



# 机器无关代码优化

## Part1：常见的优化方式

徐伟

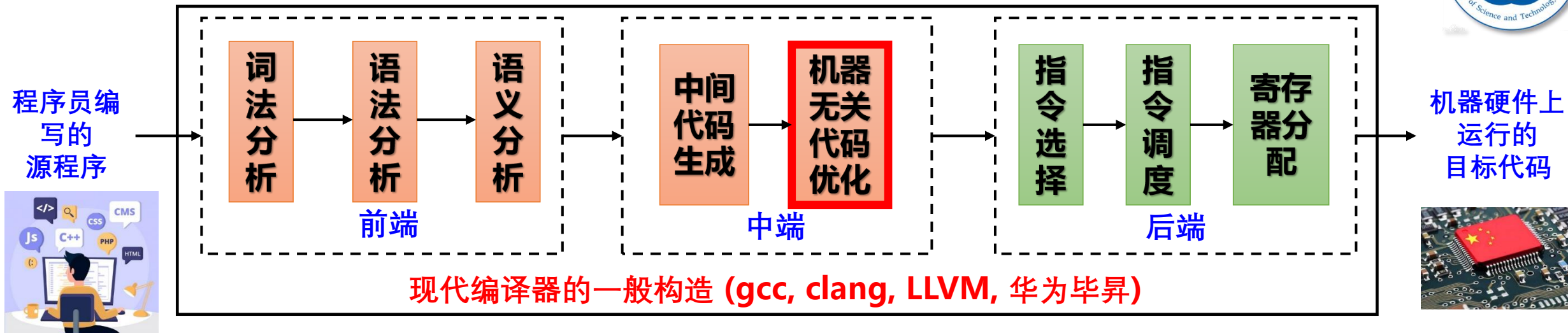
国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年4月17日



# 本节提纲



## • 代码优化的定义及背景

## • 常见的优化方式

- 公共子表达式删除优化
- 死代码删除、复制传播、常量合并
- 循环系列优化
  - 强度削弱、删除归纳变量、代码移动



# 什么是代码优化?



- 在不改变程序运行效果的前提下，对程序代码进行**等价变换**，使之能**生成更加高效目标代码**。
- 优化目标：
  - 运行时间更短
  - 占用空间更小
- 代码优化是编写程序过程中不可或缺的关键环节!

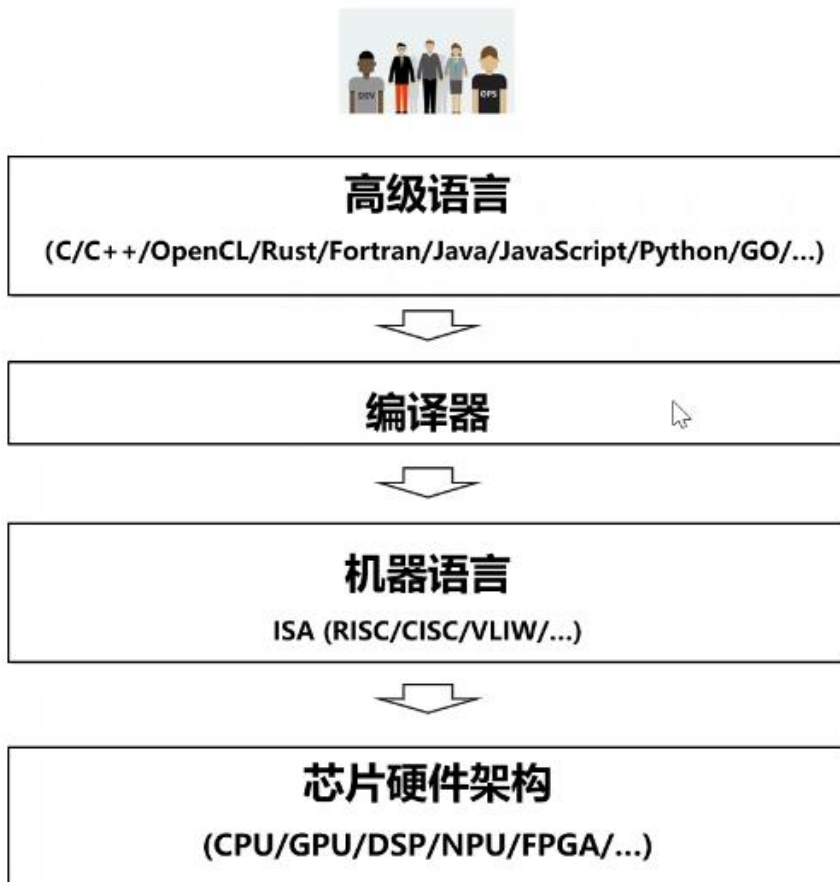


# 为什么需要代码优化?



源头1: 程序员编写的代码存在低效计算

源头2: 代码翻译过程中, 产生了冗余代码



## • 高级语言

- 直接面向开发者
- 与数学公式类似
- 编程效率高

## • 机器语言

- 驱动硬件完成具体任务
- 编程效率低

## • 编译器

- 实现人机交流, 将人类易懂的高级语言翻译成硬件可执行的目标机器语言
- 但目标代码可能运行效率低下



- **程序中存在着许多程序员无法避免的冗余运算**

如  $A[i][j]$  和  $X.f1$  这样访问数组元素和结构体的域的操作

- 编译后，这些访问操作展开成多步低级算术运算
- 对同一个数据结构多次访问导致许多公共低级运算



# 举例——快速排序代码



- 排序是最基本、应用最广泛的可通过计算机高效求解的问题之一
- 快速排序是最经典、效率较高的排序算法之一

```
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;
```

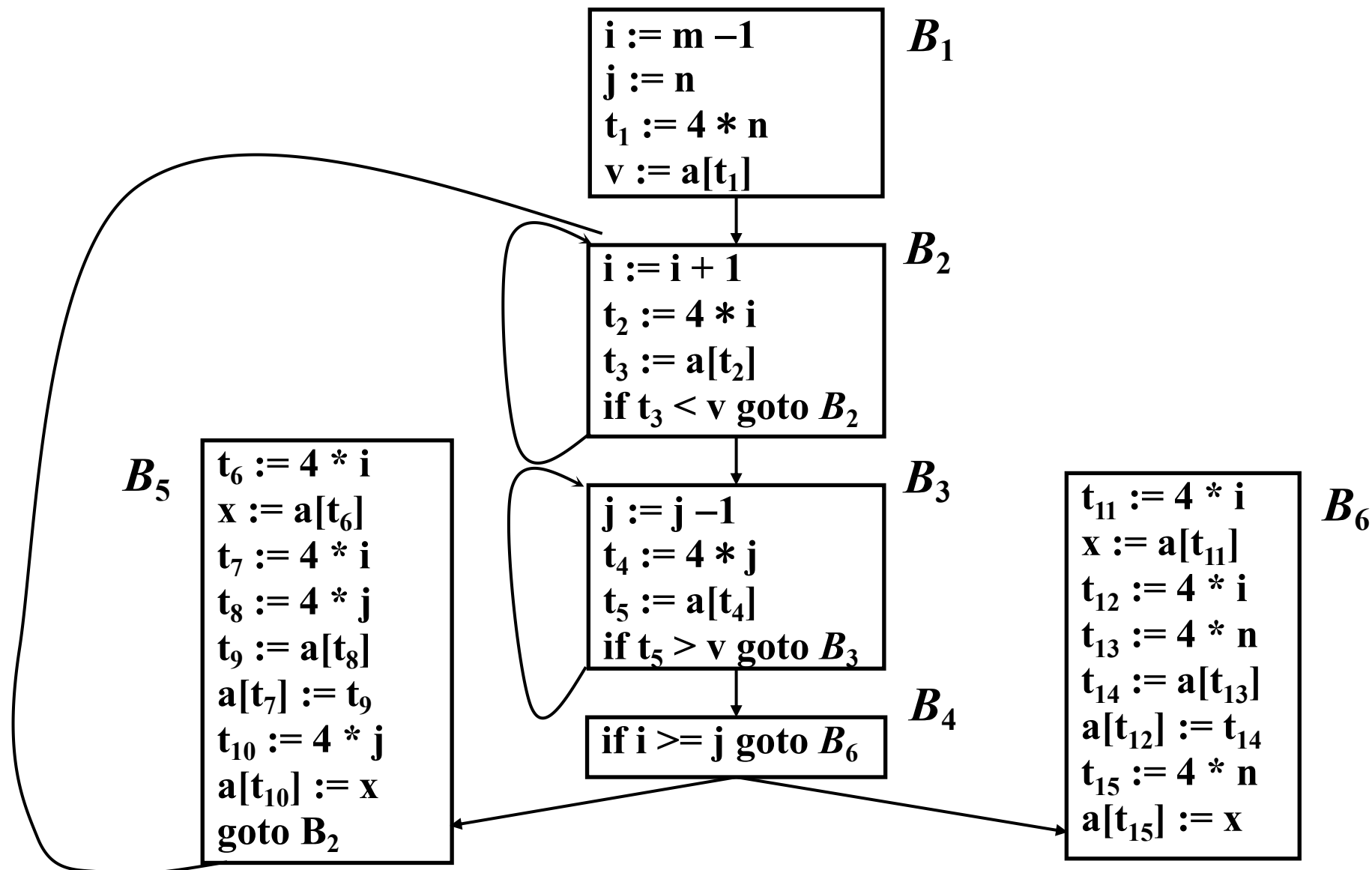
高级语言代码

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

生成的目标机器代码



# 举例——快速排序代码





# 编译器的优化选项



优化等级	简要说明
-Ofast	在-O3级别的基础上，开启更多 <b>激进优化项</b> ，该优化等级不会严格遵循语言标准
-O3	在-O2级别的基础上，开启了更多的 <b>高级优化项</b> ，以编译时间、代码大小、内存为代价获取更高的性能。
-Os	在-O2级别的基础上，开启 <b>降低生成代码体量</b> 的优化
-O2	开启了大多数 <b>中级优化</b> ，会改善编译时间开销和最终生成代码性能
-O/-O1	优化效果介于-O1和-O2之间
-O0	默认优化等级，即 <b>不开启编译优化</b> ，只尝试减少编译时间

延伸阅读: <https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options>





## • 1000000000次循环迭代累加

```
#include <stdlib.h>
#include <time.h>
void main() {
    int loop = 1000000000;
    long sum = 0;
    int start_time = clock();
    int index = 0;
    for (index = 0; index < loop; index++)
    {
        sum += index;
    }
    int end_time = clock();
    printf("Sum : %ld, Time Cost : %lf \n", sum, (end_time - start_time) * 1.0 / CLOCKS_PER_SEC);
}
```

**循环次数定义**

**开始计时**

**循环体**

**结束计时**

**代码运行时间输出**

- gcc -O0 无优化执行

```
gloit@gloit-x1c > ~/2022_compiler_demo } master gcc -O0 add.c
gloit@gloit-x1c > ~/2022_compiler_demo } master ./a.out
Sum: 499999999500000000, Time Cost: 3.415244
```

- gcc -O1 中级优化执行

```
gloit@gloit-x1c > ~/2022_compiler_demo } master gcc -O1 add.c
gloit@gloit-x1c > ~/2022_compiler_demo } master ./a.out
Sum: 499999999500000000, Time Cost: 0.554717
```

- gcc -O2 高级优化执行

```
gloit@gloit-x1c > ~/2022_compiler_demo } master gcc -O2 add.c
gloit@gloit-x1c > ~/2022_compiler_demo } master ./a.out
Sum: 499999999500000000, Time Cost: 0.000002
```

