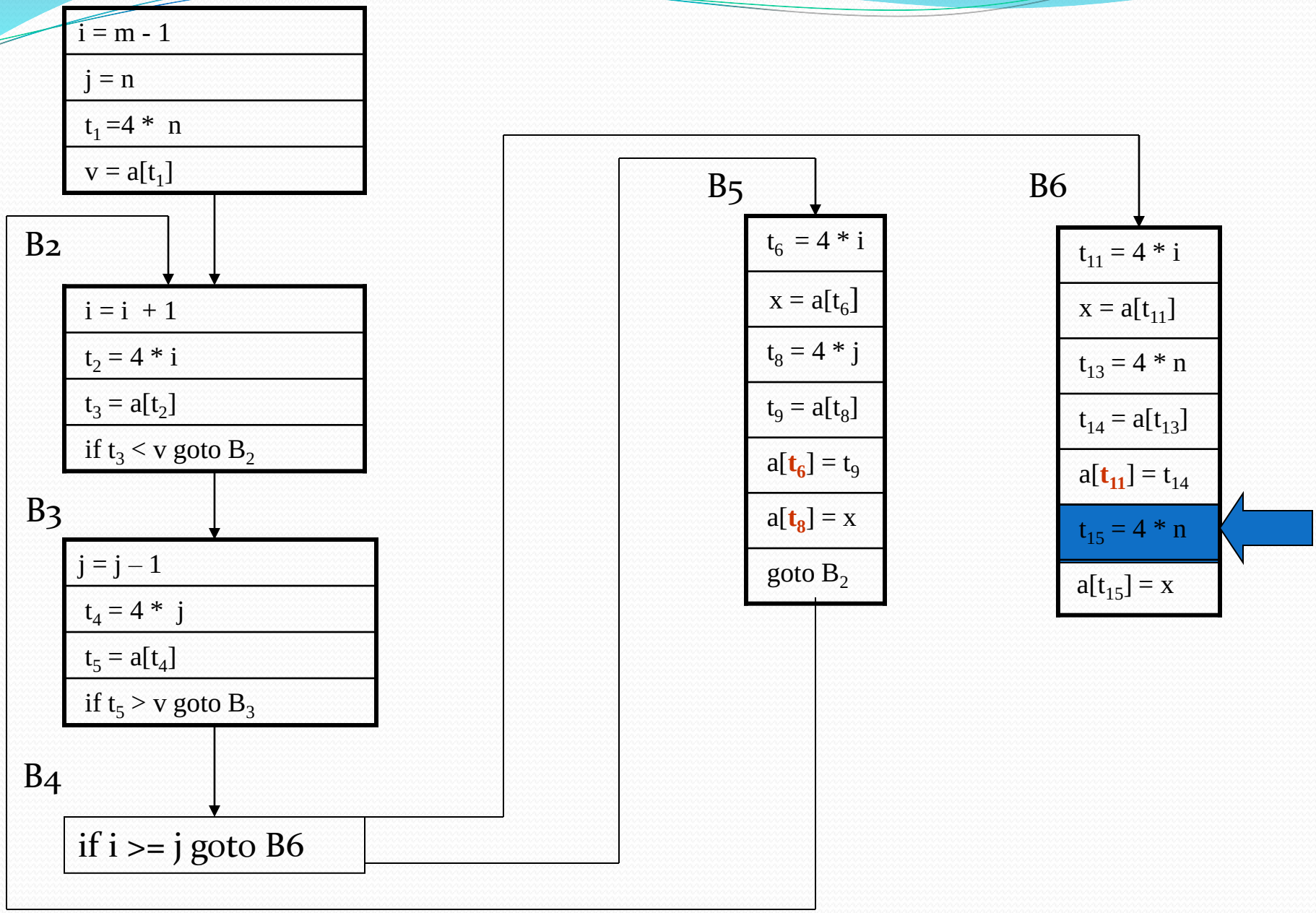
B1 Elimination



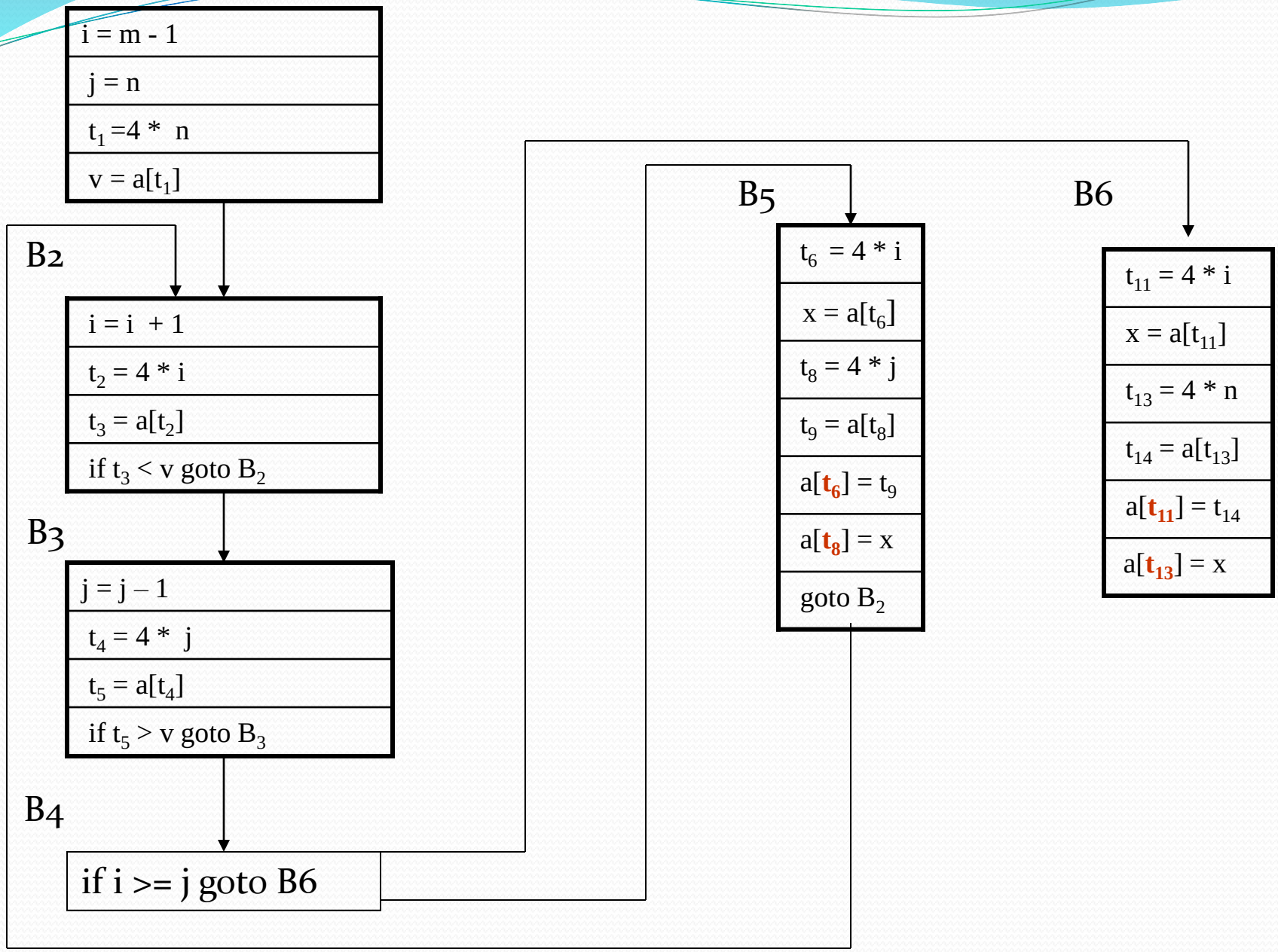
