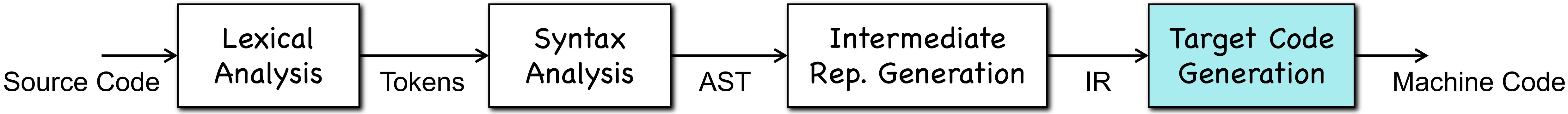
Made by Qingkai Shi
qingkaishi@nju.edu.cn

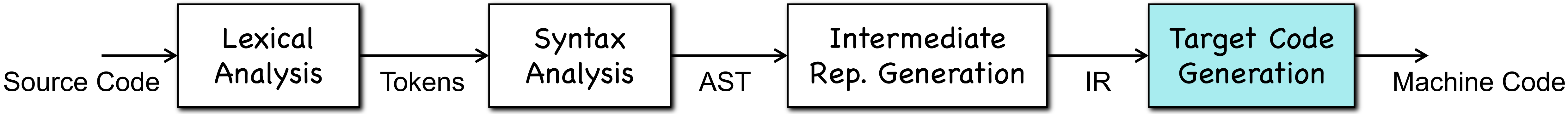


# Chapter 8-3
# Register Allocation

# Register Allocation



# Register Allocation




- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation**: ∞ virtual registers ➞ k physical registers

# PART I: Local Register Allocation

# Register Allocation

- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation**: ∞ virtual registers ➞ k physical registers

- **Requirement:**
  - Produce correct code using k or fewer registers

# Register Allocation

- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation**: ∞ virtual registers ➞ k physical registers

- **Requirement:**
  - Produce correct code using k or fewer registers
  - Minimize loads, stores, and space to hold spilled values

# Register Allocation

- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation**: ∞ virtual registers → k physical registers

- **Requirement:**
  - Produce correct code using k or fewer registers
  - Minimize loads, stores, and space to hold spilled values
  - Efficient register allocation (O(n) or O(nlogn))

# Register Allocation

- Two values CANNOT be mapped to the same register wherever they are both **live** (before their last use)

```
LD   R1, y            // R1 = y
LD   R2, z            // R2 = z
SUB  R1, R1, R2       // R1 = R1 - R2
ST   x, R1            // x = R1
```

**x = y - z**

# Register Allocation

- Two values CANNOT be mapped to the same register wherever they are both **live** (before their last use)

- **Spilling**:
  - saves a value from a register to memory

# Register Allocation

- Two values CANNOT be mapped to the same register wherever they are both **live** (before their last use)

- **Spilling**:
  - saves a value from a register to memory
  - the register is then free for other values

# Register Allocation

- Two values CANNOT be mapped to the same register wherever they are both **live** (before their last use)

- **Spilling**:
  - saves a value from a register to memory
  - the register is then free for other values
  - we can load the spilled value from memory to register when necessary

# Local Register Allocation

- Register allocation in a block

# Local Register Allocation

- Register allocation in a block

- **MAXLIVE**: the maximum of values live at each instruction
  - **MAXLIVE ≤ k** --- Allocation is trivial
  - **MAXLIVE > k** --- Values must be spilled to the memory

# MAXLIVE

```
1. LD    R1    #1028        // R1 <- 1028
2. LD    R2    *R1          // R2 <- contents(R1), assume y
3. MUL   R3    R1    R2     // R3 <- 1028 * y
4. LD    R4    x            // R4 <- x
5. SUB   R5    R4    R2     // R5 <- x−y
6. LD    R6    z            // R6 <- z
7. MUL   R7    R5    R6     // R7 <- z * (x-y)
8. SUB   R8    R7    R3     // R8 <- z * (x-y) – 1028 * y
9. ST    *R1   R8           // contents(R1) <- z * (x-y) – 1028 * y
```

# MAXLIVE

```
1. LD   R1   #1028       // R1 <- 1028
2. LD   R2   *R1         // R2 <- contents(R1), assume y
3. MUL  R3   R1    R2    // R3 <- 1028 * y
4. LD   R4   x           // R4 <- x
5. SUB  R5   R4    R2    // R5 <- x−y
6. LD   R6   z           // R6 <- z
7. MUL  R7   R5    R6    // R7 <- z * (x-y)
8. SUB  R8   R7    R3    // R8 <- z * (x-y) – 1028 * y
9. ST   *R1  R8          // contents(R1) <- z * (x-y) – 1028 * y
```