性能提升5倍

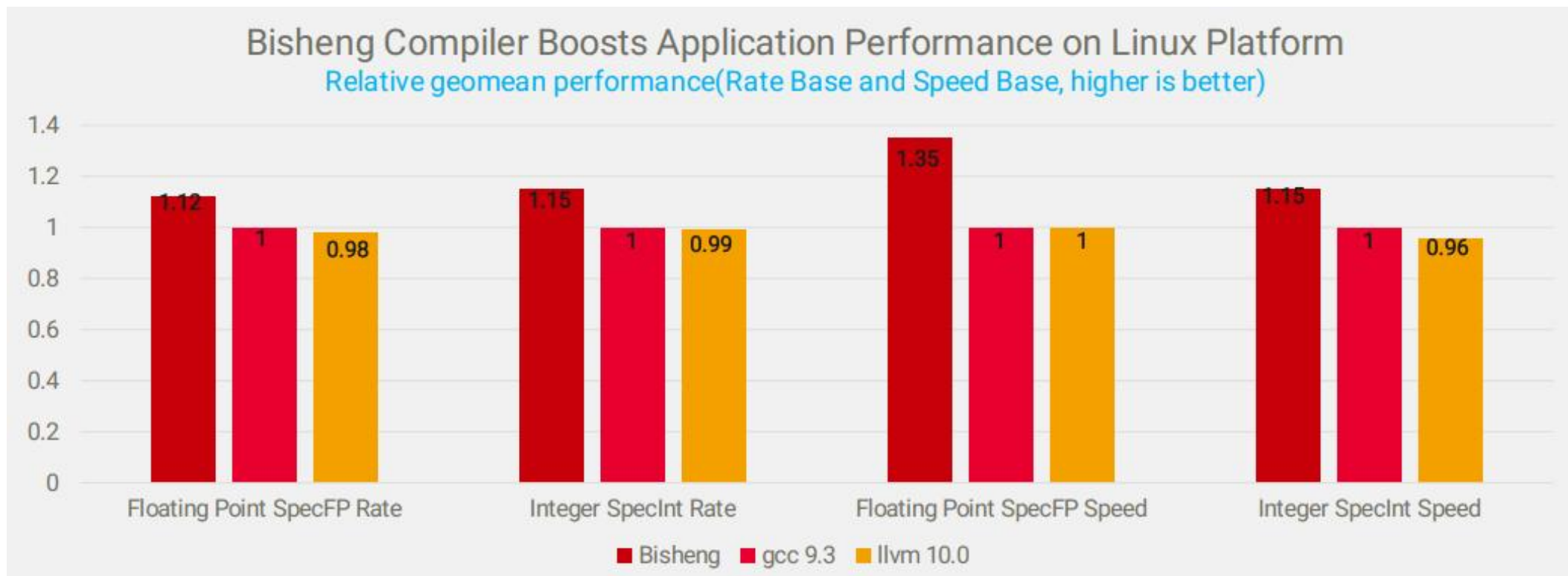
性能提升数十万倍



# 国产开源编译器——毕昇编译器



- 毕昇编译器通过**编译优化**提升鲲鹏硬件平台上业务的性能体验，SPEC2017性能较业界编译器平均**高15%以上**。



SPEC作为业界芯片性能评分标准，SPEC的分数可以直观的体现出硬件的性能，越高越好



# 本节提纲



## • 代码优化的定义及背景

## • 常见的优化方式

- 公共子表达式删除优化
- 死代码删除、复制传播、常量合并
- 循环系列优化
  - 强度削弱、删除归纳变量、代码移动



## • 公共子表达式

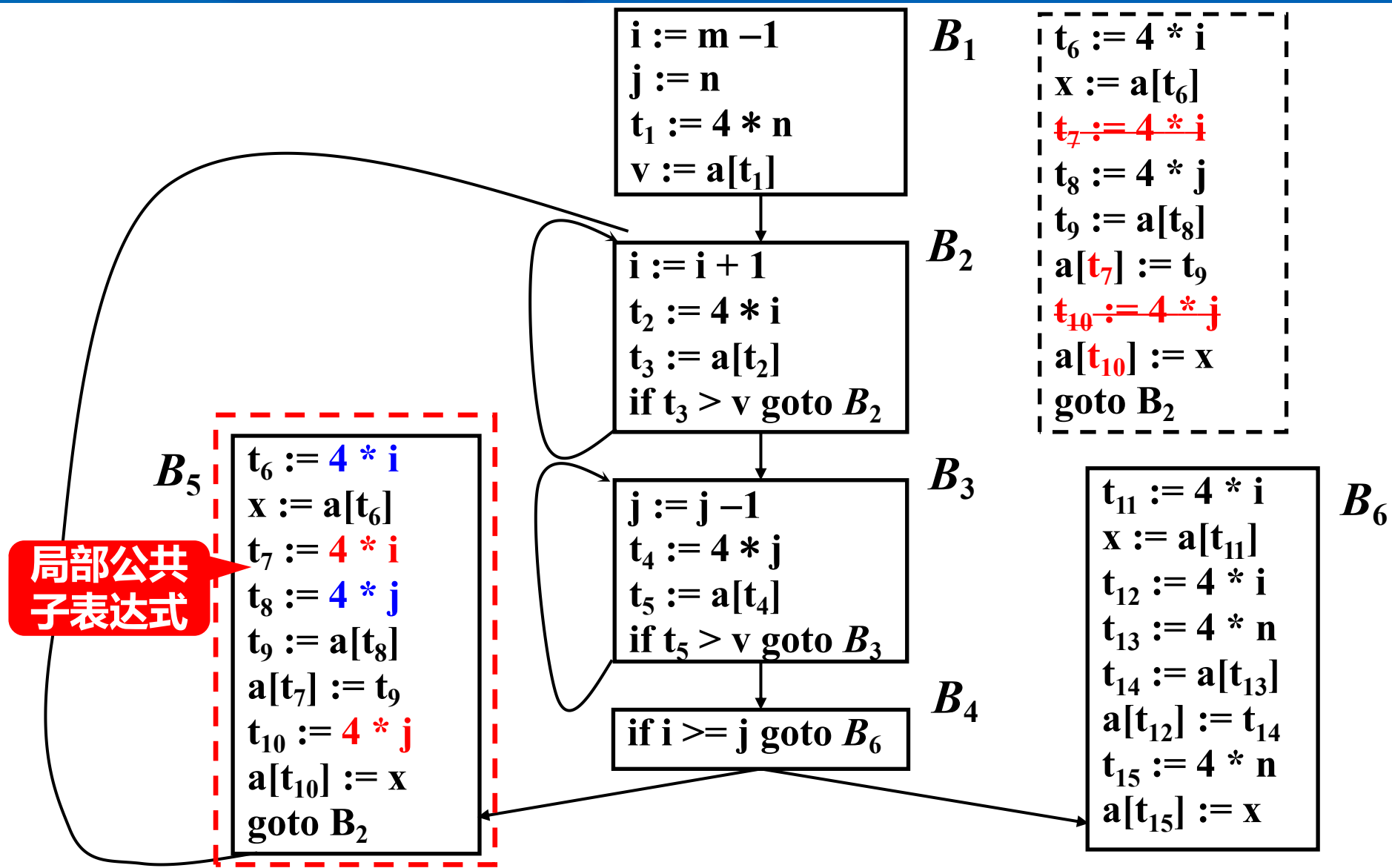
- 如  $x \text{ op } y$  已被计算过，且到现在为止， $x$  和  $y$  的值未变，那么该表达式的本次出现是公共子表达式