B1 Elimination



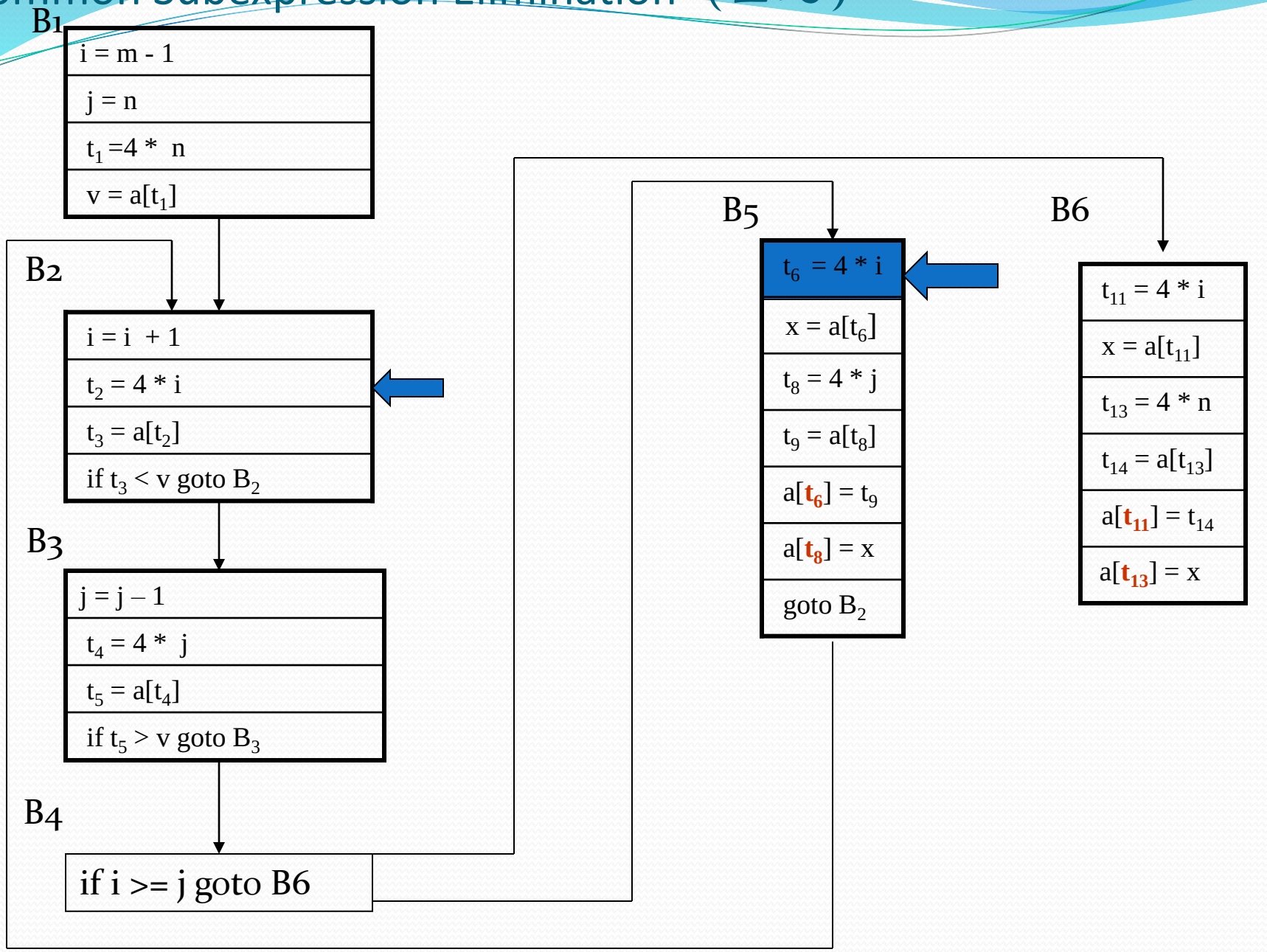
B1 Elimination



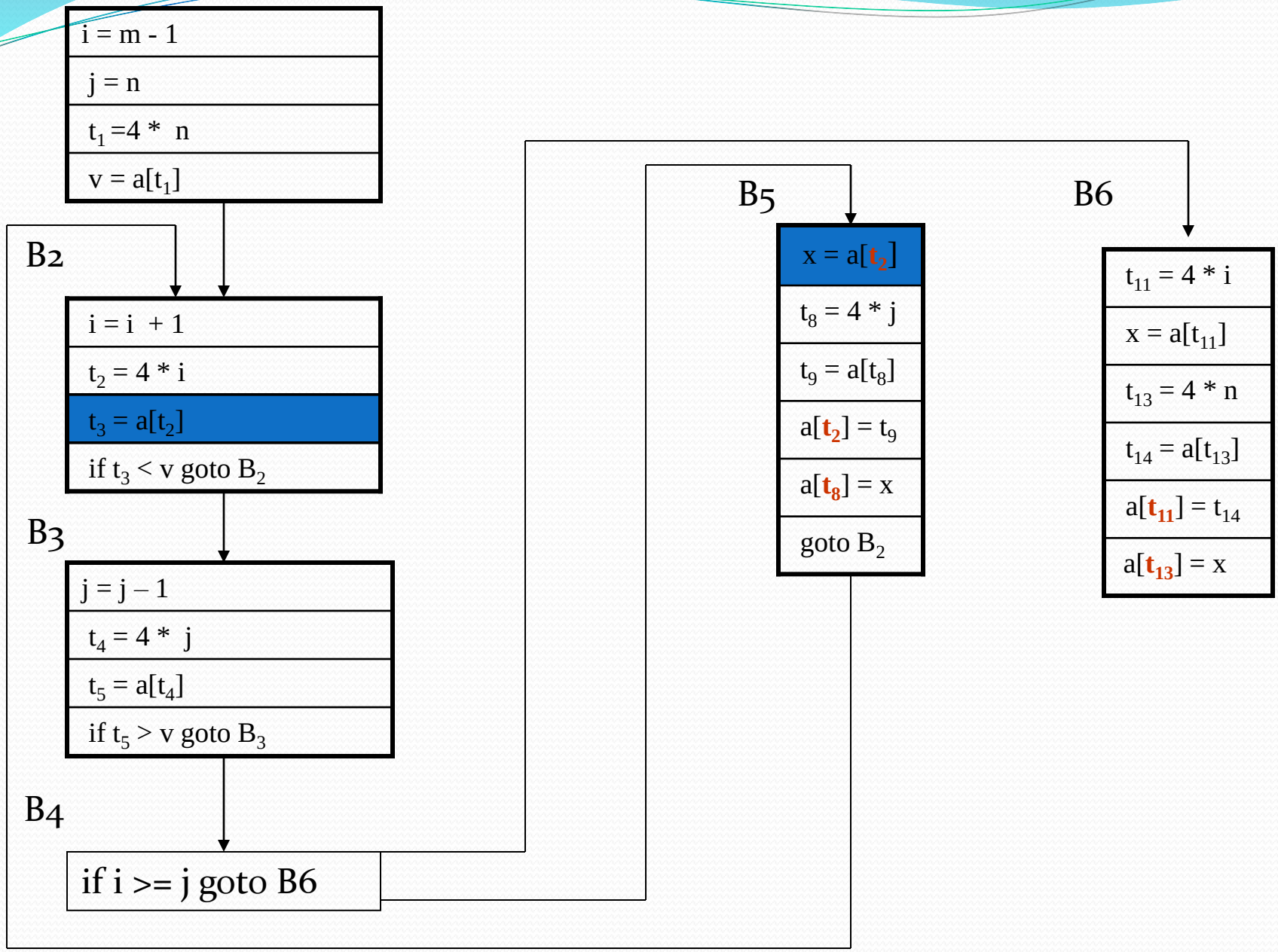