# MAXLIVE

```
1. LD    R1   #1028        // R1 <- 1028
2. LD    R2   *R1          // R2 <- contents(R1), assume y
3. MUL   R3   R1    R2     // R3 <- 1028 * y
4. LD    R4   x            // R4 <- x
5. SUB   R5   R4    R2     // R5 <- x−y
6. LD    R6   z            // R6 <- z
7. MUL   R7   R5    R6     // R7 <- z * (x-y)
8. SUB   R8   R7    R3     // R8 <- z * (x-y) – 1028 * y
9. ST    *R1  R8           // contents(R1) <- z * (x-y) – 1028 * y
```

# MAXLIVE

```
1. LD    R1   #1028       // R1 <- 1028
2. LD    R2   *R1         // R2 <- contents(R1), assume y
3. MUL   R3   R1    R2    // R3 <- 1028 * y
4. LD    R4   x           // R4 <- x
5. SUB   R5   R4    R2    // R5 <- x−y
6. LD    R6   z           // R6 <- z
7. MUL   R7   R5    R6    // R7 <- z * (x-y)
8. SUB   R8   R7    R3    // R8 <- z * (x-y) – 1028 * y
9. ST    *R1  R8          // contents(R1) <- z * (x-y) – 1028 * y
```

# MAXLIVE

```
1. LD    R1   #1028       // R1 <- 1028
2. LD    R2   *R1         // R2 <- contents(R1), assume y
3. MUL   R3   R1    R2    // R3 <- 1028 * y
4. LD    R4   x           // R4 <- x
5. SUB   R5   R4    R2    // R5 <- x−y
6. LD    R6   z           // R6 <- z
7. MUL   R7   R5    R6    // R7 <- z * (x-y)
8. SUB   R8   R7    R3    // R8 <- z * (x-y) – 1028 * y
9. ST    *R1  R8          // contents(R1) <- z * (x-y) – 1028 * y
```

# MAXLIVE

```
1. LD    R1   #1028      // R1 <- 1028
2. LD    R2   *R1        // R2 <- contents(R1), assume y
3. MUL   R3   R1    R2   // R3 <- 1028 * y
4. LD    R4   x          // R4 <- x
5. SUB   R5   R4    R2   // R5 <- x-y
6. LD    R6   z          // R6 <- z
7. MUL   R7   R5    R6   // R7 <- z * (x-y)
8. SUB   R8   R7    R3   // R8 <- z * (x-y) - 1028 * y
9. ST    *R1  R8         // contents(R1) <- z * (x-y) - 1028 * y
```

# MAXLIVE

```
1. LD    R1    #1028          // R1 <- 1028
2. LD    R2    *R1            // R2 <- contents(R1), assume y
3. MUL   R3    R1     R2      // R3 <- 1028 * y
4. LD    R4    x              // R4 <- x
5. SUB   R5    R4     R2      // R5 <- x-y
6. LD    R6    z              // R6 <- z
7. MUL   R7    R5     R6      // R7 <- z * (x-y)
8. SUB   R8    R7     R3      // R8 <- z * (x-y) - 1028 * y
9. ST    *R1   R8             // contents(R1) <- z * (x-y) - 1028 * y
```

# MAXLIVE

```
1. LD    R1    #1028          // R1 <- 1028
2. LD    R2    *R1            // R2 <- contents(R1), assume y
3. MUL   R3    R1     R2      // R3 <- 1028 * y
4. LD    R4    x              // R4 <- x
5. SUB   R5    R4     R2      // R5 <- x−y
6. LD    R6    z              // R6 <- z
7. MUL   R7    R5     R6      // R7 <- z * (x-y)
8. SUB   R8    R7     R3      // R8 <- z * (x-y) – 1028 * y
9. ST    *R1   R8             // contents(R1) <- z * (x-y) – 1028 * y
```

# MAXLIVE

```
1. LD    R1   #1028       // R1 <- 1028
2. LD    R2   *R1         // R2 <- contents(R1), assume y
3. MUL   R3   R1    R2    // R3 <- 1028 * y
4. LD    R4   x           // R4 <- x
5. SUB   R5   R4    R2    // R5 <- x−y
6. LD    R6   z           // R6 <- z
7. MUL   R7   R5    R6    // R7 <- z * (x-y)
8. SUB   R8   R7    R3    // R8 <- z * (x-y) – 1028 * y
9. ST    *R1  R8          // contents(R1) <- z * (x-y) – 1028 * y
```