## • 公共子表达式的删除

- 本次计算可以被删除
- 其值用上一次计算值替代



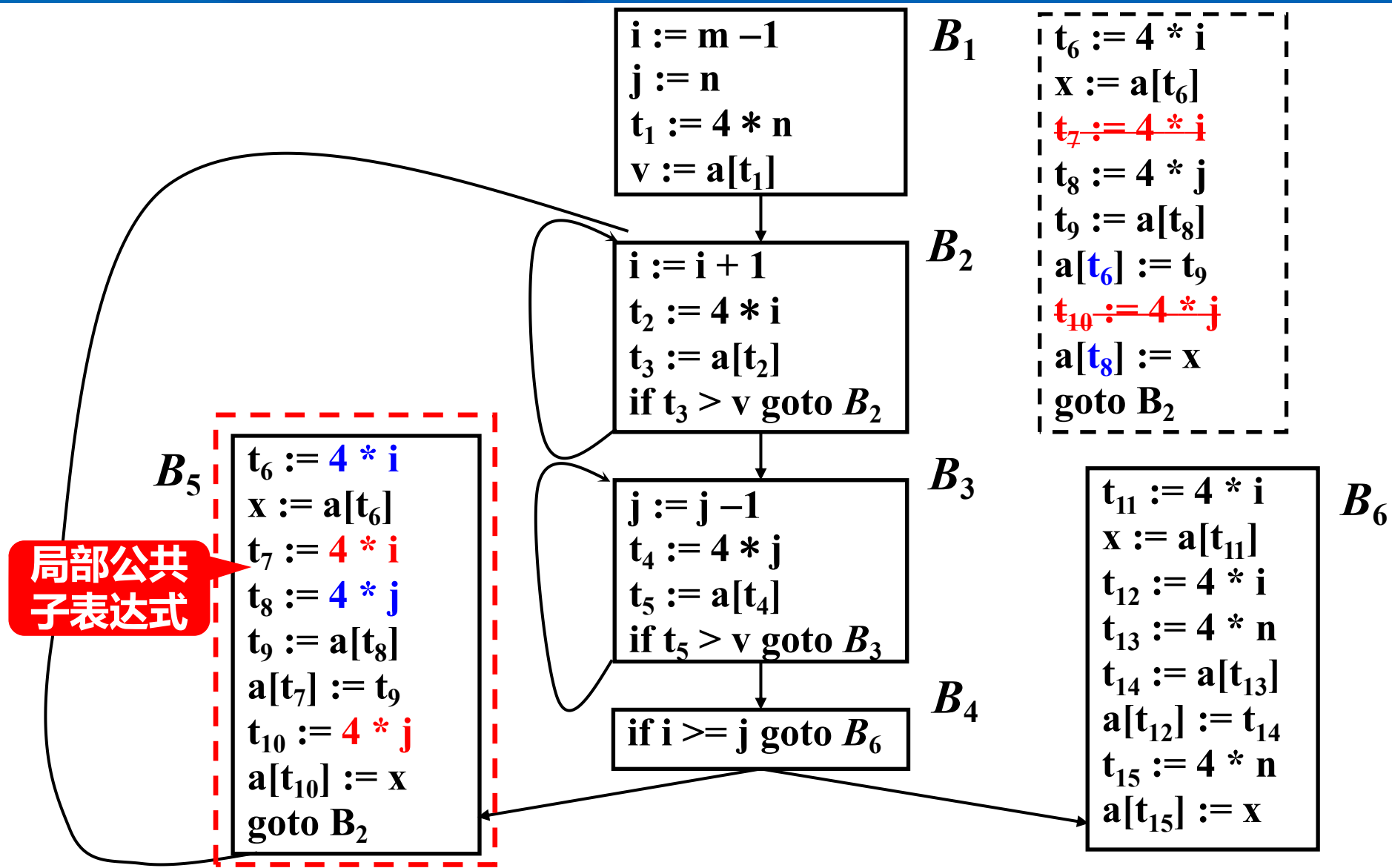
# 快排中的公共子表达式删除





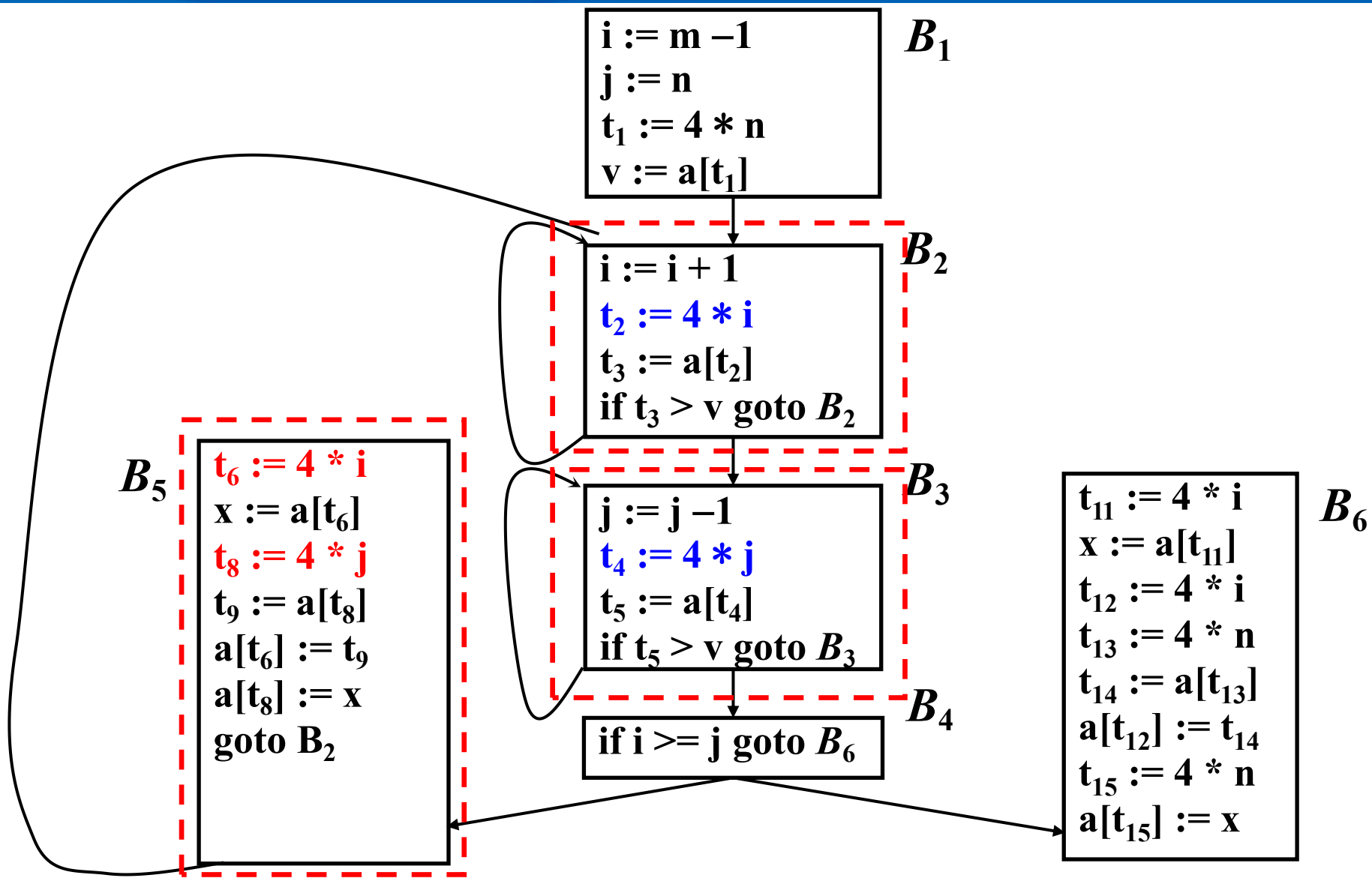


# 快排中的公共子表达式删除





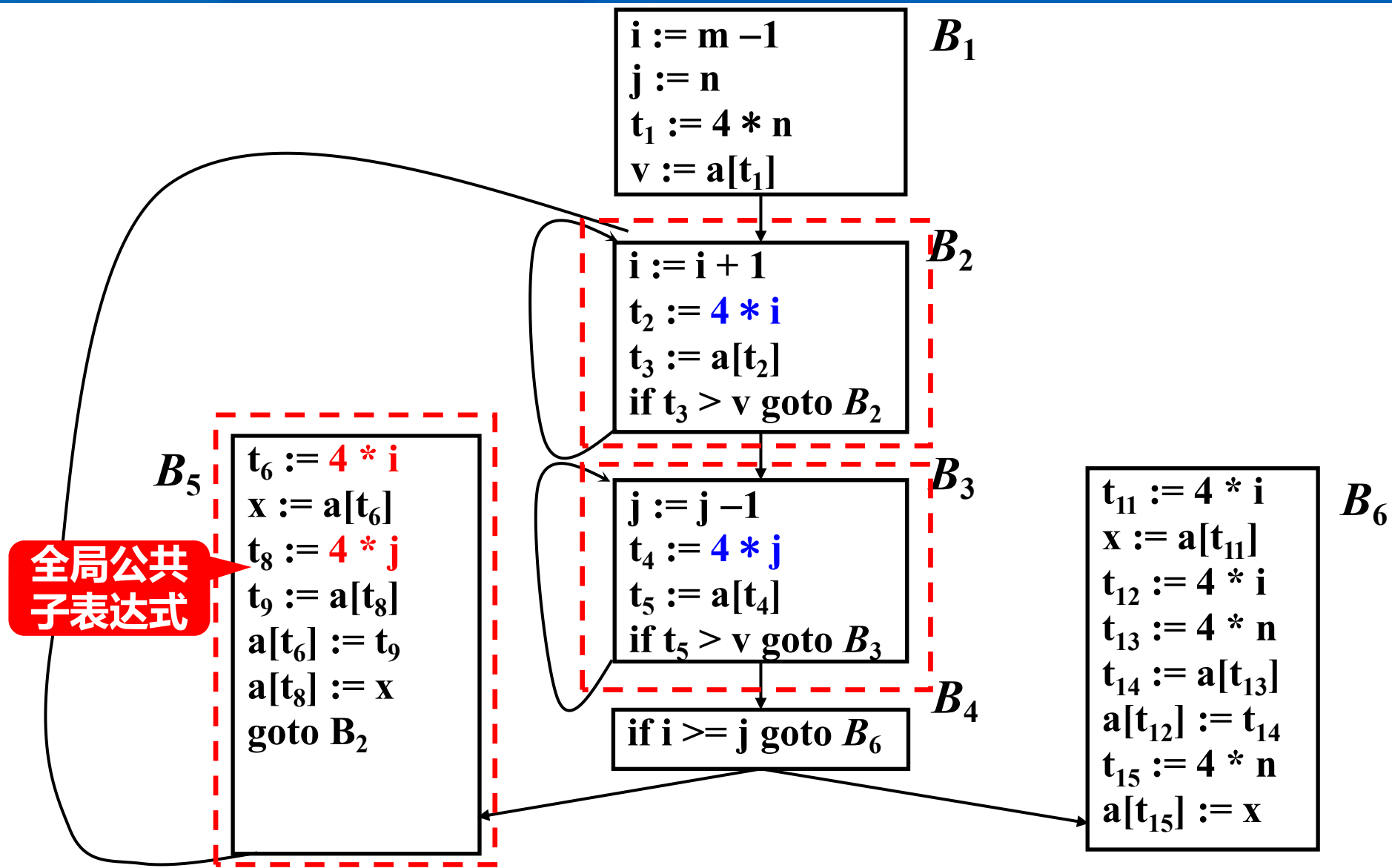
# 快排中的公共子表达式删除





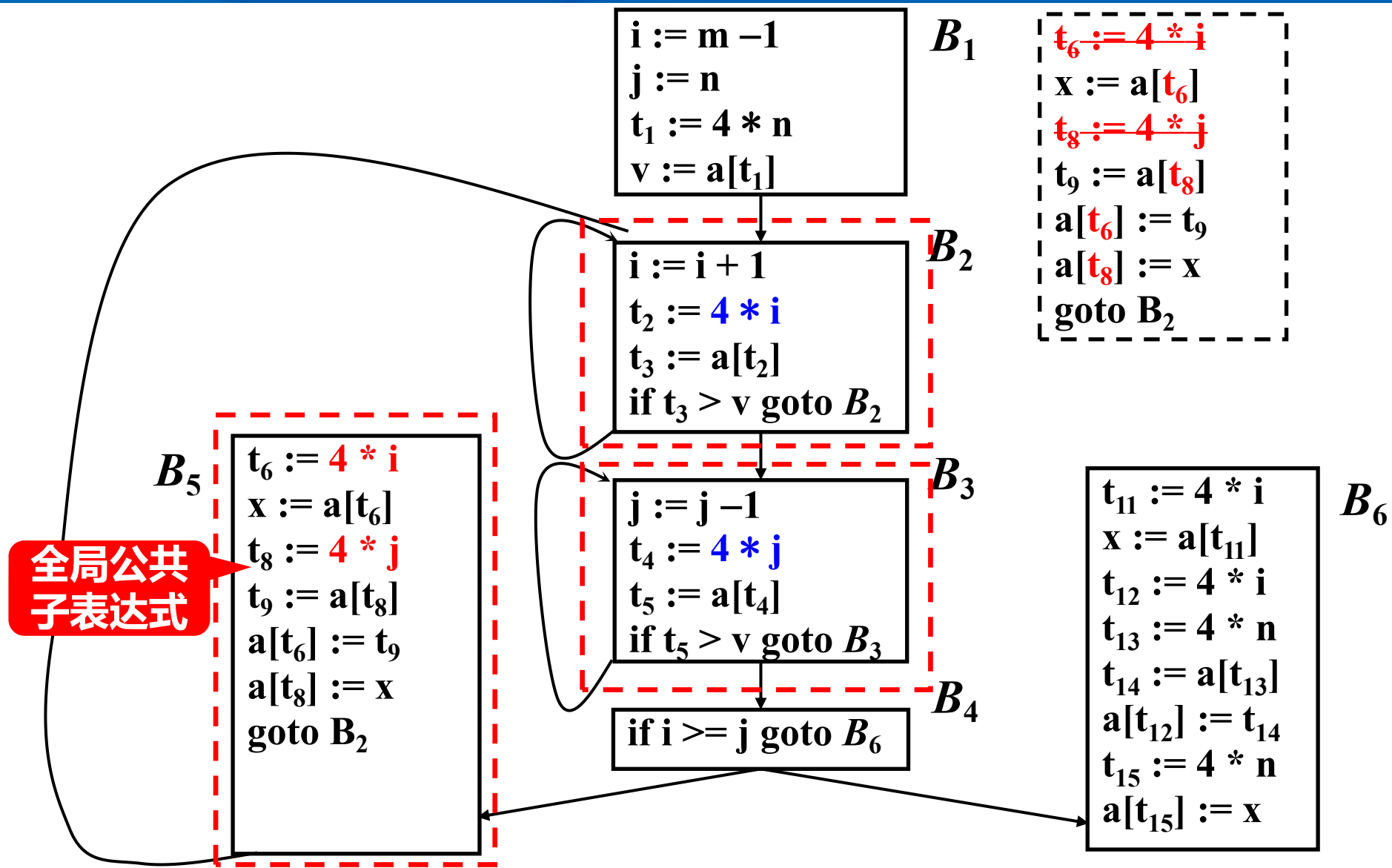


# 快排中的公共子表达式删除



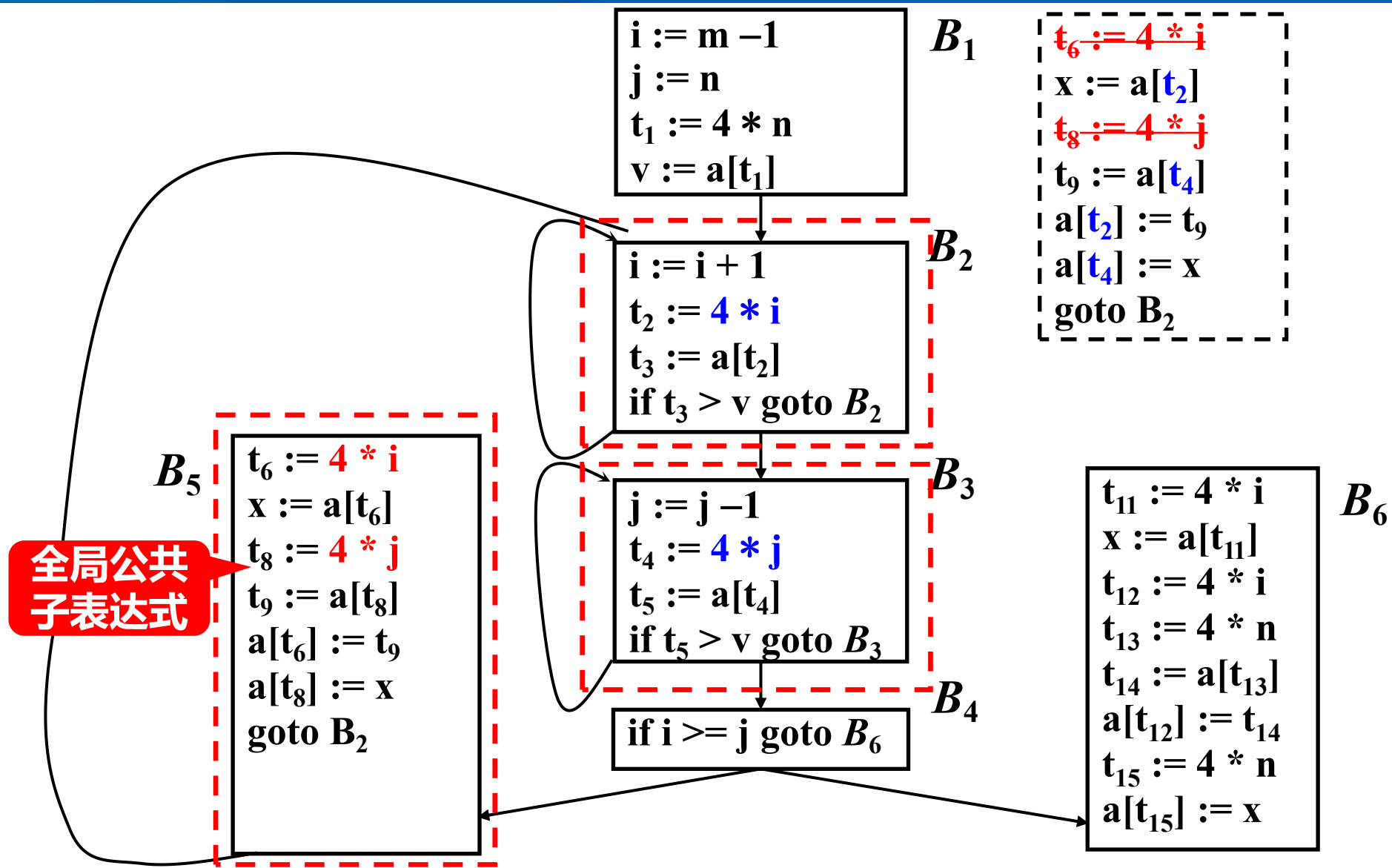


# 快排中的公共子表达式删除



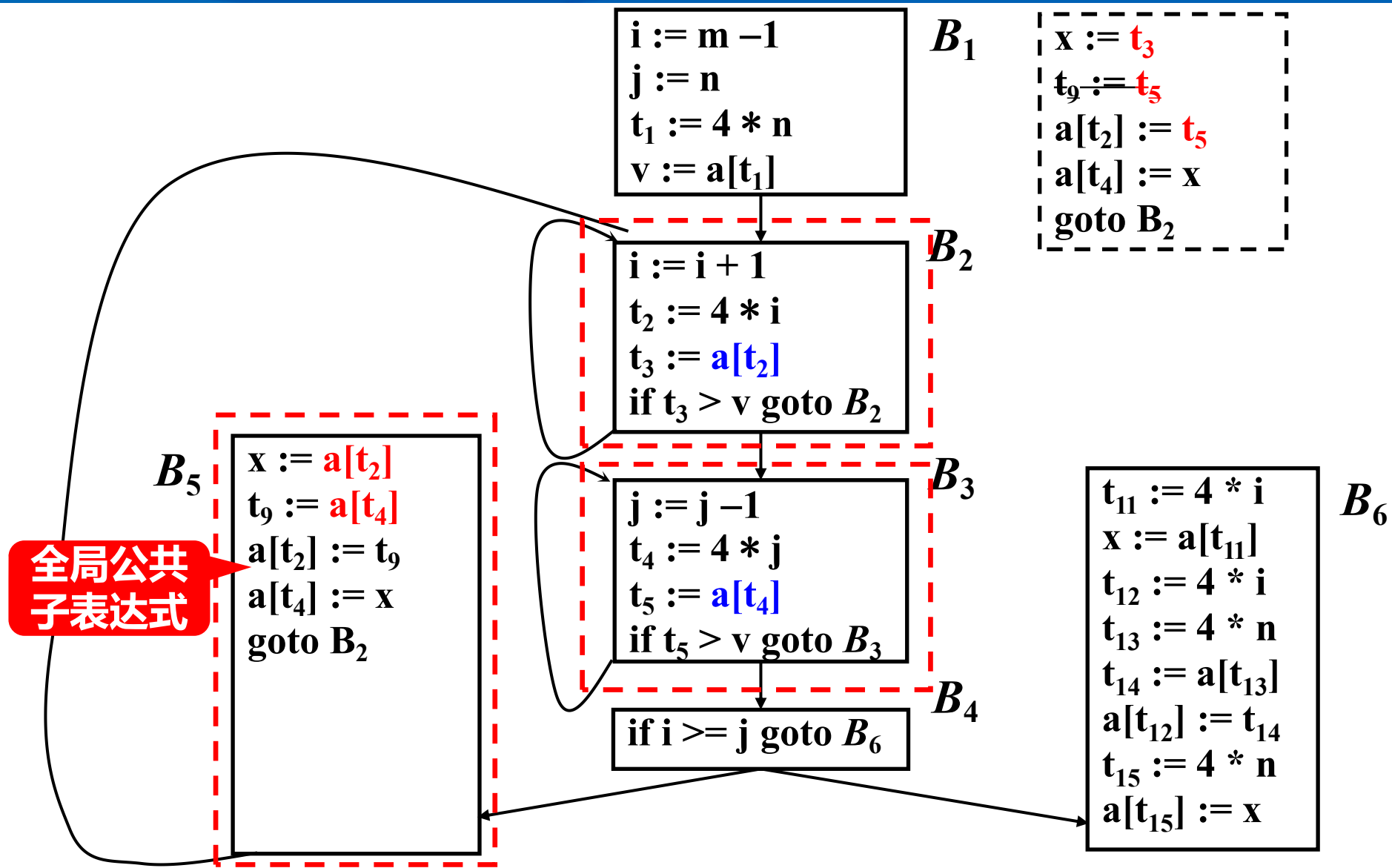


# 快排中的公共子表达式删除



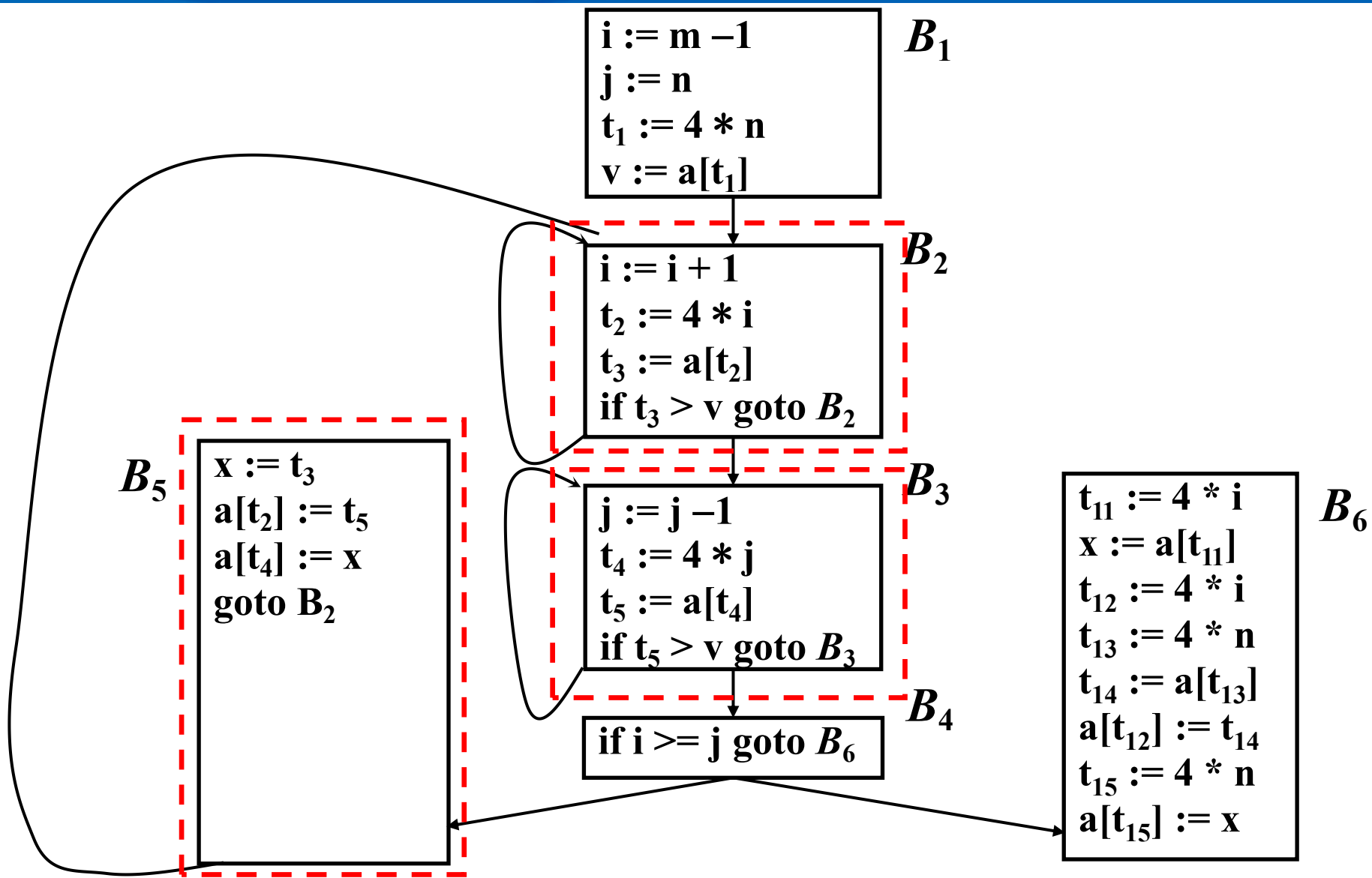


# 快排中的公共子表达式删除



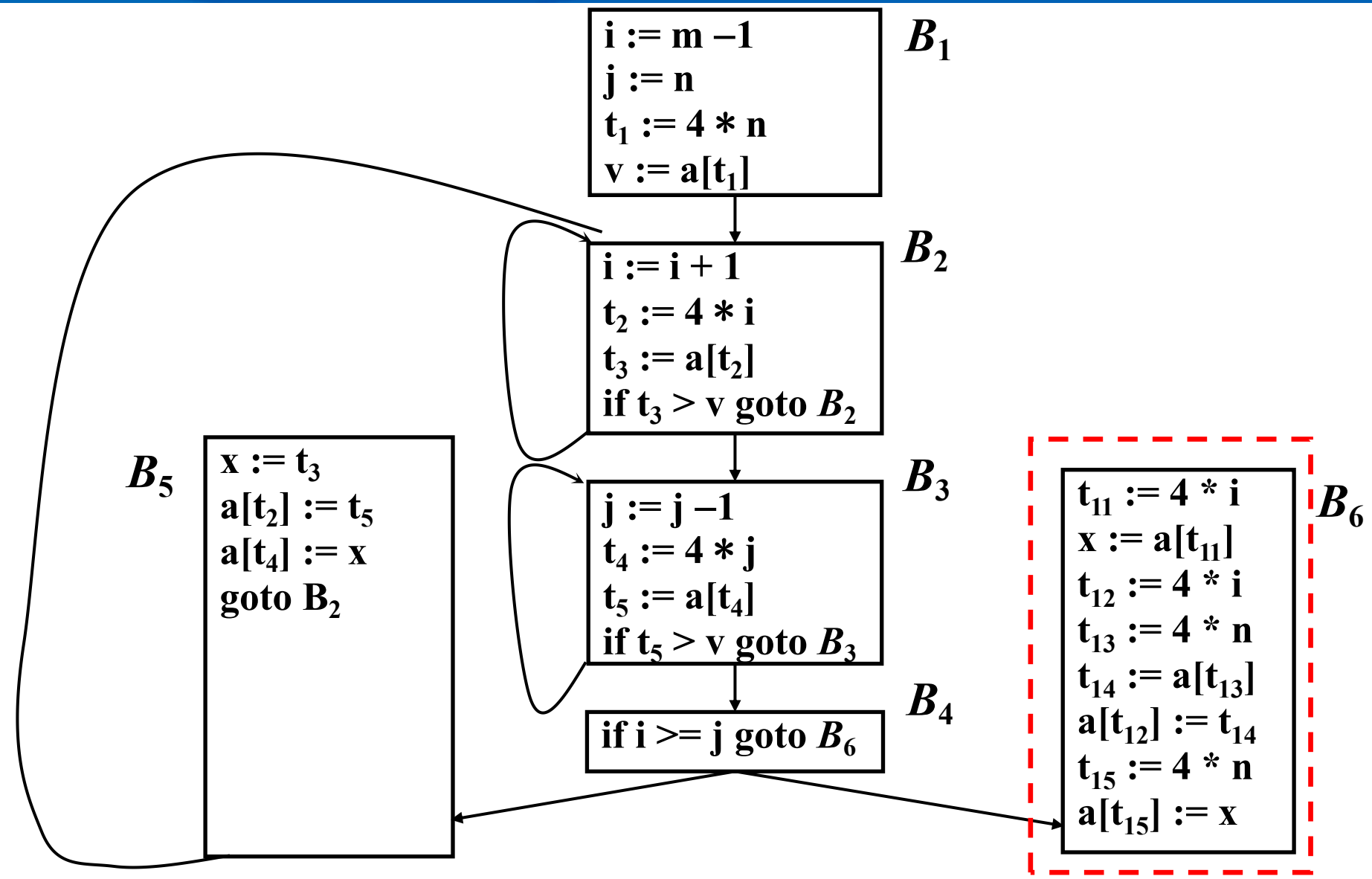


# 快排中的公共子表达式删除



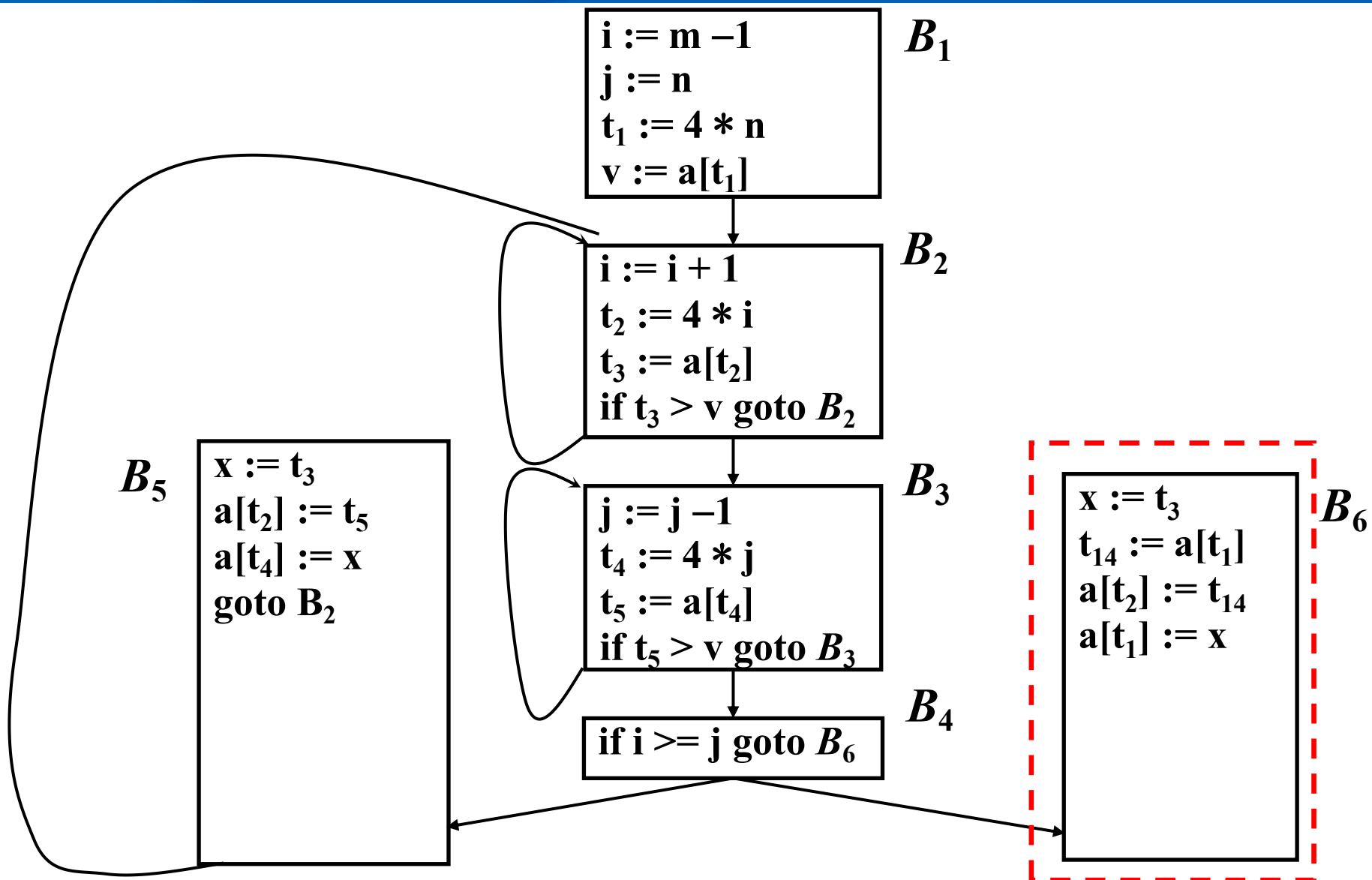


# 快排中的公共子表达式删除



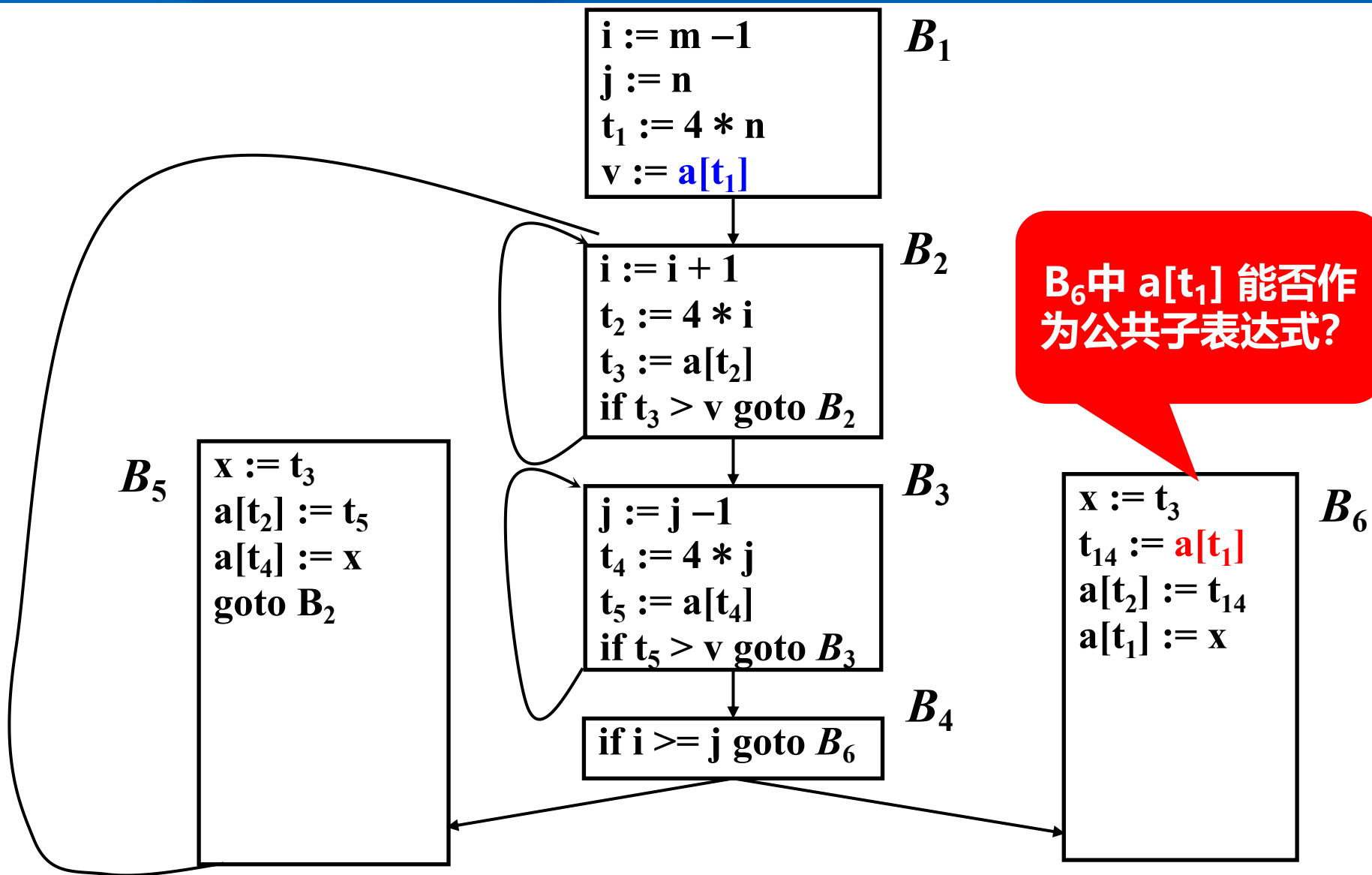


# 快排中的公共子表达式删除





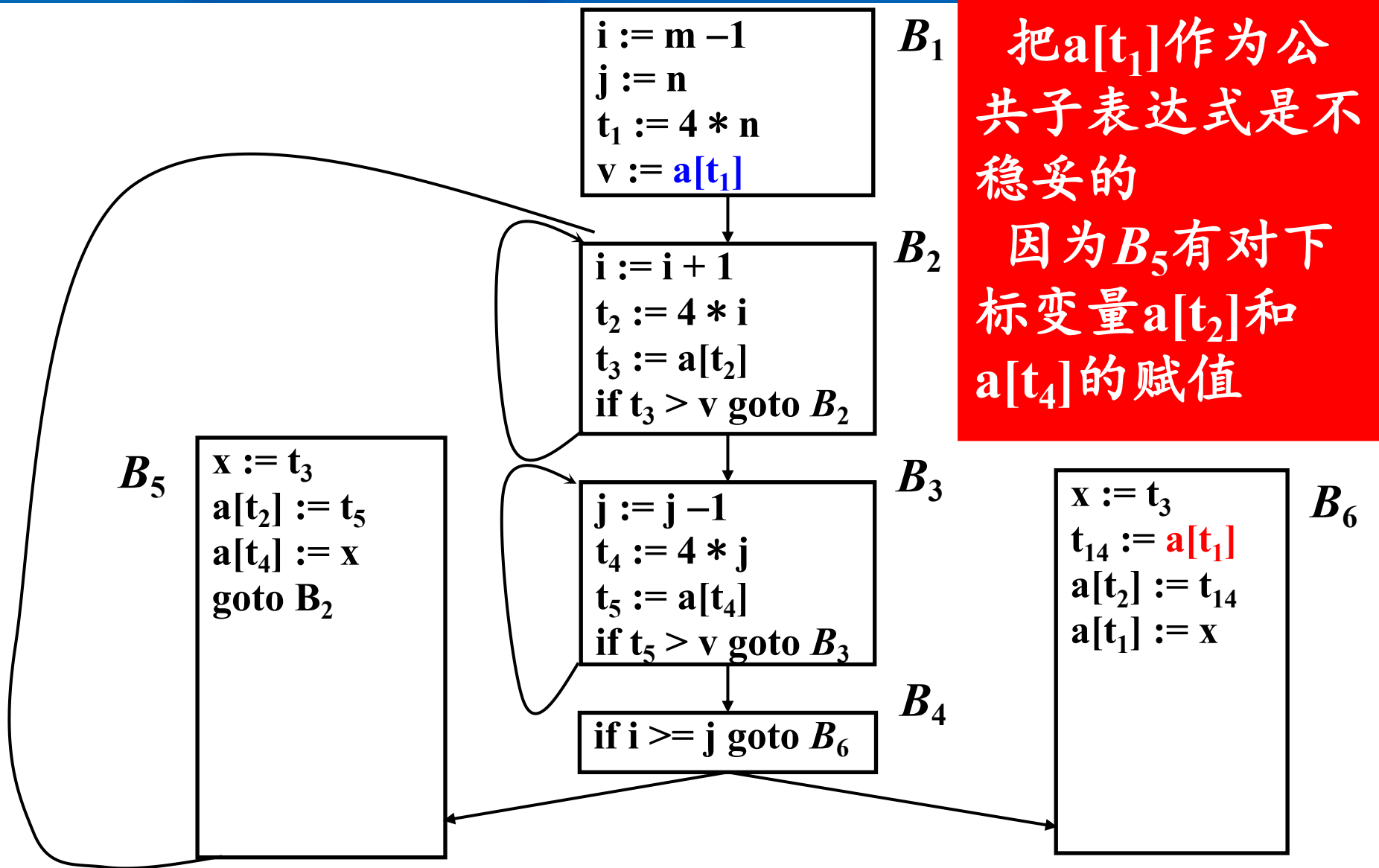
# 快排中的公共子表达式删除







# 快排中的公共子表达式删除



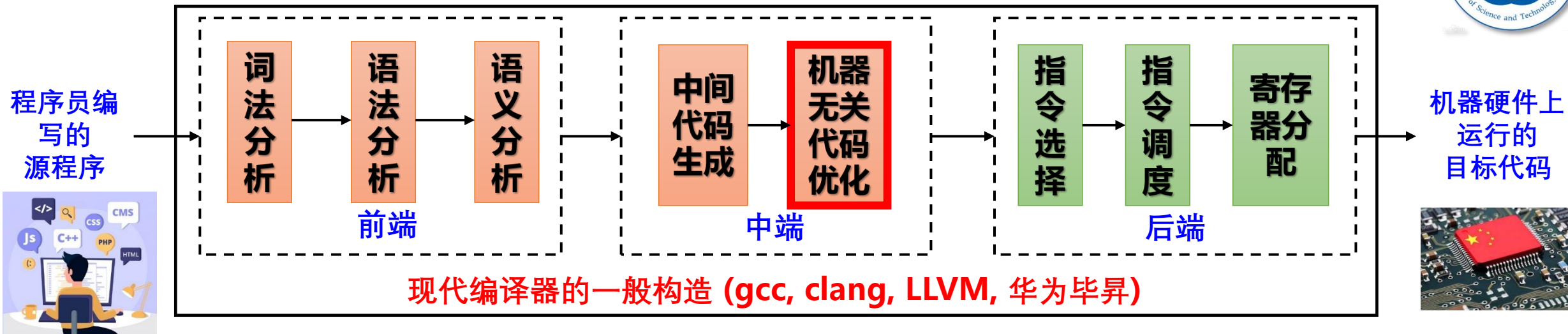


## • 工程实现公共子表达式删除的关键理论与技术点

- 可用表达式数据流分析 (教材第9章第9.2节)
- 公共子表达式删除的工业界实现代码