B1 Elimination



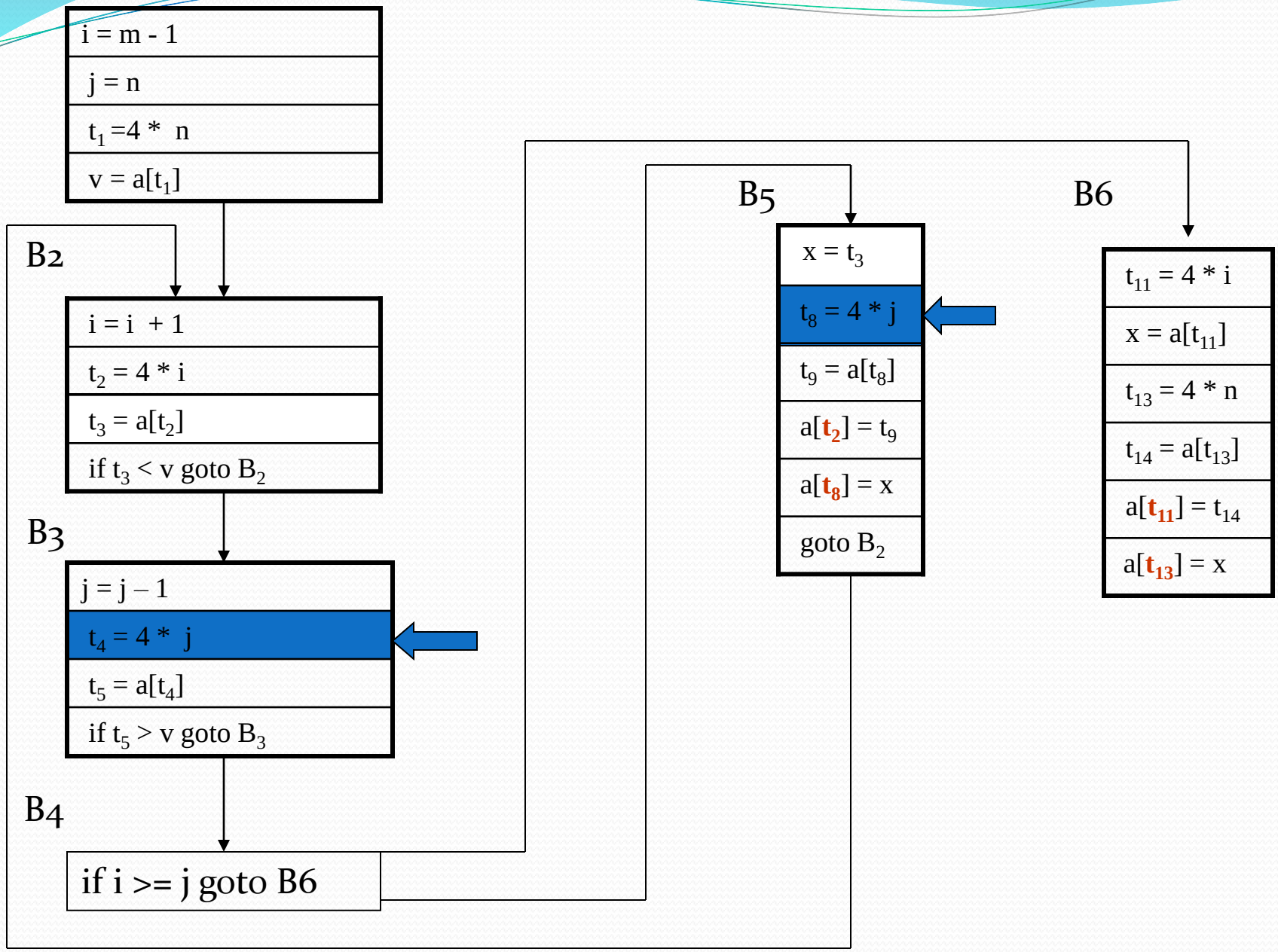
Common Subexpression Elimination（全局）