# MAXLIVE

```
1. LD     R1    #1028          // R1 <- 1028
2. LD     R2    *R1            // R2 <- contents(R1), assume y
3. MUL    R3    R1     R2      // R3 <- 1028 * y
4. LD     R4    x              // R4 <- x
5. SUB    R5    R4     R2      // R5 <- x-y
6. LD     R6    z              // R6 <- z
7. MUL    R7    R5     R6      // R7 <- z * (x-y)
8. SUB    R8    R7     R3      // R8 <- z * (x-y) - 1028 * y
9. ST     *R1   R8             // contents(R1) <- z * (x-y) - 1028 * y
```

# MAXLIVE

```
1. LD    R1    #1028      // R1
2. LD    R2    *R1        // R1 R2
3. MUL   R3    R1    R2   // R1 R2 R3
4. LD    R4    x          // R1 R2 R3 R4
5. SUB   R5    R4    R2   // R1    R3    R5
6. LD    R6    z          // R1    R3    R5 R6
7. MUL   R7    R5    R6   // R1    R3          R7
8. SUB   R8    R7    R3   // R1                   R8
9. ST    *R1   R8         //
```

# MAXLIVE

```
1. LD     R1     #1028          // R1
2. LD     R2     *R1            // R1 R2
3. MUL    R3     R1      R2     // R1 R2 R3
4. LD     R4     x              // R1 R2 R3 R4
5. SUB    R5     R4      R2     // R1    R3    R5
6. LD     R6     z              // R1    R3    R5 R6
7. MUL    R7     R5      R6     // R1    R3          R7
8. SUB    R8     R7      R3     // R1                   R8
9. ST     *R1    R8             //
```

# MAXLIVE

```
1. LD     R1    #1028        // R1
2. LD     R2    *R1          // R1  R2
3. MUL    R3    R1     R2    // R1  R2  R3
4. LD     R4    x            // R1  R2  R3  R4
5. SUB    R5    R4     R2    // R1      R3      R5
6. LD     R6    z            // R1      R3      R5  R6
7. MUL    R7    R5     R6    // R1      R3              R7
8. SUB    R8    R7     R3    // R1                          R8
9. ST     *R1   R8           //
```

**MAXLIVE = 4**

# MAXLIVE

```
1. LD     R1    #1028         // R1
2. LD     R2    *R1           // R1 R2
3. MUL    R3    R1     R2     // R1 R2 R3
4. LD     R4    x             // R1 R2 R3 R4
5. SUB    R5    R4     R2     // R1    R3    R5
6. LD     R6    z             // R1    R3    R5 R6
7. MUL    R7    R5     R6     // R1    R3          R7
8. SUB    R8    R7     R3     // R1                   R8
9. ST     *R1   R8            //
```