  - **文档**链接: [https://llvm.org/doxygen/EarlyCSE\\_8cpp.html](https://llvm.org/doxygen/EarlyCSE_8cpp.html)
  - **源码**链接: [https://llvm.org/doxygen/EarlyCSE\\_8cpp\\_source.html](https://llvm.org/doxygen/EarlyCSE_8cpp_source.html)
- 基于Global Value Numbering算法的工业界实现代码
  - **文档**链接: <https://llvm.org/docs/Passes.html#gvn-global-value-numbering>
  - **源码**链接: [https://llvm.org/doxygen/GVN\\_8cpp\\_source.html](https://llvm.org/doxygen/GVN_8cpp_source.html)



# 本节提纲



## • 代码优化的定义及背景

## • 常见的优化方式

- 公共子表达式删除优化
- 死代码删除、复制传播、常量合并
- 循环系列优化
  - 强度削弱、删除归纳变量、代码移动



- 死代码是指计算的结果永远不被引用的语句

**例： 为便于调试，可能在程序上加打印语句，测试后改成右边的形式**

`debug = true;`

`...`

`if (debug) print ...`

|

`debug = false;`

|

`...`

|

`if (debug) print ...`



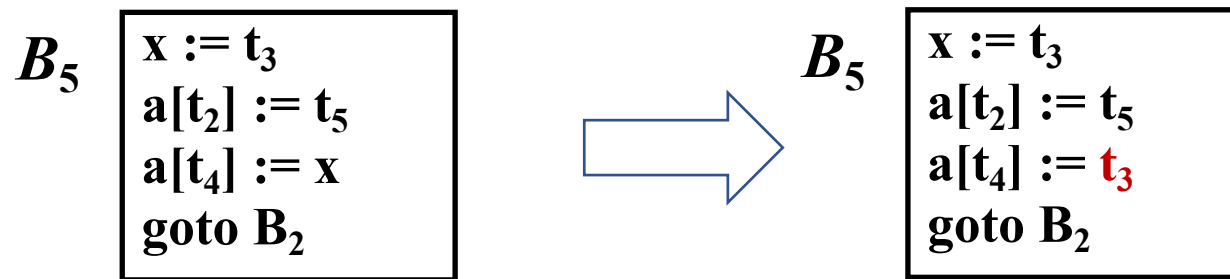
- 死代码是指计算的结果永远不被引用的语句
- 一些优化变换可能会引起死代码
  - 如：复制传播、常量合并

例： 为便于调试，可能在程序上加打印语句，测试后改成右边的形式

debug = true;		debug = false;
. . .		. . .
if (debug) print ...		if (debug) print ...