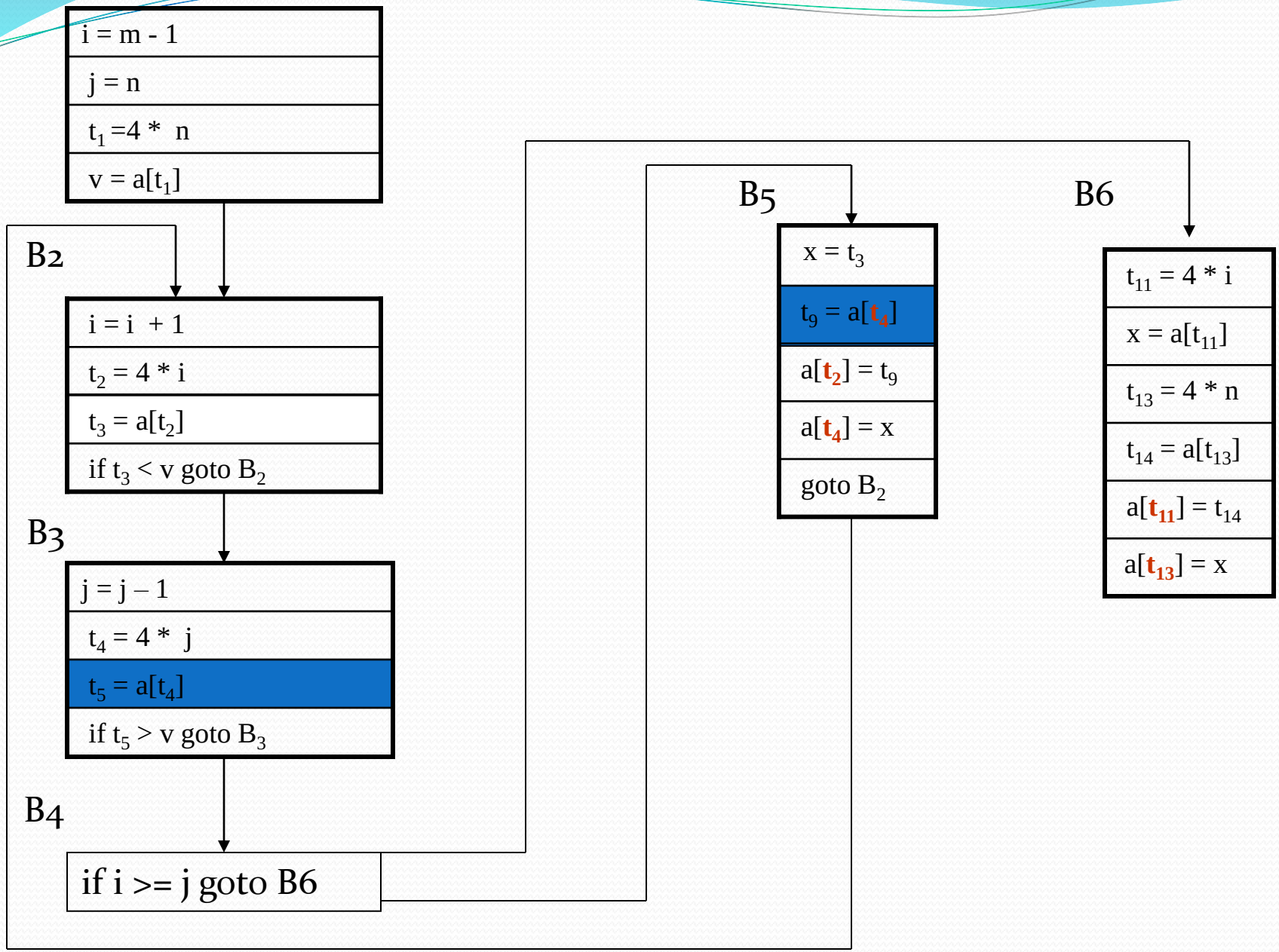
B1 Elimination



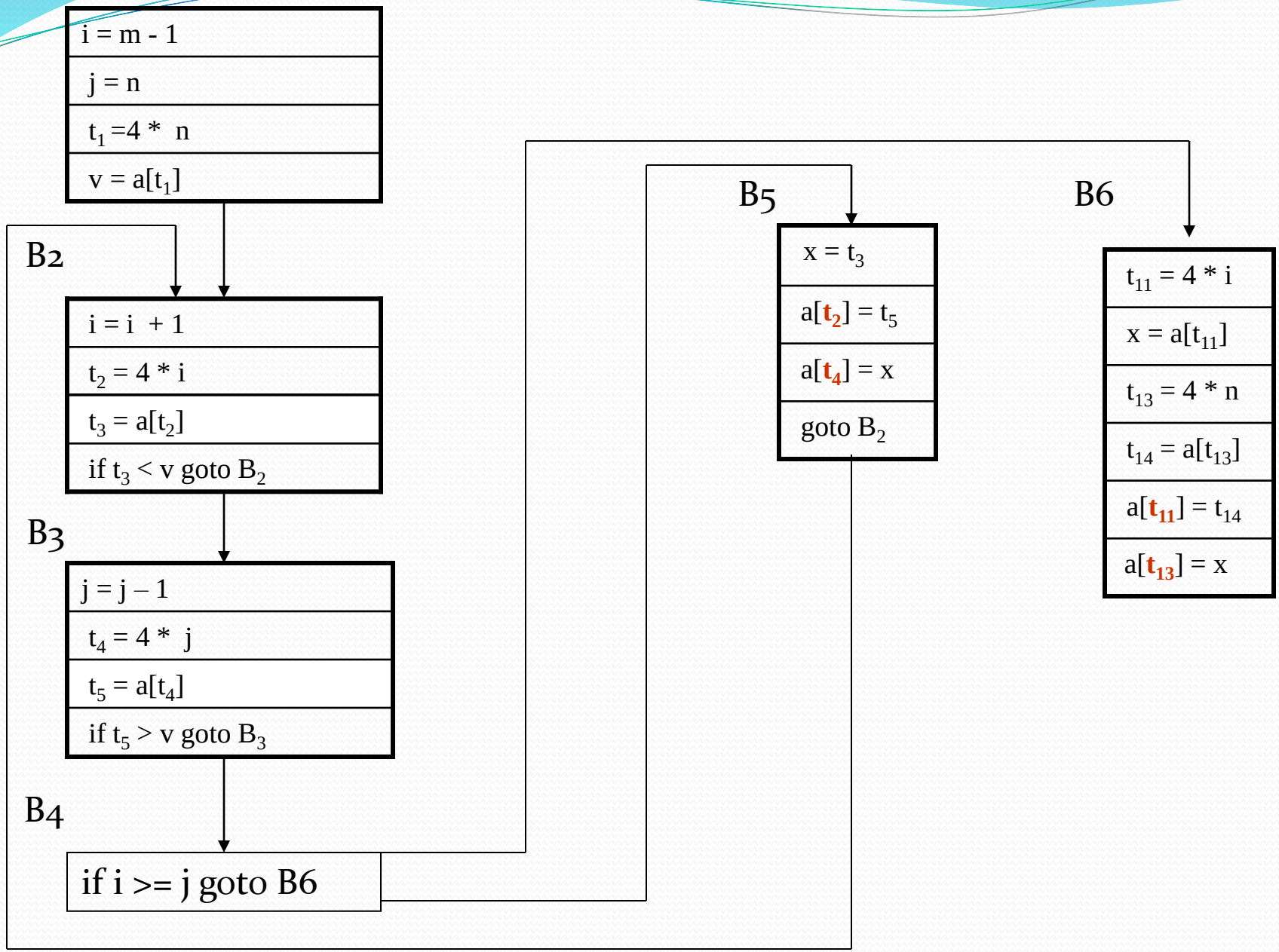