**MAXLIVE = 4**

Enough to have 4 registers

# Local Register Allocation

```
1. LD    R1   #1028      // R1
2. LD    R2   *R1        // R1 R2
3. MUL   R3   R1    R2   // R1 R2 R3
4. LD    R4   x          // R1 R2 R3 R4
5. SUB   R5   R4    R2   // R1    R3    R5
6. LD    R6   z          // R1    R3    R5 R6
7. MUL   R7   R5    R6   // R1    R3          R7
8. SUB   R8   R7    R3   // R1                   R8
9. ST    *R1  R8         //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1    #1028        // R1
2. LD    R2    *R1          // R1 R2
3. MUL   R3    R1    R2     // R1 R2 R3
4. LD    R4    x            // R1 R2 R3 R4
5. SUB   R5    R4    R2     // R1    R3    R5
6. LD    R6    z            // R1    R3    R5 R6
7. MUL   R7    R5    R6     // R1    R3          R7
8. SUB   R8    R7    R3     // R1                   R8
9. ST    *R1   R8           //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1   #1028      // R1
2. LD    R2   *R1        // R1 R2
3. MUL   R3   R1    R2   // R1 R2 R3
4. LD    R4   x          // R1 R2 R3 R4
5. SUB   R2   R4    R2   // R1    R3    R2
6. LD    R6   z          // R1    R3    R2 R6
7. MUL   R7   R2    R6   // R1    R3          R7
8. SUB   R8   R7    R3   // R1                   R8
9. ST    *R1  R8         //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1   #1028         // R1
2. LD    R2   *R1           // R1 R2
3. MUL   R3   R1     R2     // R1 R2 R3
4. LD    R4   x             // R1 R2 R3 R4
5. SUB   R2   R4     R2     // R1    R3    R2
6. LD    R6   z             // R1    R3    R2 R6
7. MUL   R7   R2     R6     // R1    R3          R7
8. SUB   R8   R7     R3     // R1                   R8
9. ST    *R1  R8            //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1    #1028        // R1
2. LD    R2    *R1          // R1 R2
3. MUL   R3    R1    R2     // R1 R2 R3
4. LD    R4    x            // R1 R2 R3 R4
5. SUB   R2    R4    R2     // R1    R3    R2
6. LD    R4    z            // R1    R3    R2 R4
7. MUL   R7    R2    R4     // R1    R3          R7
8. SUB   R8    R7    R3     // R1                   R8
9. ST    *R1   R8           //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1    #1028        // R1
2. LD    R2    *R1          // R1 R2
3. MUL   R3    R1    R2     // R1 R2 R3
4. LD    R4    x            // R1 R2 R3 R4
5. SUB   R2    R4    R2     // R1    R3    R2
6. LD    R4    z            // R1    R3    R2 R4
7. MUL   R7    R2    R4     // R1    R3          R7
8. SUB   R8    R7    R3     // R1                   R8
9. ST    *R1   R8           //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1   #1028       // R1
2. LD    R2   *R1         // R1 R2
3. MUL   R3   R1    R2    // R1 R2 R3
4. LD    R4   x           // R1 R2 R3 R4
5. SUB   R2   R4    R2    // R1    R3    R2
6. LD    R4   z           // R1    R3    R2 R4
7. MUL   R2   R2    R4    // R1    R3          R2
8. SUB   R8   R2    R3    // R1                   R8
9. ST    *R1  R8          //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1   #1028        // R1
2. LD    R2   *R1          // R1 R2
3. MUL   R3   R1    R2     // R1 R2 R3
4. LD    R4   x            // R1 R2 R3 R4
5. SUB   R2   R4    R2     // R1    R3    R2
6. LD    R4   z            // R1    R3    R2 R4
7. MUL   R2   R2    R4     // R1    R3          R2
8. SUB   R8   R2    R3     // R1                   R8
9. ST    *R1  R8           //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1    #1028        // R1
2. LD    R2    *R1          // R1 R2
3. MUL   R3    R1    R2     // R1 R2 R3
4. LD    R4    x            // R1 R2 R3 R4
5. SUB   R2    R4    R2     // R1    R3    R2
6. LD    R4    z            // R1    R3    R2 R4
7. MUL   R2    R2    R4     // R1    R3          R2
8. SUB   R2    R2    R3     // R1                   R2
9. ST    *R1   R2           //
```

**MAXLIVE = 4**

if k ≥ 4, e.g., k = 4

# Local Register Allocation

```
1. LD    R1    #1028        // R1
2. LD    R2    *R1          // R1 R2
3. MUL   R3    R1    R2     // R1 R2 R3
4. LD    R4    x            // R1 R2 R3 R4
5. SUB   R5    R4    R2     // R1    R3    R5
6. LD    R6    z            // R1    R3    R5 R6
7. MUL   R7    R5    R6     // R1    R3          R7
8. SUB   R8    R7    R3     // R1                   R8
9. ST    *R1   R8           //
```

**MAXLIVE = 4**

if k < 4, e.g., k = 3

# Local Register Allocation

```
1. LD     R1    #1028       // R1
2. LD     R2    *R1         // R1 R2
3. MUL    R3    R1    R2    // R1 R2 R3
4. LD     R4    x           // R1 R2 R3 R4
5. SUB    R5    R4    R2    // R1    R3    R5
6. LD     R6    z           // R1    R3    R5 R6
7. MUL    R7    R5    R6    // R1    R3          R7
8. SUB    R8    R7    R3    // R1                   R8
9. ST     *R1   R8          //
```

**MAXLIVE = 4**

if k < 4, e.g., k = 3

# Local Register Allocation

```
1. LD    R1    #1028        // R1
2. LD    R2    *R1          // R1 R2
3. MUL   R3    R1    R2     // R1 R2 R3
4. LD    R4    x            // R1 R2 R3 R4
5. SUB   R5    R4    R2     // R1    R3    R5
6. LD    R6    z            // R1    R3    R5 R6
7. MUL   R7    R5    R6     // R1    R3          R7
8. SUB   R8    R7    R3     // R1                   R8
9. ST    *R1   R8           //
```

**Spill R3 by** ST v R3

**MAXLIVE = 4**

if k < 4, e.g., k = 3

# Local Register Allocation

```
1. LD    R1    #1028       // R1
2. LD    R2    *R1         // R1 R2
3. MUL   R3    R1    R2    // R1 R2 R3
4. LD    R4    x           // R1 R2 R3 R4
5. SUB   R5    R4    R2    // R1    R3    R5
6. LD    R6    z           // R1    R3    R5 R6
7. MUL   R7    R5    R6    // R1    R3          R7
8. SUB   R8    R7    R3    // R1                   R8
9. ST    *R1   R8          //
```