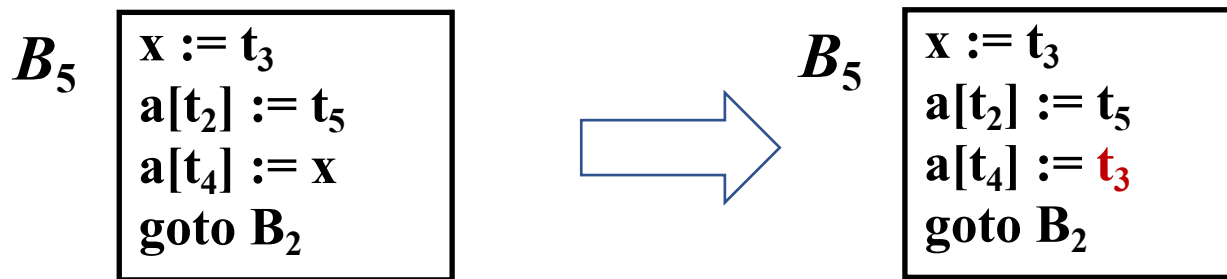


- 定义：在复制语句  $x = y$  之后尽可能用  $y$  代替  $x$





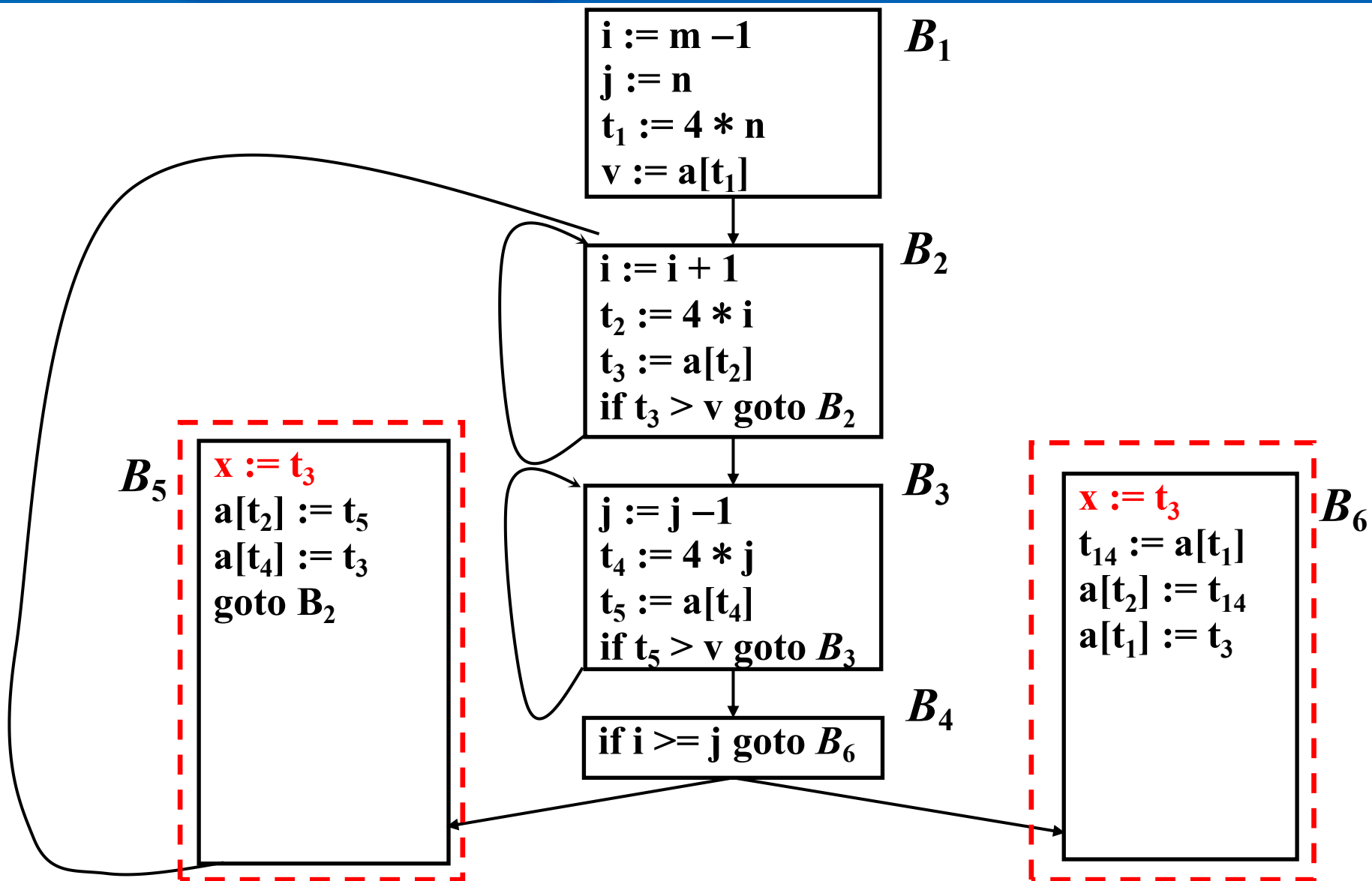
- 定义：在复制语句  $x = y$  之后尽可能用  $y$  代替  $x$



- 常用的公共子表达式删除和其他一些优化会引入一些复制语句
- 复制传播本身没有优化的意义，但可以给死代码删除创造机会



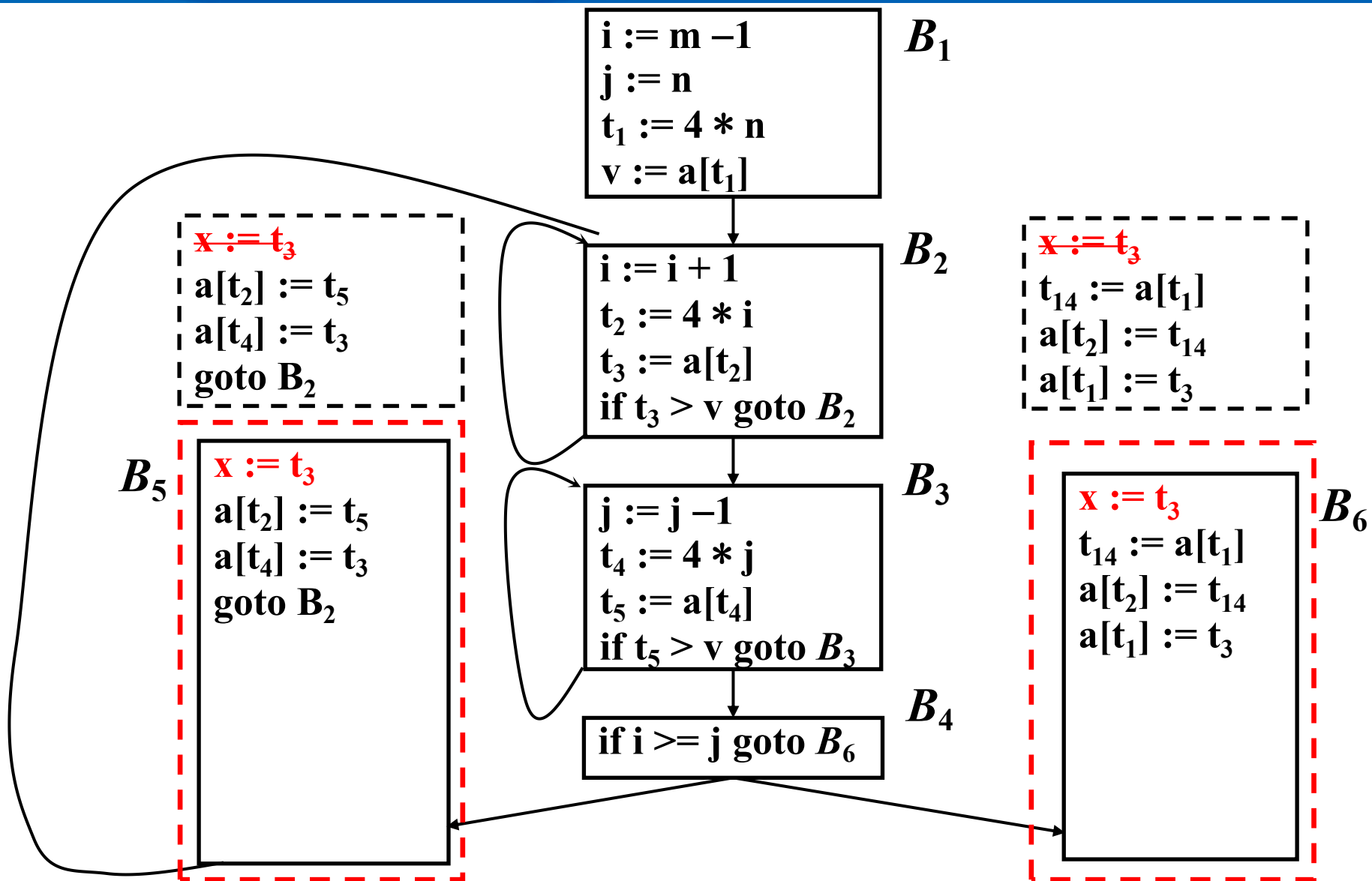
# 快排中的死代码删除





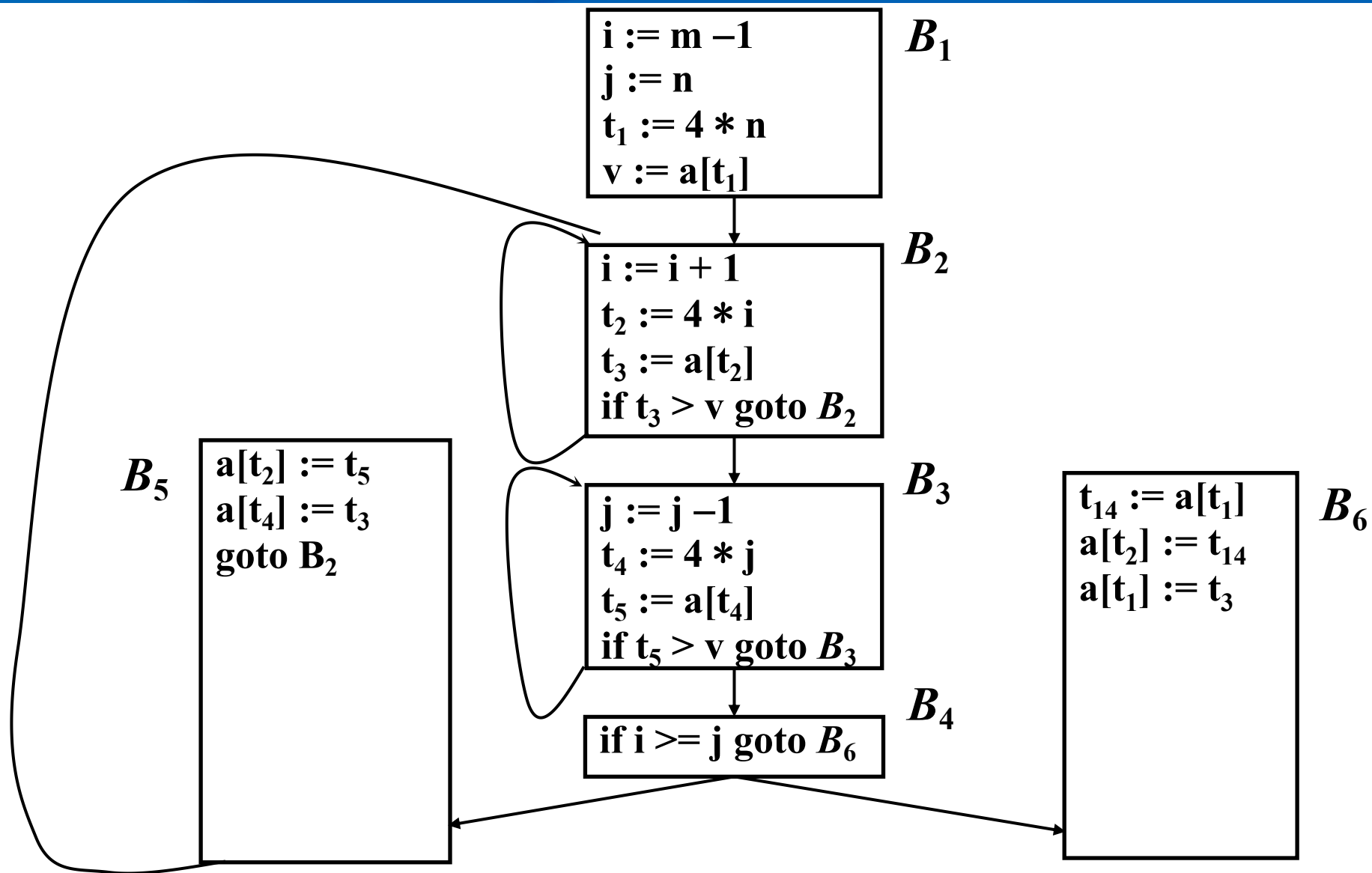


# 快排中的死代码删除





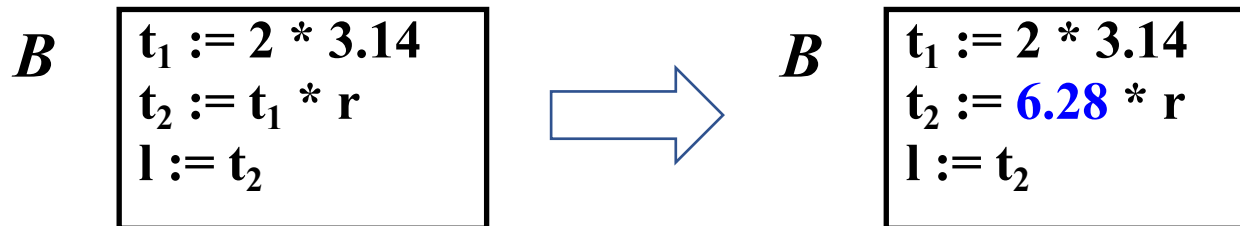
# 快排中的死代码删除





- 如果在**编译时刻**推导出一个表达式的值是常量，就可以使用该常量来代替这个表达式。

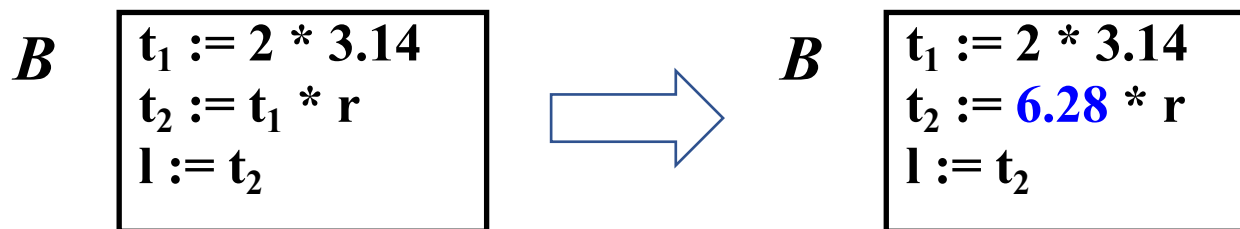
- 例：计算圆周长的表达式  $l = 2 * 3.14 * r$





- 如果在**编译时刻**推导出一个表达式的值是常量，就可以使用该常量来代替这个表达式。

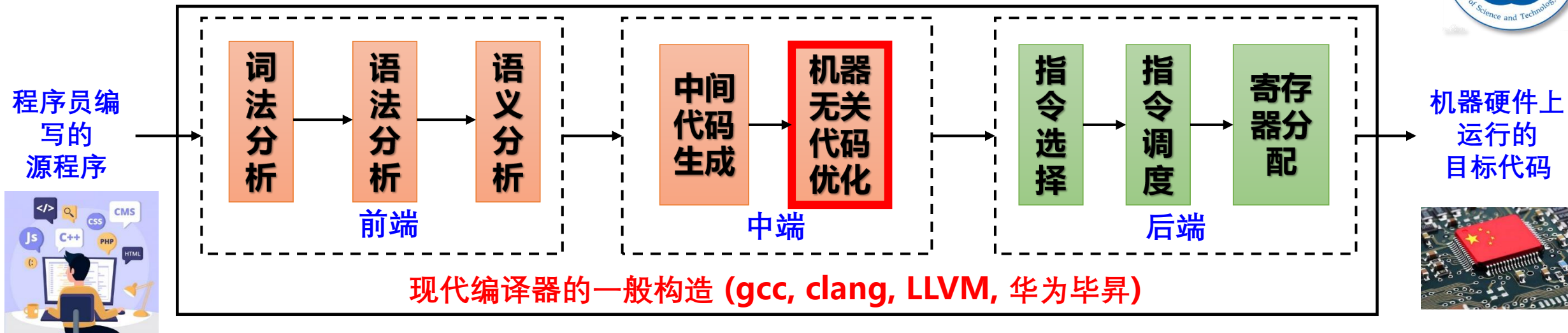
- 例：计算圆周长的表达式  $l = 2 * 3.14 * r$



- 常量合并本身没有优化的意义，**但可以给死代码删除创造机会**



# 本节提纲



## • 代码优化的定义及背景

## • 常见的优化方式

- 公共子表达式删除优化
- 死代码删除、复制传播、常量合并
- 循环系列优化
  - 强度削弱、删除归纳变量、代码移动



- 在循环中的代码会被执行多次
  - 迭代次数越多，执行时间越长
  - 降低每次迭代计算的复杂度是循环优化的重要方法
- 潜在的优化可能——强度削弱 (Strength Reduction)
  - 将程序中执行时间较长的运算替换为执行时间较短的运算