B1 Elimination



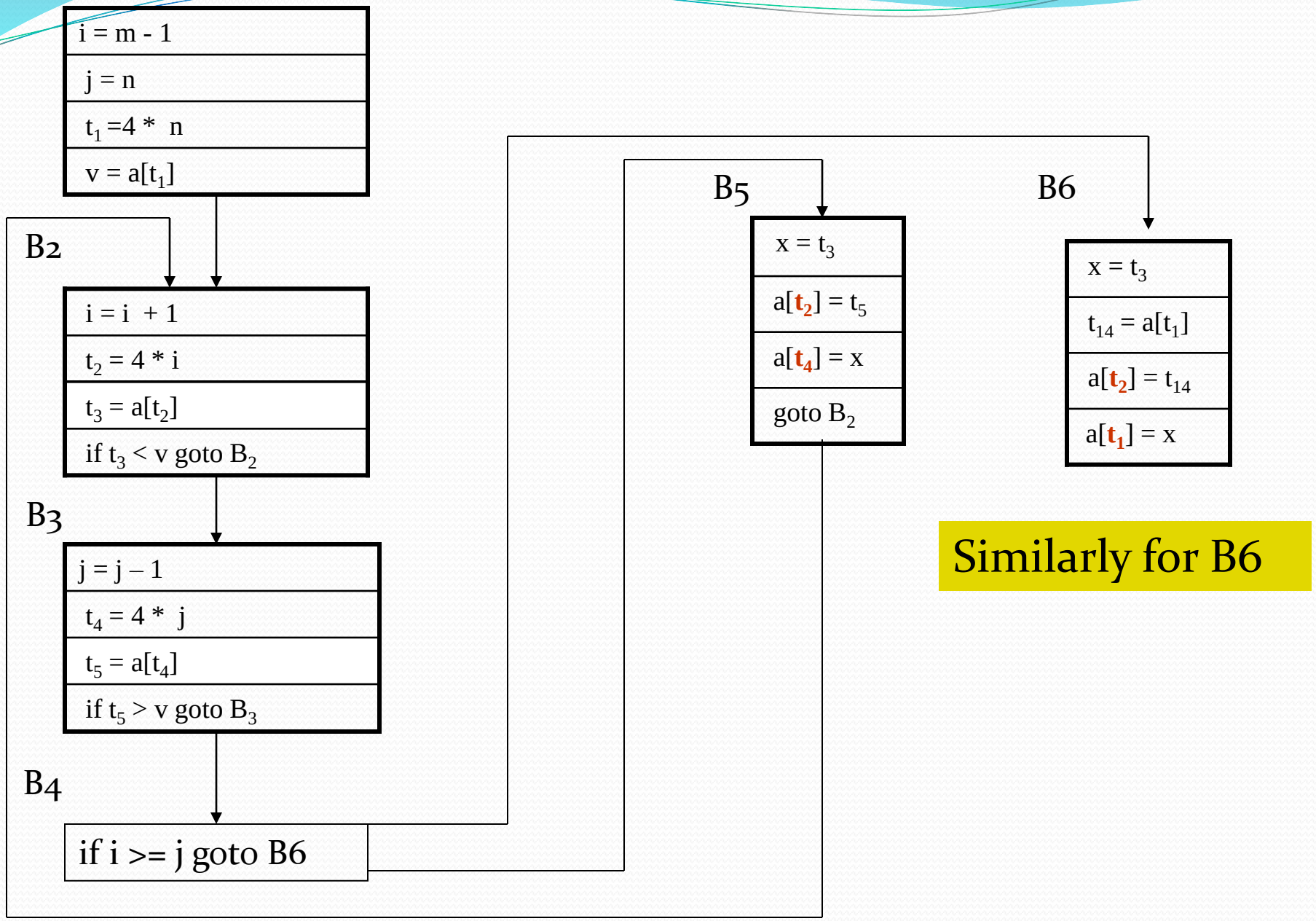
B1 Elimination



B1 Elimination