**Spill R3 by** ST v R3

**MAXLIVE = 4**

if k < 4, e.g., k = 3

# Local Register Allocation

```
1. LD   R1   #1028      // R1
2. LD   R2   *R1        // R1 R2
3. MUL  R3   R1   R2    // R1 R2 R3
4. LD   R4   x          // R1 R2    R4
5. SUB  R5   R4   R2    // R1          R5
6. LD   R6   z          // R1          R5 R6
7. MUL  R7   R5   R6    // R1                R7
8. SUB  R8   R7   R3    // R1                   R8
9. ST   *R1  R8         //
```

**Spill R3 by** ST v R3

**MAXLIVE = 4**

if k < 4, e.g., k = 3

# Local Register Allocation

```
1. LD    R1    #1028        // R1
2. LD    R2    *R1          // R1 R2
3. MUL   R3    R1    R2     // R1 R2 R3
4. LD    R4    x            // R1 R2    R4
5. SUB   R5    R4    R2     // R1          R5
6. LD    R6    z            // R1          R5 R6
7. MUL   R7    R5    R6     // R1                R7
8. SUB   R8    R7    R3     // R1                   R8
9. ST    *R1   R8           //
```

**Spill R3 by** ST v R3

**MAXLIVE = 4**

if k < 4, e.g., k = 3

**Reload R3 by** LD R3 v

# Local Register Allocation

```
1. LD    R1   #1028       // R1
2. LD    R2   *R1         // R1 R2
3. MUL   R3   R1    R2    // R1 R2 R3
4. LD    R3   x           // R1 R2    R3
5. SUB   R5   R3    R2    // R1          R5
6. LD    R6   z           // R1          R5 R6
7. MUL   R7   R5    R6    // R1                R7
8. SUB   R8   R7    R3    // R1                   R8
9. ST    *R1  R8          //
```

**Spill R3 by** <u>ST v R3</u>

**MAXLIVE = 4**

if k < 4, e.g., k = 3

**Reload R3 by** <u>LD R3 v</u>

# Why R3? Many Heuristic Methods

```
1. LD    R1   #1028       // R1
2. LD    R2   *R1         // R1 R2
3. MUL   R3   R1    R2    // R1 R2 R3
4. LD    R4   x           // R1 R2 R3 R4
5. SUB   R5   R4    R2    // R1    R3    R5
6. LD    R6   z           // R1    R3    R5 R6
7. MUL   R7   R5    R6    // R1    R3          R7
8. SUB   R8   R7    R3    // R1                   R8
9. ST    *R1  R8          //
```

**Spill R3 by** ST v R3

**MAXLIVE = 4**

if k < 4, e.g., k = 3

# Spill Value, Next Use, Farthest

```
1. LD     R1    #1028         // R1
2. LD     R2    *R1           // R1 R2
3. MUL    R3    R1     R2     // R1 R2 R3
4. LD     R4    x             // R1 R2 R3 R4
5. SUB    R5    R4     R2     // R1    R3    R5
6. LD     R6    z             // R1    R3    R5 R6
7. MUL    R7    R5     R6     // R1    R3          R7
8. SUB    R8    R7     R3     // R1                   R8
9. ST     *R1   R8            //
```

**Spill R1 by** ST v R1

**MAXLIVE = 4**

if k < 4, e.g., k = 3

# PART II: Global Register Allocation

# Global Register Allocation

- Local register allocation does not capture reuse of values across multiple basic blocks

- Global allocation often uses the **graph-coloring** paradigm

# Global Register Allocation

- Local register allocation does not capture reuse of values across multiple basic blocks
- Global allocation often uses the **graph-coloring** paradigm
  - Build a **conflict/interference graph**

# Global Register Allocation

- Local register allocation does not capture reuse of values across multiple basic blocks

- Global allocation often uses the **graph-coloring** paradigm
  - Build a **conflict/interference graph**
  - Find a **k-coloring** for the graph, or change the code to a nearby problem that it can color

# Global Register Allocation

- Local register allocation does not capture reuse of values across multiple basic blocks
- Global allocation often uses the **graph-coloring** paradigm
  - Build a **conflict/interference graph**
  - Find a **k-coloring** for the graph, or change the code to a nearby problem that it can color
  - **NP-complete** under nearly all assumptions, so heuristics are needed

# K-Coloring

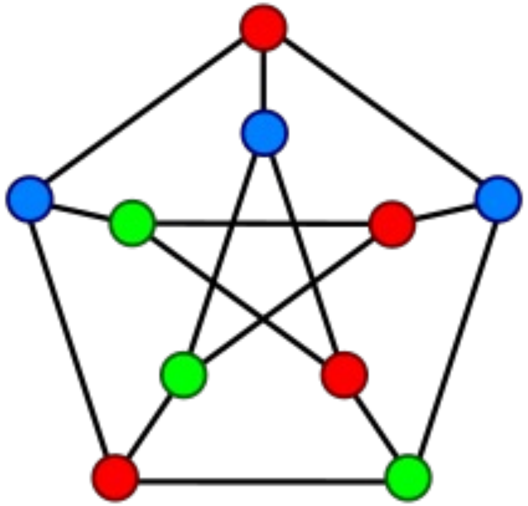
- **Vertex Coloring**: assign a color to each vertex such that no edge connects vertices with the same color.