$2 * x$  或者  $2.0 * x$

$x/2$

$x^2$

$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0$



$x + x$

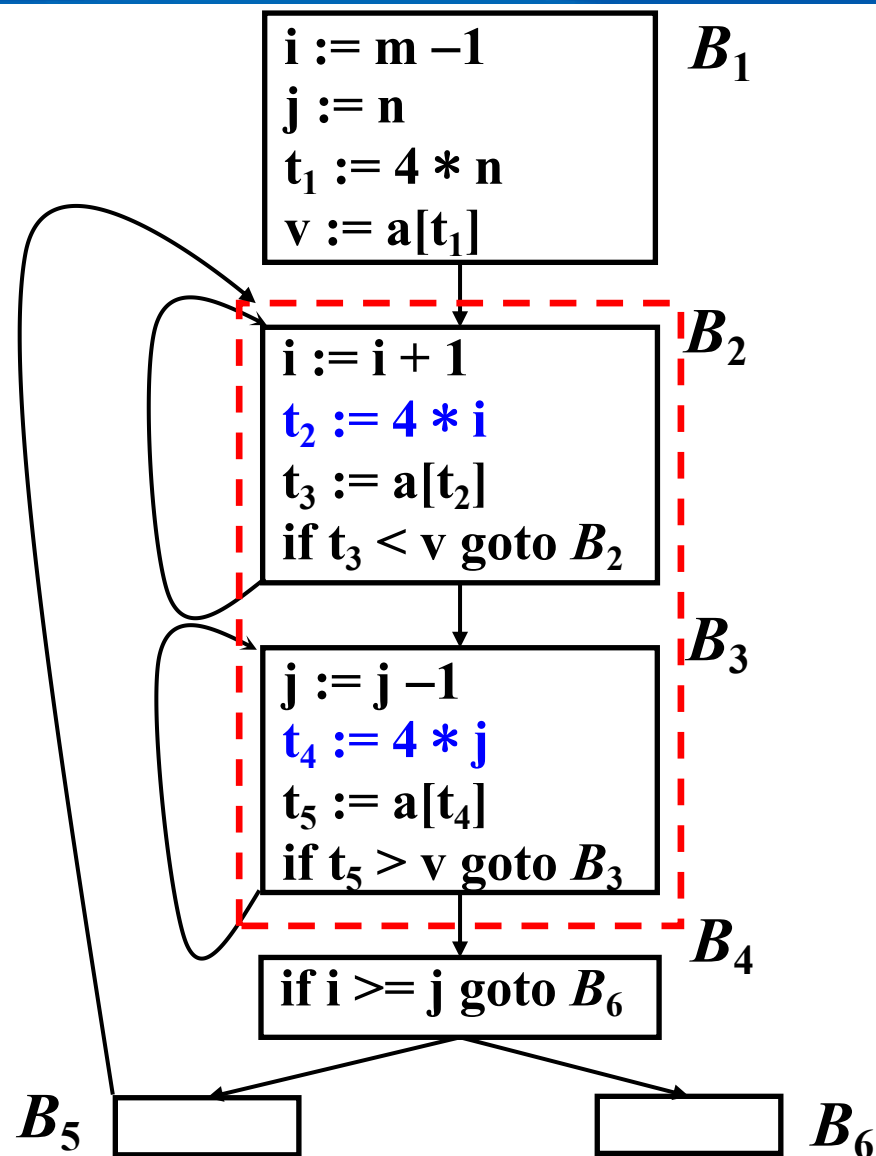
$x * 0.5$

$x * x$

$((\dots (a_n x + a_{n-1})x + a_{n-2}) \dots)x + a_1) + a_0$

开销较高的运算

开销较低的等价运算



## • 观察 $B_2$ 和 $B_3$ 中的变量 $t_2$ 和 $t_4$

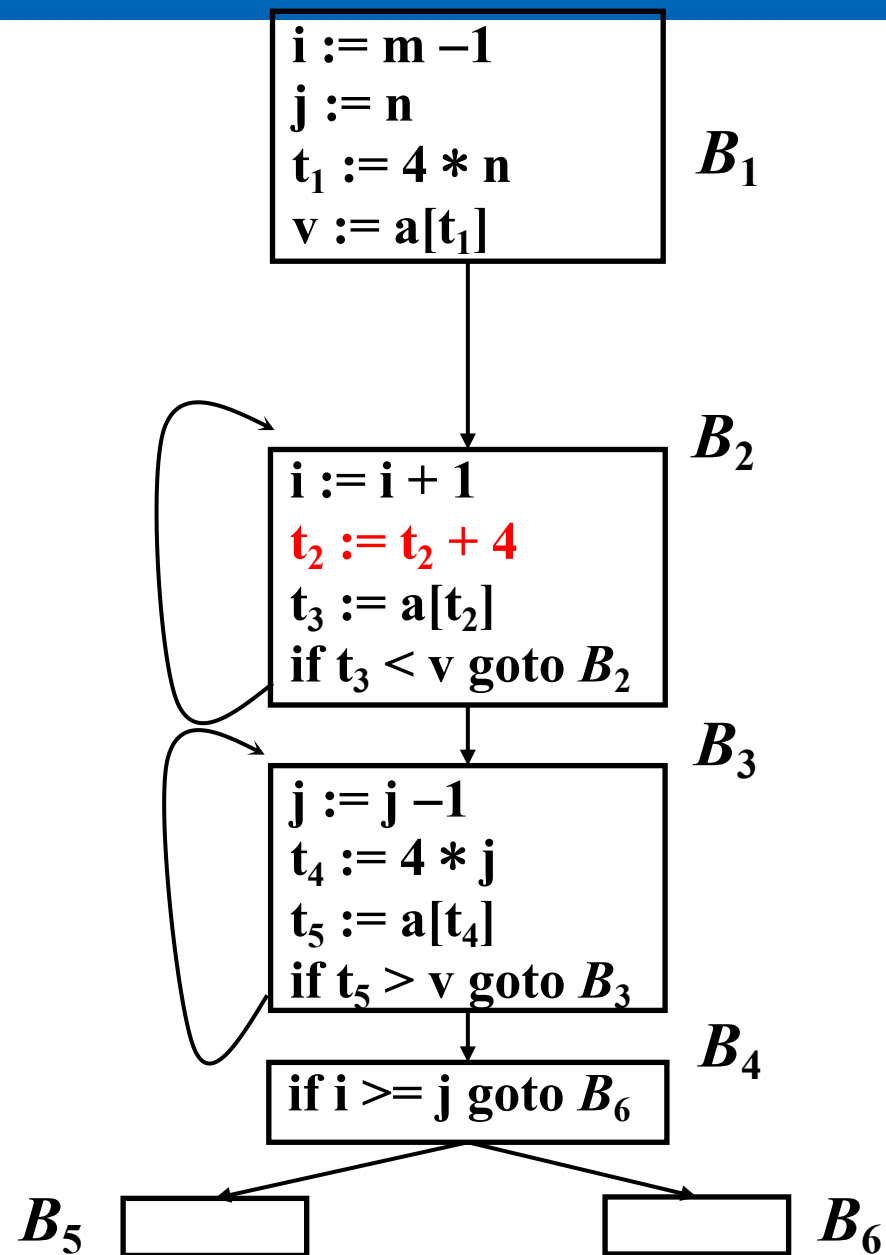
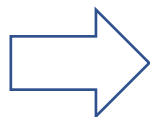
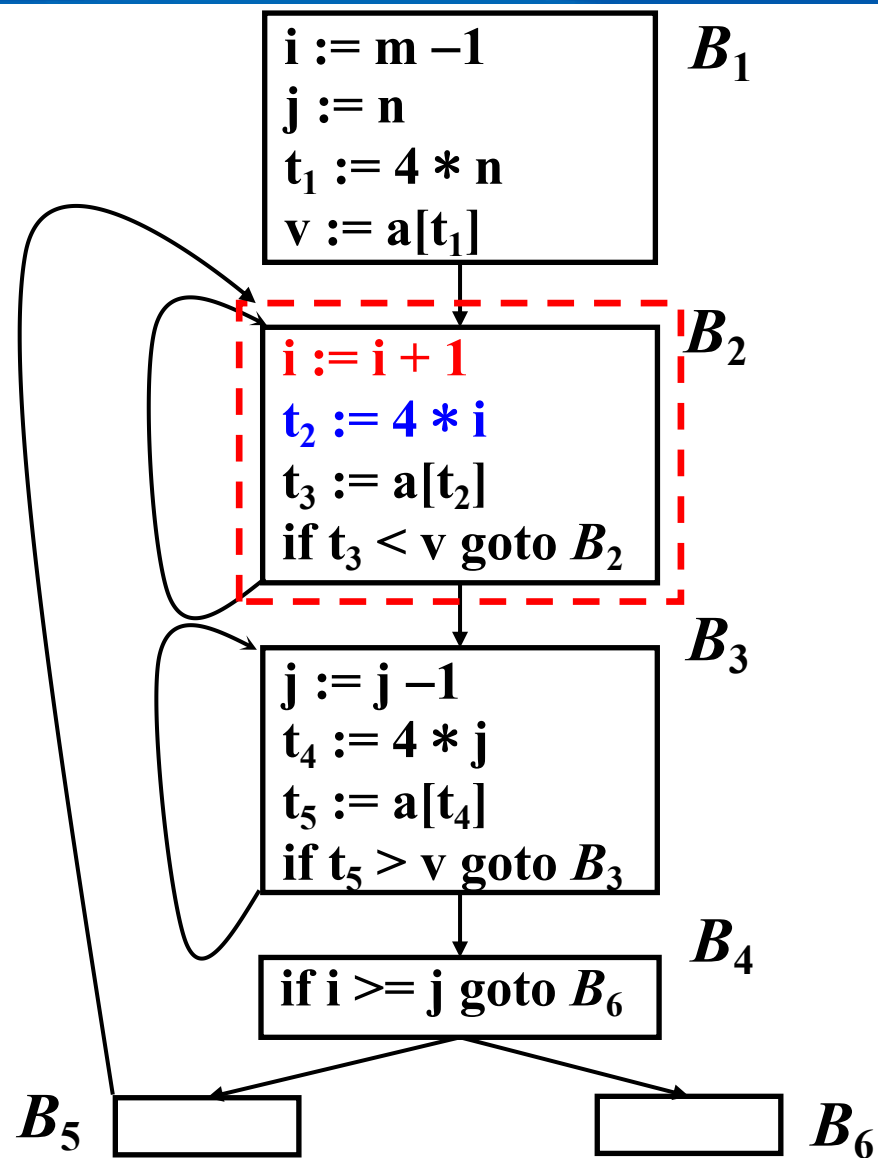
- $t_2$ 和 $t_4$ 总是按照一定的步幅递增或递减

## • 归纳变量

- 如果存在一个常量 $c$ , 使 $x$ 的每一次赋值总是增加 $c$ , 则称 $x$ 为归纳变量。
- 基本形式:  $x = x + c$
- 高级形式:  $x = c * i + d$  ( $c, d$ 常量,  $i$ 归纳变量)
- 强度削弱: **用增量运算(加或减)替代**