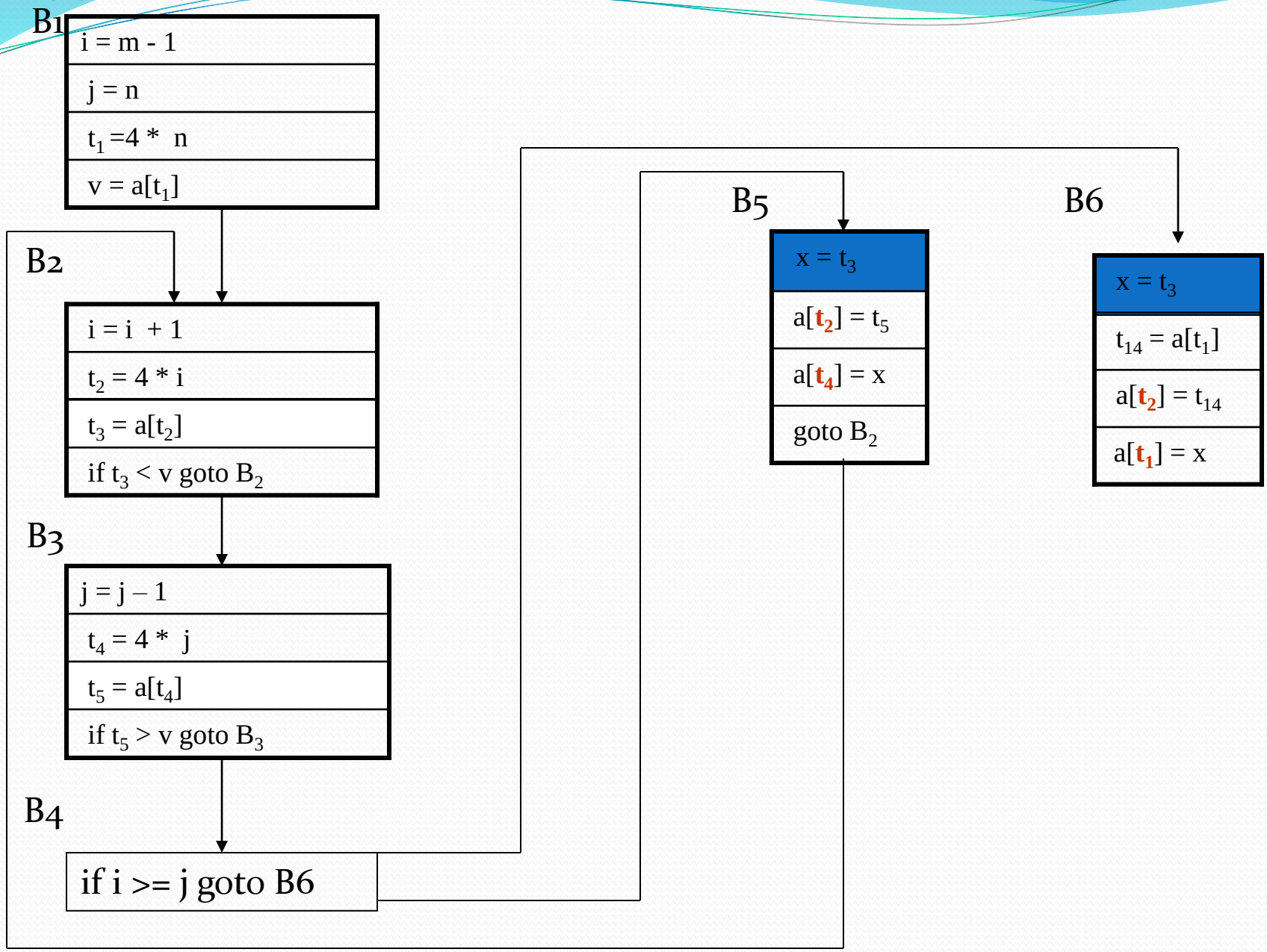


B1 Elimination

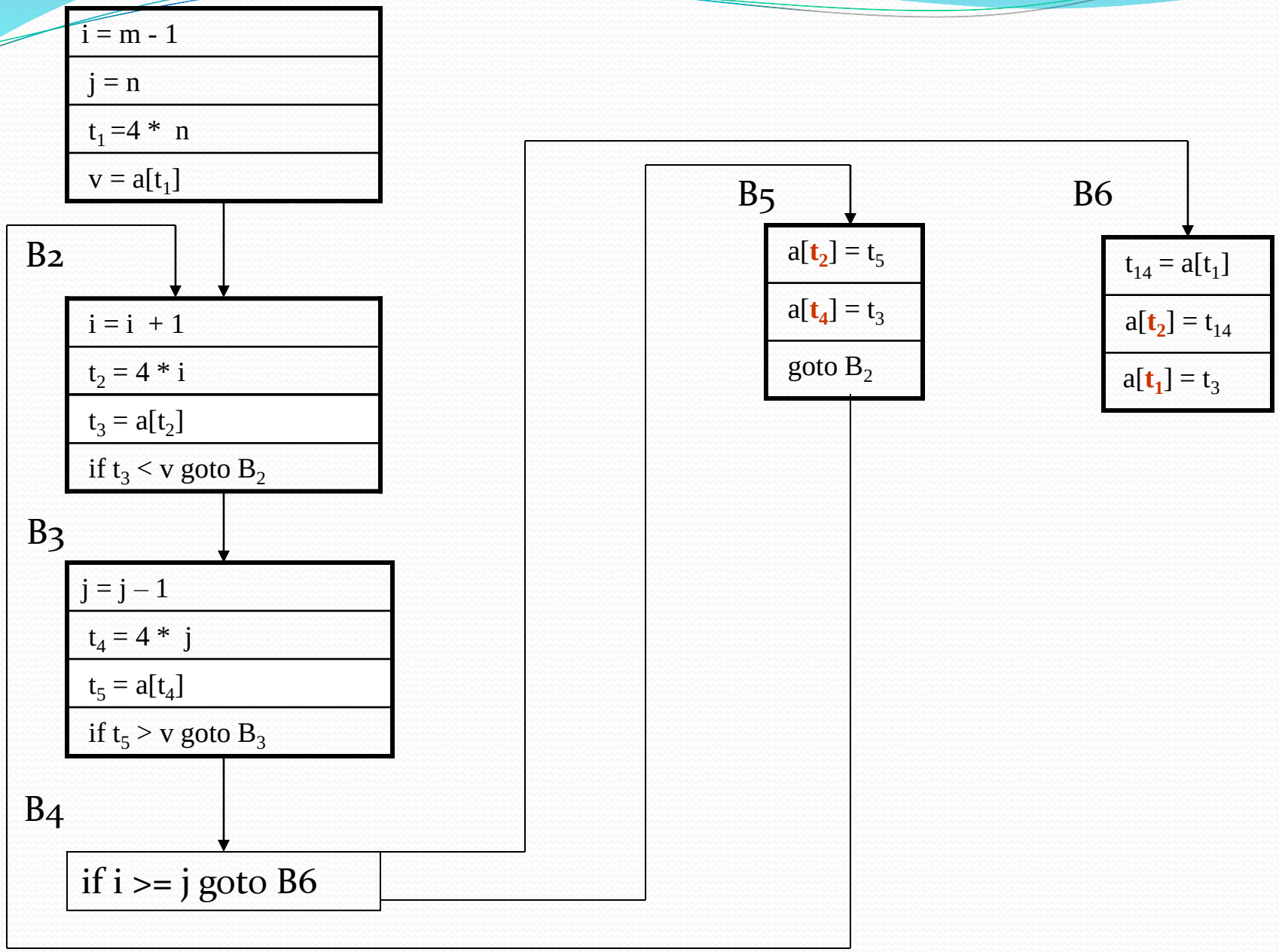