# K-Coloring

- **Vertex Coloring**: assign a color to each vertex such that no edge connects vertices with the same color.
- **K-Coloring**: a coloring using at most k colors

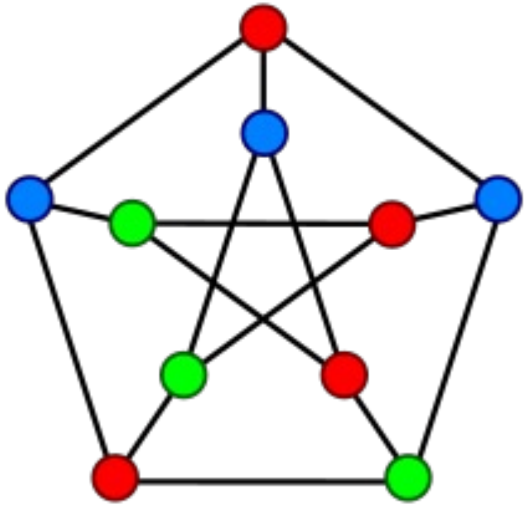
# K-Coloring

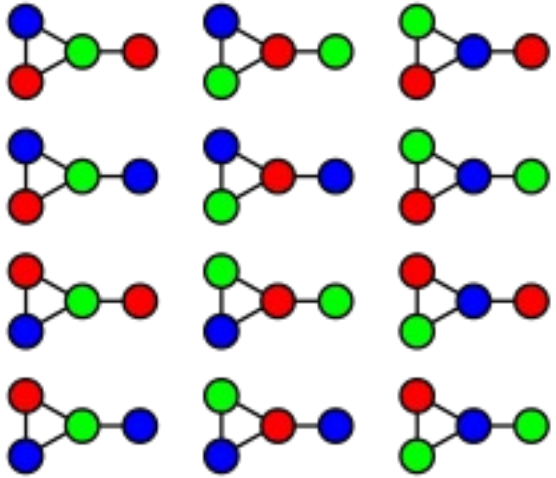
- **Vertex Coloring**: assign a color to each vertex such that no edge connects vertices with the same color.
- **K-Coloring**: a coloring using at most k colors