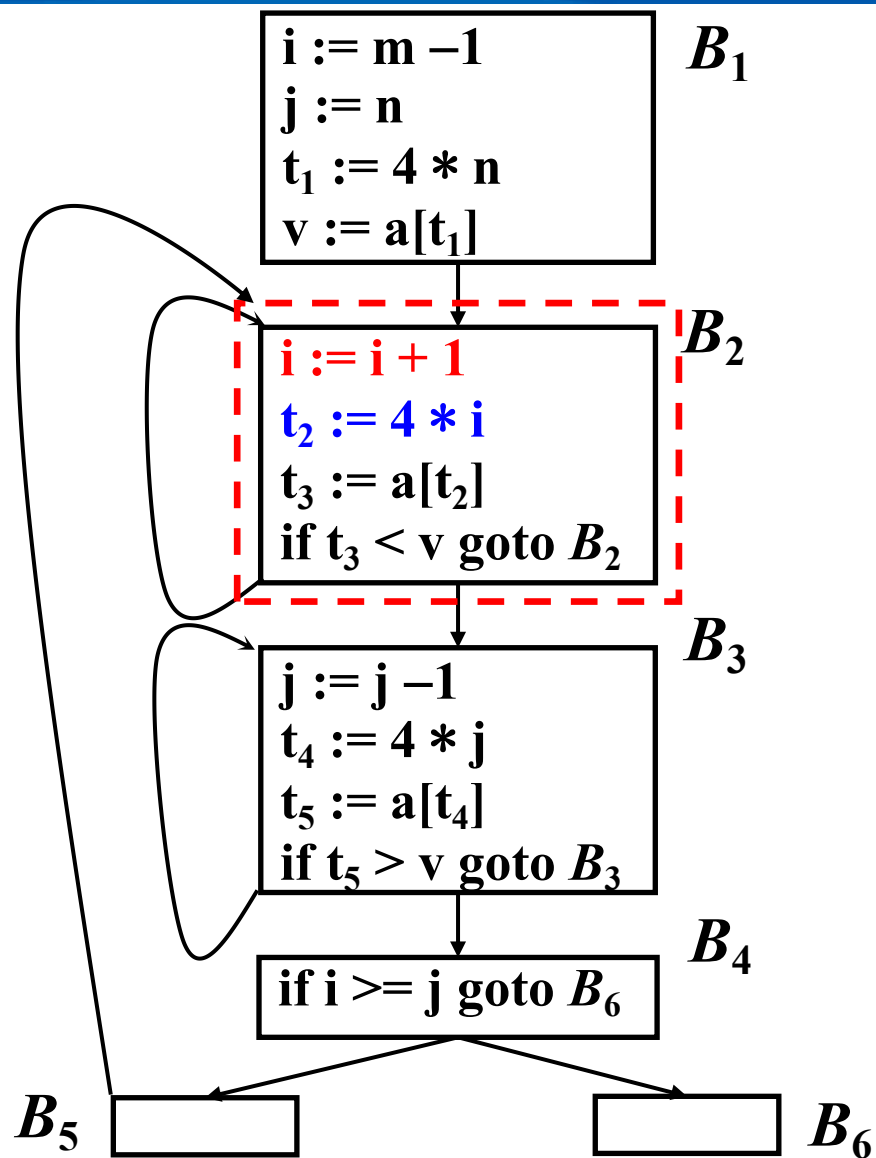
# 循环中的强度削弱



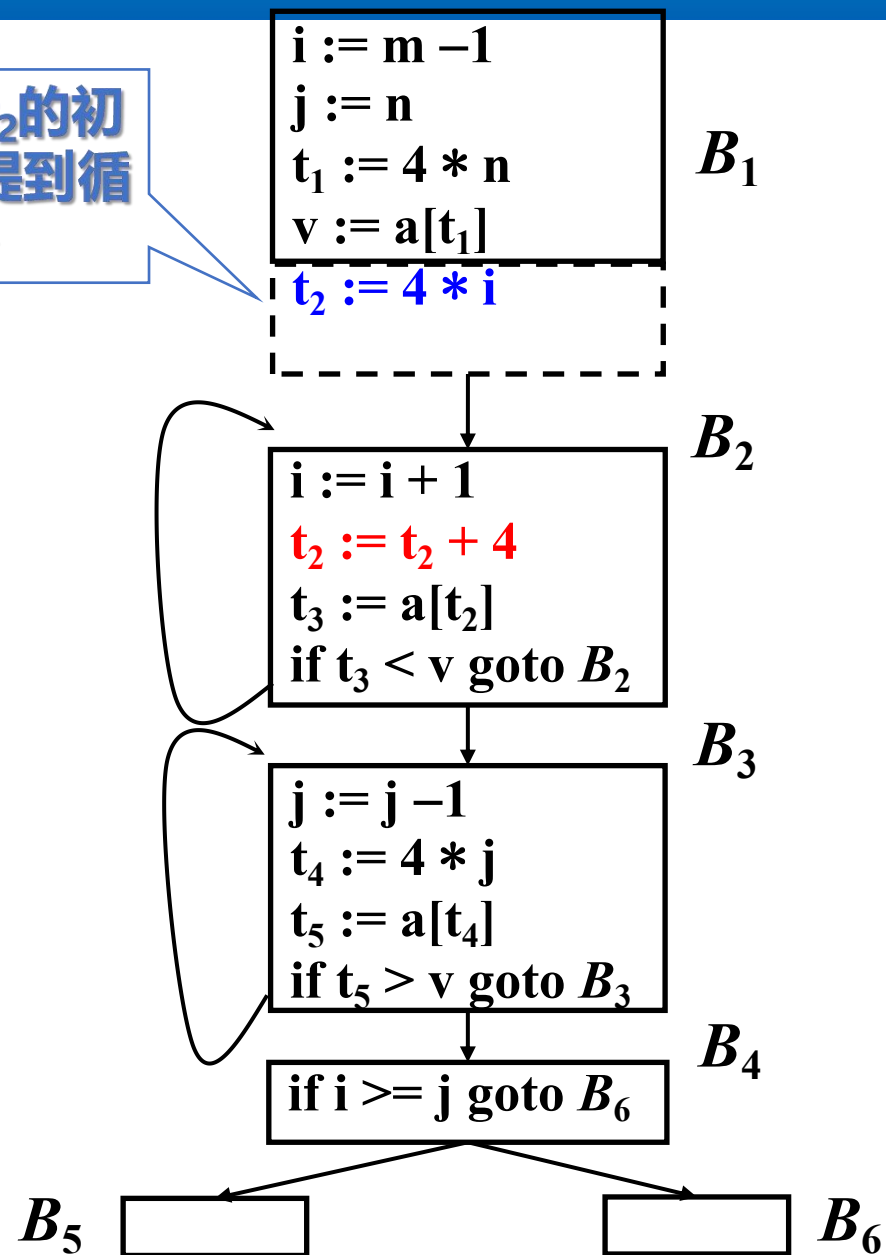




# 循环中的强度削弱

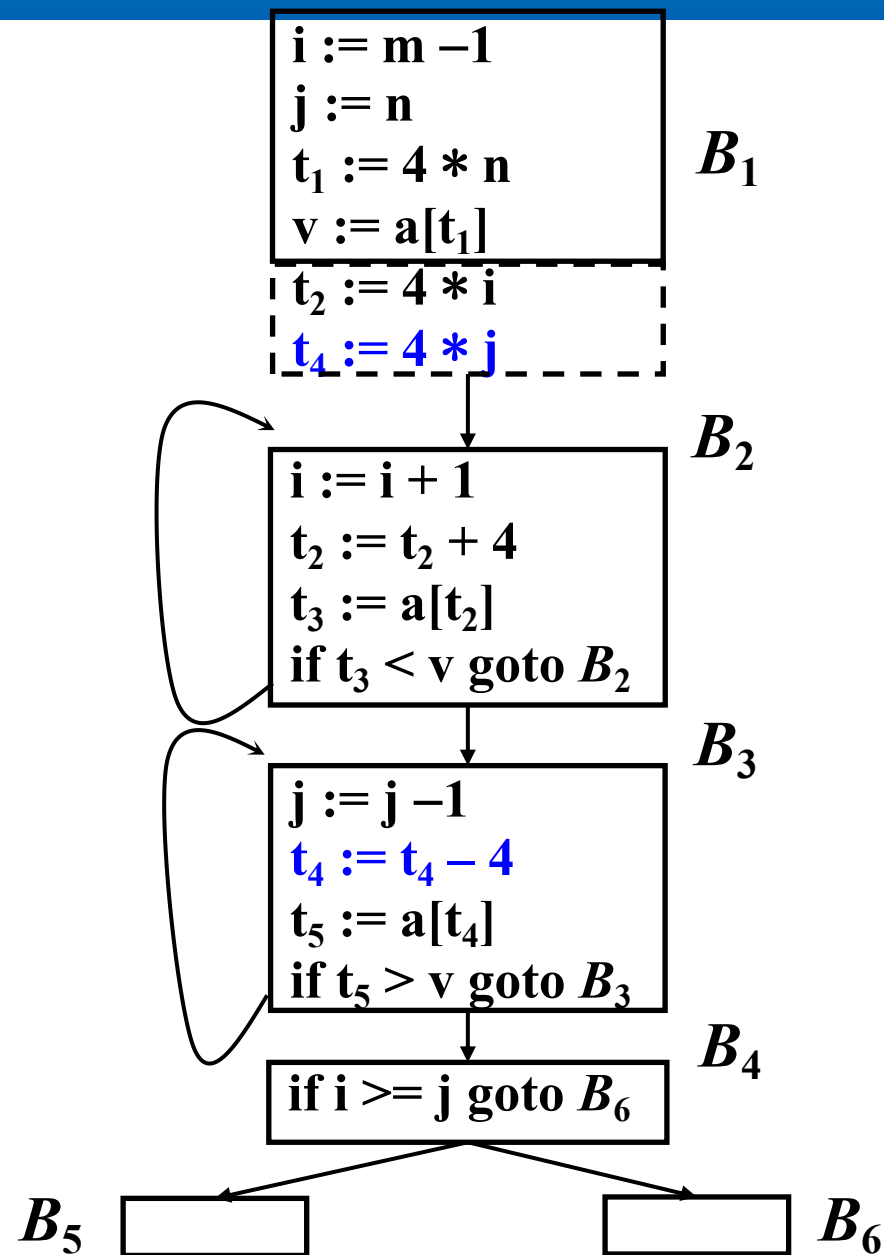
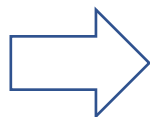
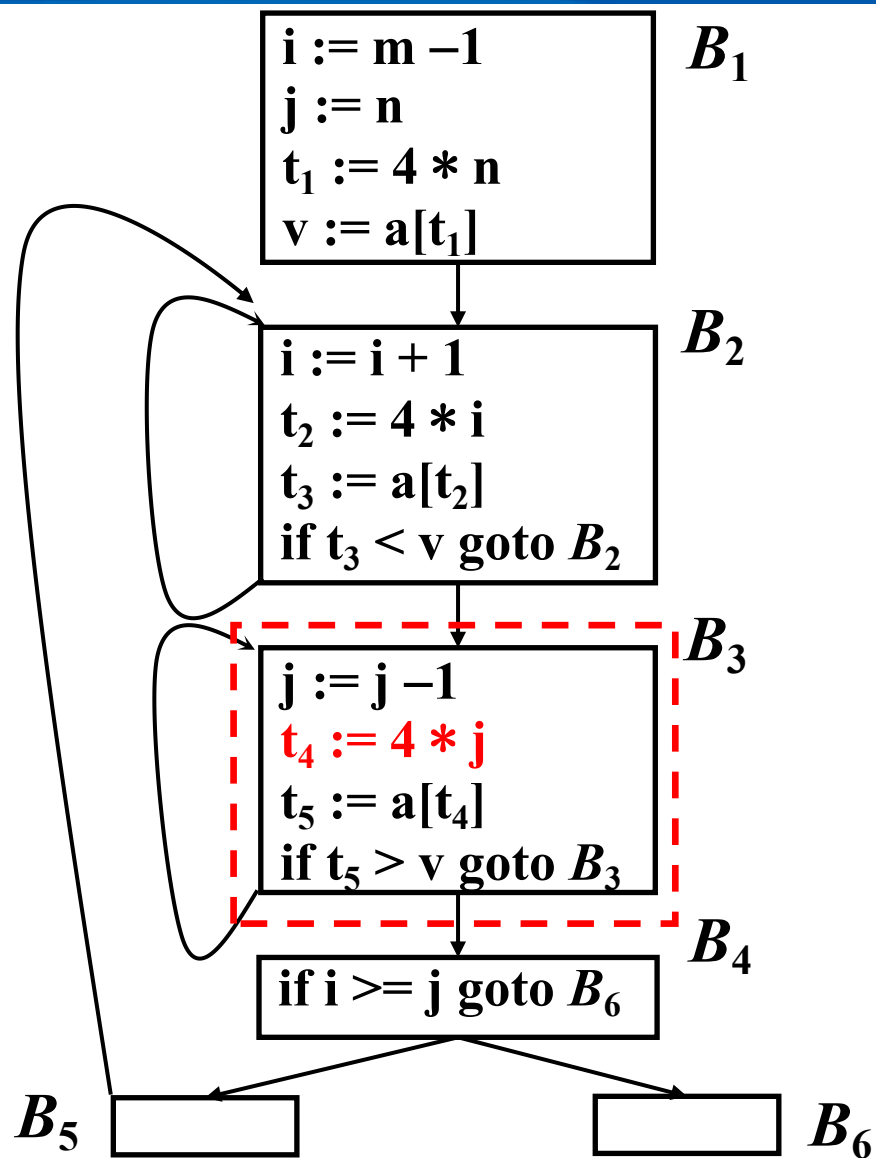