Similarly for B6

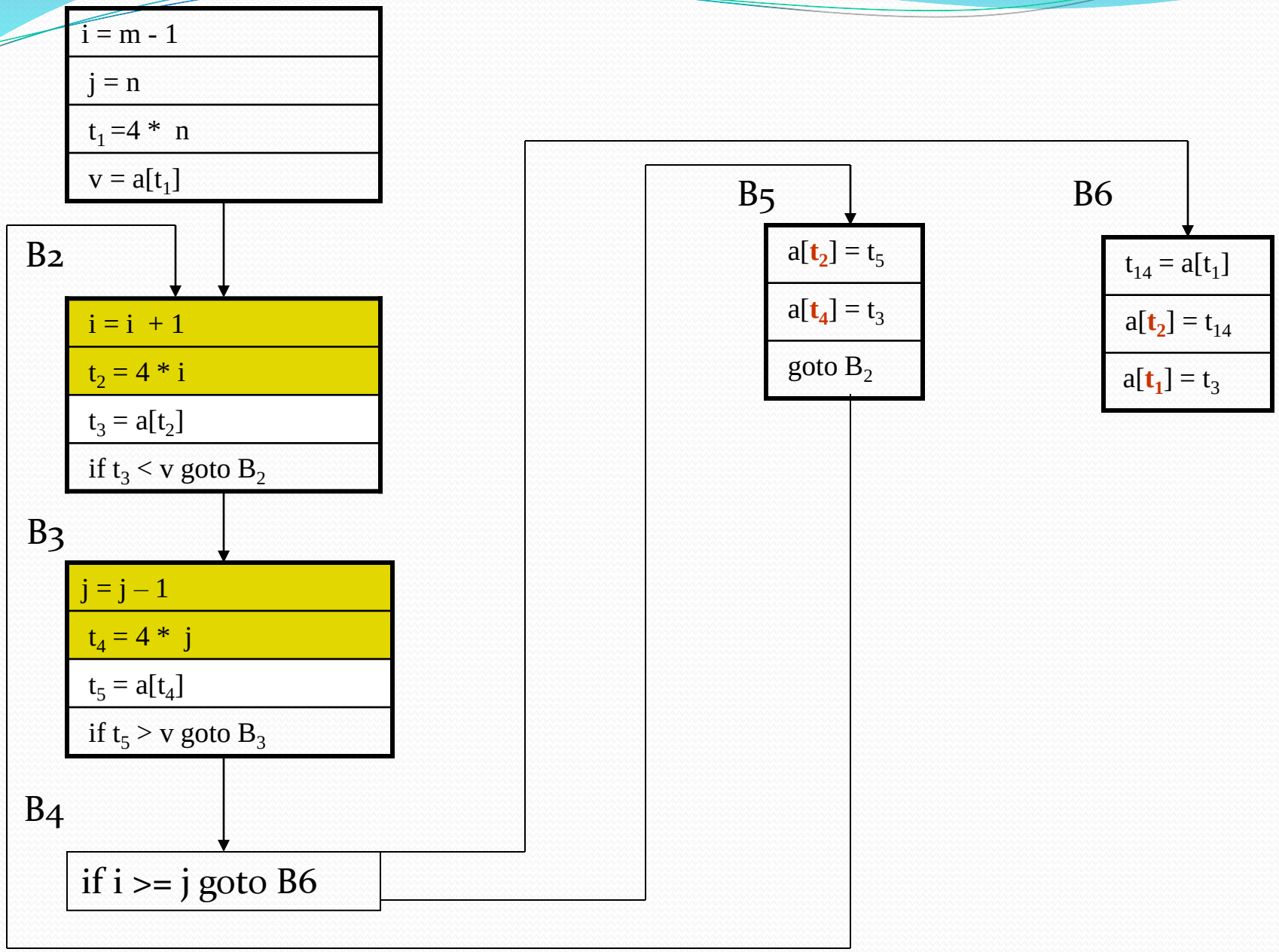
Copy Propogation & Dead Code Elimination