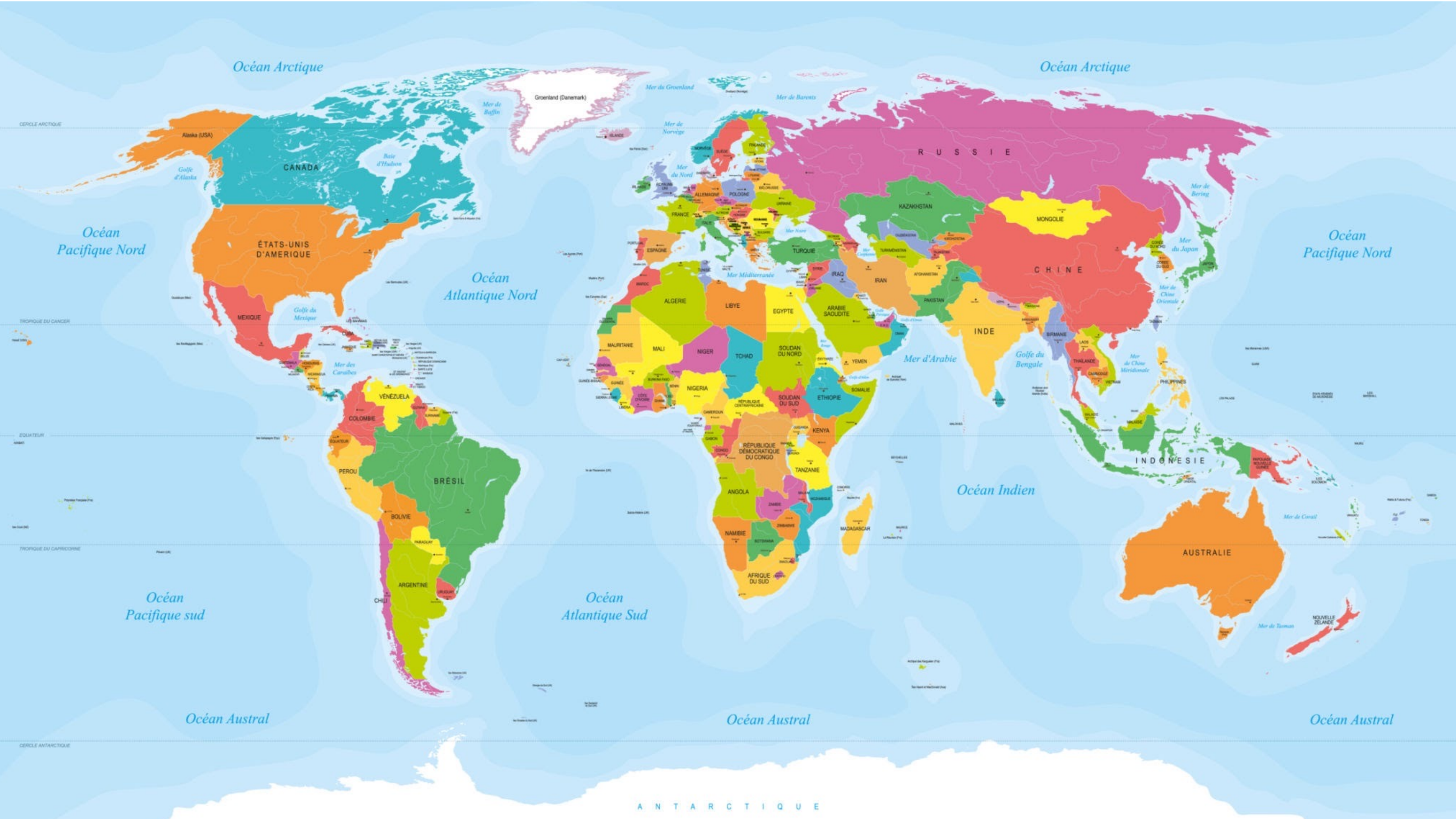
# K-Coloring

- **Vertex Coloring**: assign a color to each vertex such that no edge connects vertices with the same color.
- **K-Coloring**: a coloring using at most k colors





**3-coloring in 12 ways**



Océan Arctique
Océan Arctique
Groenland (Danemark)
Mer du Groenland
Mer de Barents
Mer de Baffin
Mer de Norvège
CERCLE ARCTIQUE
Alaska (USA)
CANADA
Baie d'Hudson
Golfe d'Alaska
ISLANDE
Mer du Nord
NORVÈGE
SUÈDE
FINLANDE
IRLANDE
ALLEMAGNE
POLOGNE
BÉLORUSSIE
UKRAINE
FRANCE
ITALIE
ESPAGNE
PORTUGAL
R U S S I E
Mer de Bering
KAZAKHSTAN
MONGOLIE
Mer du Japon
JAPON
Océan Pacifique Nord
Océan Pacifique Nord
ÉTATS-UNIS D'AMERIQUE
Océan Atlantique Nord
TURQUIE
Mer Noire
Mer Méditerranée
IRAQ
IRAN
AFGHANISTAN
PAKISTAN
C H I N E
Mer de Chine Orientale
MAROC
ALGERIE
LIBYE
EGYPTE
ARABIE SAOUDITE
MEXIQUE
Golfe du Mexique
TROPIQUE DU CANCER
CUBA
INDE
Mer d'Arabie
Golfe du Bengale
BIRMANIE
THAILANDE
Mer de Chine Méridionale
PHILIPPINES
MAURITANIE
MALI
NIGER
TCHAD
SOUDAN DU NORD
YEMEN
Mer des Caraïbes
SENEGAL
NIGERIA
ETHIOPIE
SOMALIE
VÉNÉZUELA
COLOMBIE
SOUDAN DU SUD
RÉPUBLIQUE CENTRAFRICAINE
CAMEROUN
EQUATEUR
KENYA
RÉPUBLIQUE DÉMOCRATIQUE DU CONGO
TANZANIE
INDONESIE
PEROU
BRÉSIL
ANGOLA
Océan Indien
ZAMBIE
MOZAMBIQUE
MADAGASCAR
BOLIVIE
NAMIBIE
BOTSWANA
ZIMBABWE
Mer de Corail
PARAGUAY
TROPIQUE DU CAPRICORNE
AUSTRALIE
ARGENTINE
AFRIQUE DU SUD
CHILI
URUGUAY
Océan Pacifique sud
Océan Atlantique Sud
NOUVELLE ZÉLANDE
Mer de Tasman
Océan Austral
Océan Austral
Océan Austral
CERCLE ANTARCTIQUE
A N T A R C T I Q U E