将循环中 $t_2$ 的初始化运算提到循环外





# 循环中的强度削弱

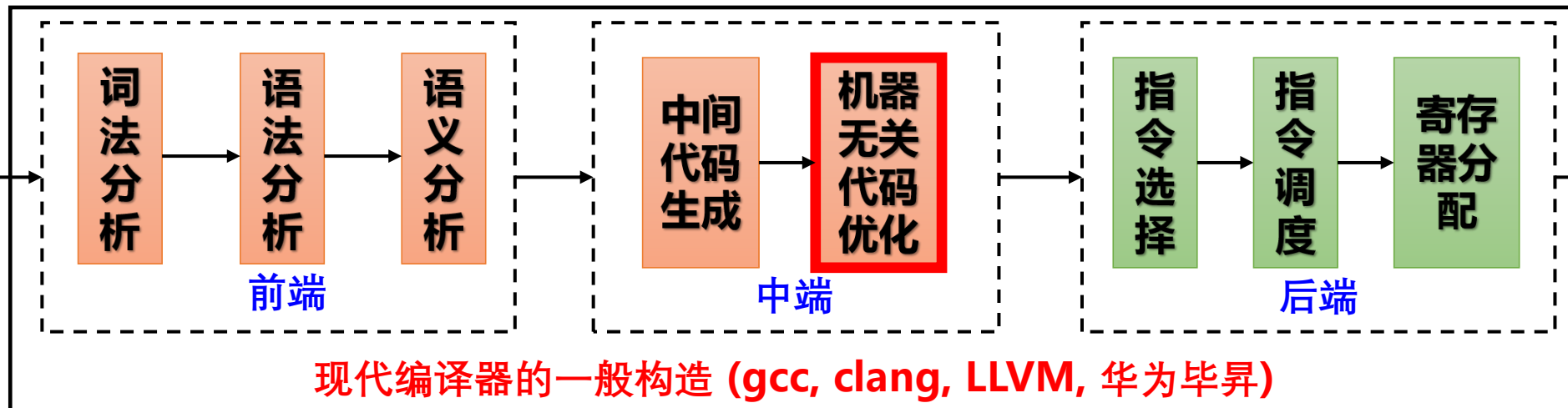




# 本节提纲



程序员编写  
的源程序



机器硬件上  
运行的  
目标代码



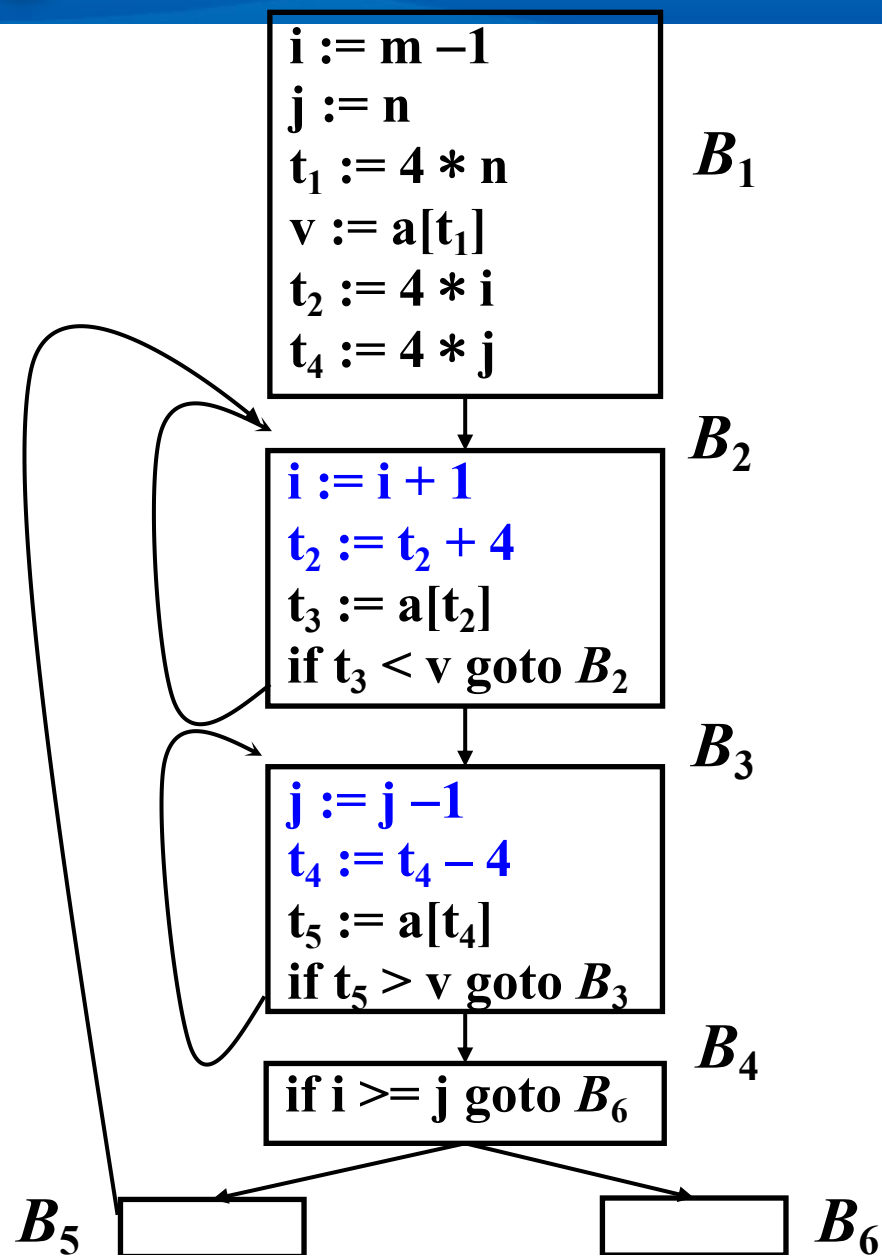
## • 代码优化的定义及背景

## • 常见的优化方式

- 公共子表达式删除优化
- 死代码删除、复制传播、常量合并
- 循环系列优化
  - 强度削弱、删除归纳变量、代码移动



# 循环中的归纳变量删除



□  $i, j, t_2$  与  $t_4$  均为归纳变量

□ 按照变化步调对归纳变量进行分组

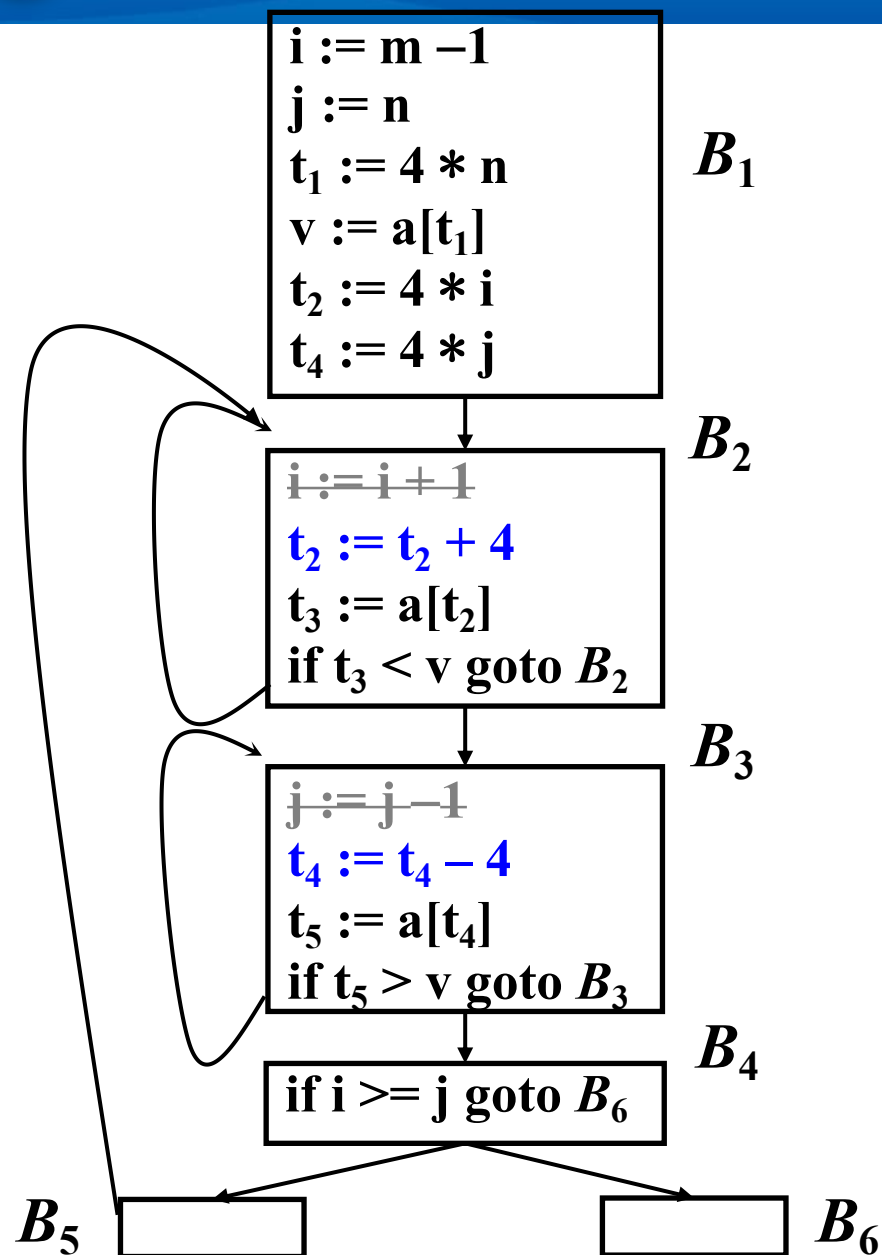
■  $i$  与  $t_2$ ,  $j$  与  $t_4$

□ 删除冗余的归纳变量

■ 一个循环中, 如一组归纳变量的值的变化保持步调一致, 可只保留一个。



# 循环中的归纳变量删除



□  $i, j, t_2$  与  $t_4$  均为归纳变量

□ 按照变化步调对归纳变量进行分组

■  $i$  与  $t_2$ ,  $j$  与  $t_4$

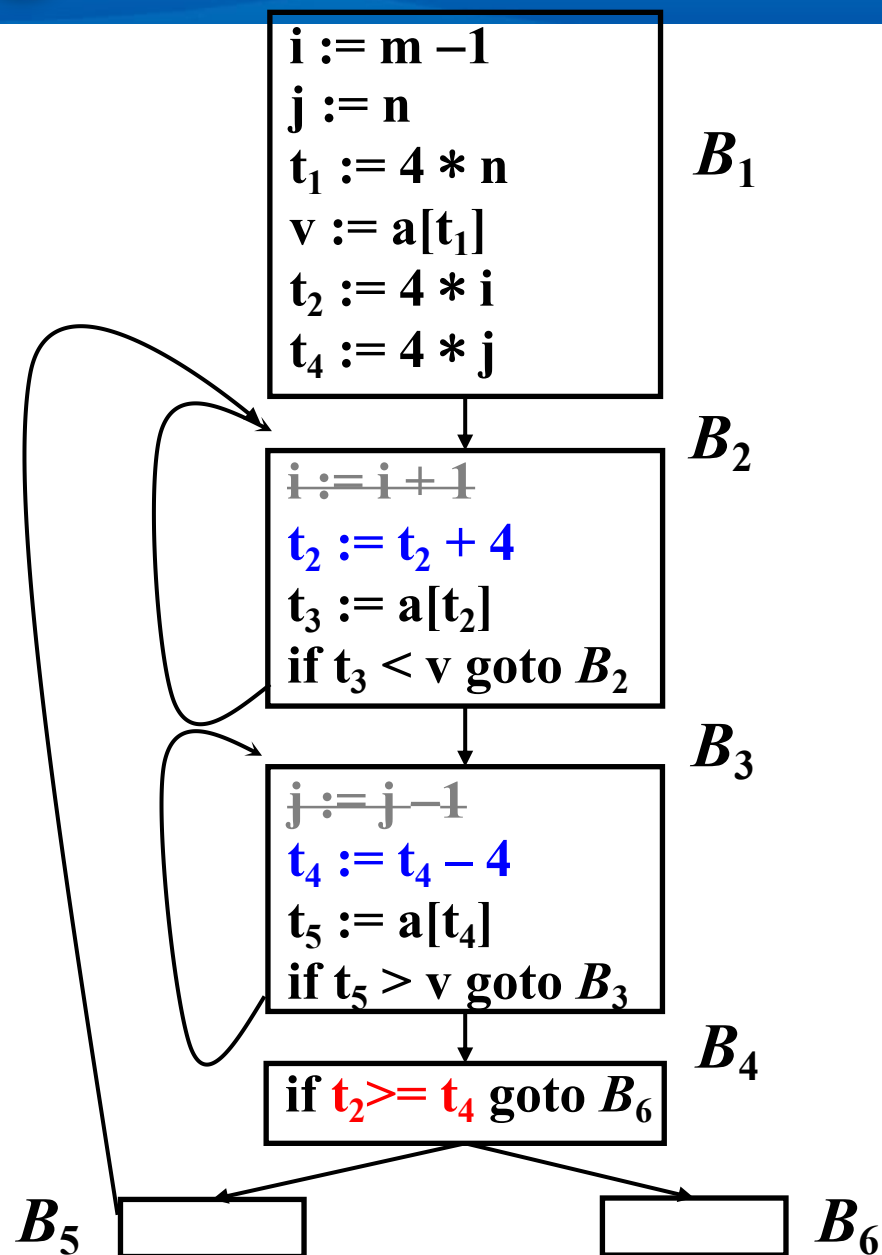
□ 删除冗余的归纳变量

■ 一个循环中, 如一组归纳变量的值的变化保持步调一致, 可只保留一个。

**第一步: 删除循环中对  $i$  和  $j$  的计算**



# 循环中的归纳变量删除



□  $i, j, t_2$  与  $t_4$  均为归纳变量

□ 按照变化步调对归纳变量进行分组

■  $i$  与  $t_2$ ,  $j$  与  $t_4$

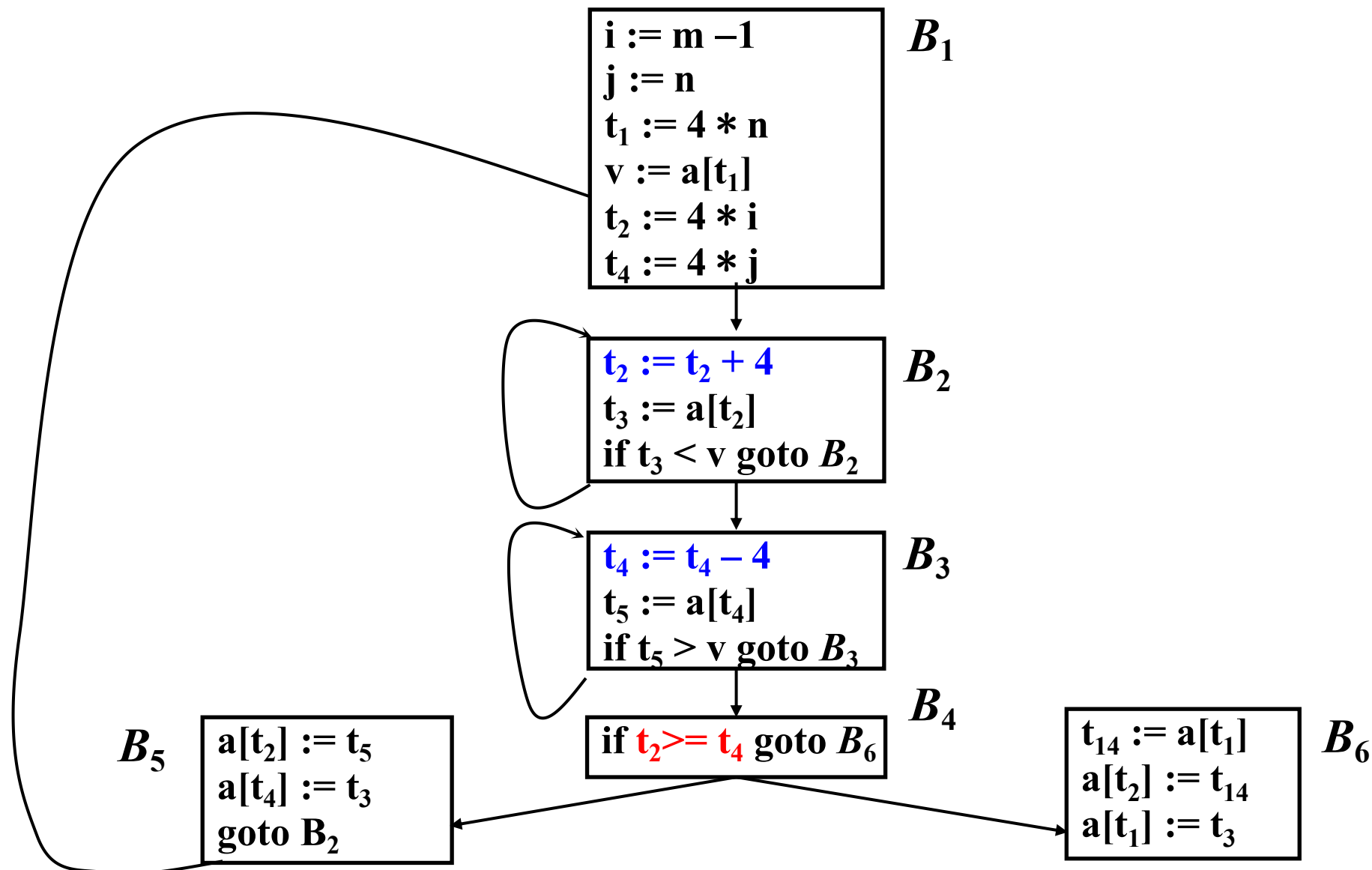
□ 删除冗余的归纳变量

■ 一个循环中，如一组归纳变量的值的变化保持步调一致，可只保留一个。

**第二步：将对  $i$  和  $j$  的引用分别替换为  $t_2$  与  $t_4$**



# 优化后的快速排序代码



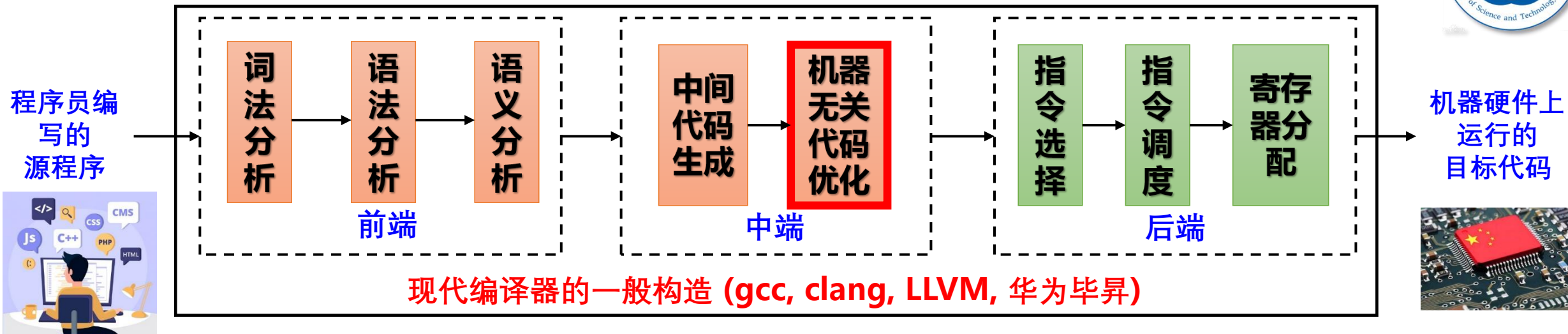
## • 工程实现循环优化的关键理论与技术点

- 如何识别归纳变量？
  - **文档**链接：<https://llvm.org/docs/Passes.html#iv-users-induction-variable-users>
  - **源码**链接：[https://llvm.org/doxygen/IVUsers\\_8cpp\\_source.html](https://llvm.org/doxygen/IVUsers_8cpp_source.html)
- 强度削弱的工业界实现代码
  - **文档**链接：<https://llvm.org/docs/Passes.html#loop-reduce-loop-strength-reduction>
  - **源码**链接：[https://llvm.org/doxygen/LoopStrengthReduce\\_8cpp.html](https://llvm.org/doxygen/LoopStrengthReduce_8cpp.html)
- 归纳变量删除的工业界实现代码
  - **文档**链接：<https://llvm.org/docs/Passes.html#indvars-canonicalize-induction-variables>
  - **源码**链接：[https://llvm.org/doxygen/IndVarSimplify\\_8cpp.html](https://llvm.org/doxygen/IndVarSimplify_8cpp.html)





# 本节提纲



## • 代码优化的定义及背景

## • 常见的优化方式

- 公共子表达式删除优化
- 死代码删除、复制传播、常量合并
- 循环系列优化
  - 强度削弱、删除归纳变量、代码移动



- 循环不变计算 (**loop-invariant computation**) 是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是**循环优化**的一种，在进入循环前就对循环不变计算进行求值



- 循环不变计算 (**loop-invariant computation**) 是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是**循环优化**的一种，在进入循环前就对循环不变计算进行求值。

例： `while (i <= limit - 2 ) ...`

代码移动后变换成

`t = limit - 2;`

`while (i <= t ) ...`



- 循环不变计算 (**loop-invariant computation**) 是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是**循环优化**的一种，在进入循环前就对循环不变计算进行求值。
- 对于多重嵌套循环，**loop-invariant computation**是相对于某一个循环的，可能对于更加外层的循环，它就不成立了。
- 因此，处理循环时，按照由里到外的方式



- **代码优化是编译技术的重要组成部分，是发挥硬件能力、提升编程水平的重要手段。**
- **常见的代码优化方法有循环优化和公共子表达式删除等。**
  - 循环的强度削弱和归纳变量删除依赖于归纳变量识别
  - 公共子表达式删除依赖于公共子表达式的识别
  - 可以看出，大部分的高级优化均需要利用代码分析



# 图灵奖获得者——法兰·艾伦



- 由于在编译优化方面的杰出贡献被授予2006年计算机图灵奖
- 是世界上第一位获得图灵奖的女科学家
- 重要的理论与实践工作有：
  - Program Optimization, 1966
  - Control Flow Analysis, 1970
  - A Basis for Program Optimization, 1970
  - A Catalog of Optimizing Transformations, 1971
  - .....



**Frances Allen**  
**(1932-2020)**  
**ACM/IEEE Fellow**  
**美国科学院院士**



# 一起努力 打造国产基础软硬件体系！

徐伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年4月17日