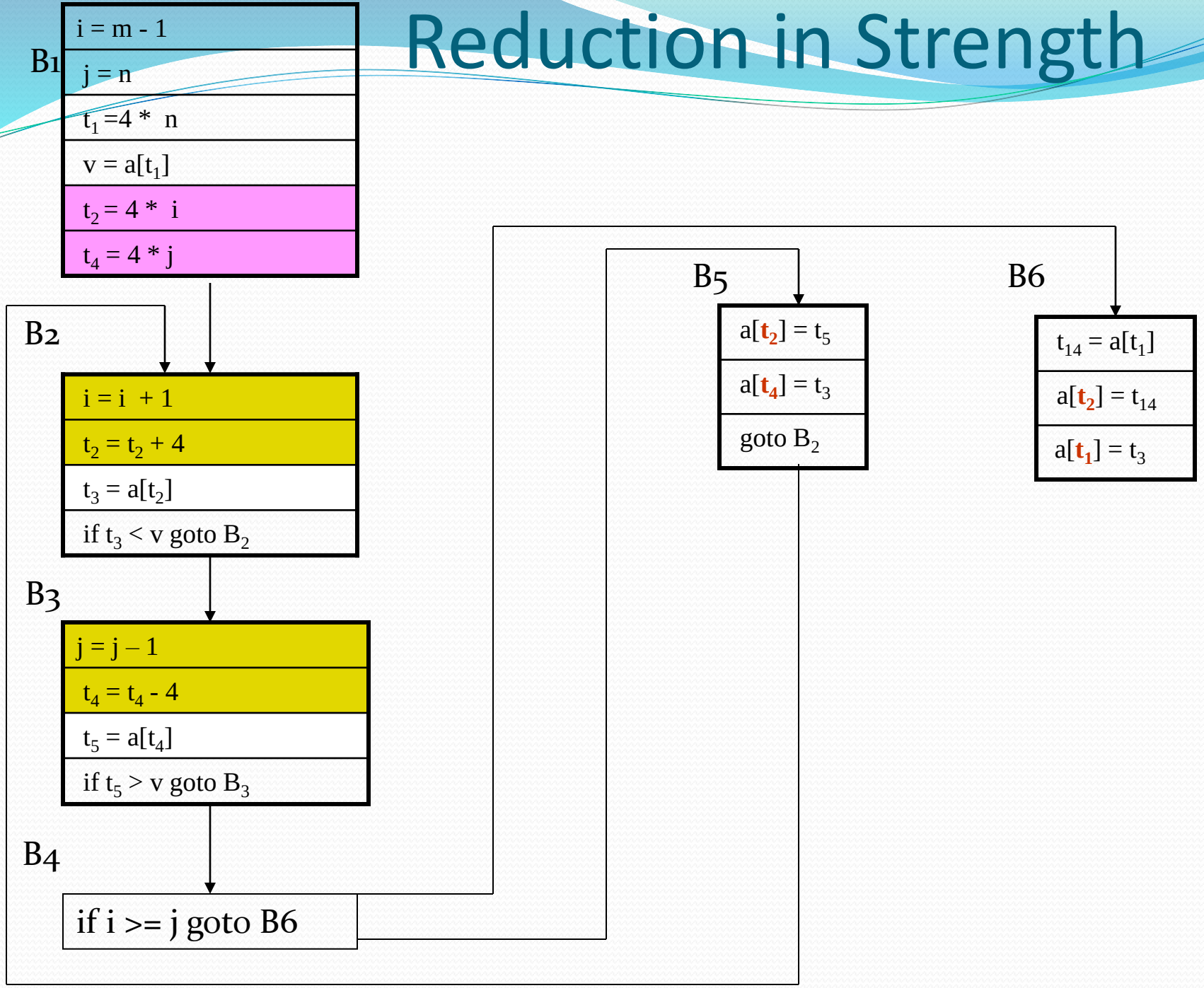
Dead Code Elimination



Reduction in Strength



Reduction in Strength



Induction Variable Elimination