# K-Coloring for Register Allocation

- Map graph vertices onto virtual registers
- Map colors onto physical registers

# K-Coloring for Register Allocation

- **Map graph vertices onto virtual registers**
- **Map colors onto physical registers**

- From live ranges construct a **conflict graph**

# K-Coloring for Register Allocation

- **Map graph vertices onto virtual registers**
- **Map colors onto physical registers**

- From live ranges construct a **conflict graph**
- Color the graph so that no two neighbors have the same color

# K-Coloring for Register Allocation

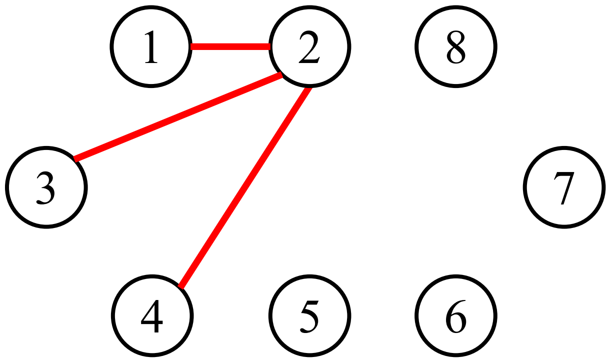
- **Map graph vertices onto virtual registers**
- **Map colors onto physical registers**

- From live ranges construct a **conflict graph**
- Color the graph so that no two neighbors have the same color
- If graph needs more than $k$ colors – Spilling

# Conflict Graph

- **Vertex**: virtual registers;
- **Edges**: virtual registers that have overlapping live range

```
1. LD     R1    #1028         // R1
2. LD     R2    *R1           // R1 R2
3. MUL    R3    R1     R2     // R1 R2 R3
4. LD     R4    x             // R1 R2 R3 R4
5. SUB    R5    R4     R2     // R1    R3    R5
6. LD     R6    z             // R1    R3    R5 R6
7. MUL    R7    R5     R6     // R1    R3          R7
8. SUB    R8    R7     R3     // R1                   R8
9. ST     *R1   R8            //
```

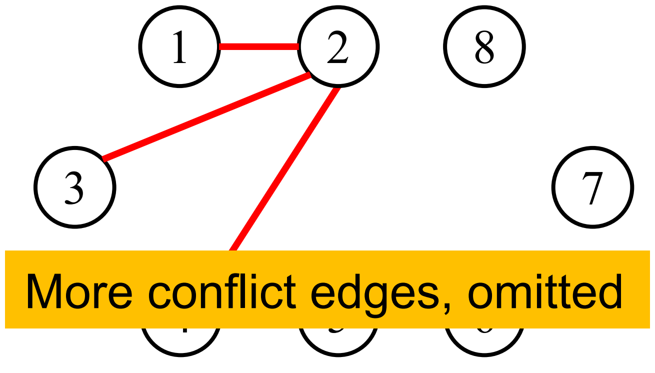



Conflict Graph

# Conflict Graph

- **Vertex**: virtual registers;
- **Edges**: virtual registers that have overlapping live range

```
1. LD     R1    #1028         // R1
2. LD     R2    *R1           // R1 R2
3. MUL    R3    R1     R2     // R1 R2 R3
4. LD     R4    x             // R1 R2 R3 R4
5. SUB    R5    R4     R2     // R1    R3    R5
6. LD     R6    z             // R1    R3    R5 R6
7. MUL    R7    R5     R6     // R1    R3          R7
8. SUB    R8    R7     R3     // R1                   R8
9. ST     *R1   R8            //
```

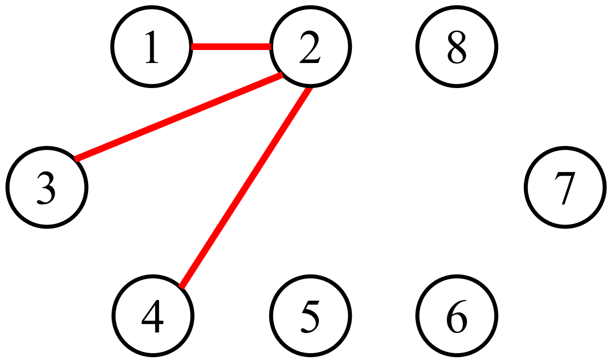



Conflict Graph

# Conflict Graph

- **Vertex**: virtual registers;
- **Edges**: virtual registers that have overlapping live range

If we can use k, e.g., 4, colors, to color the graph, the 8 virtual registers can be replaced by 4 physical registers.



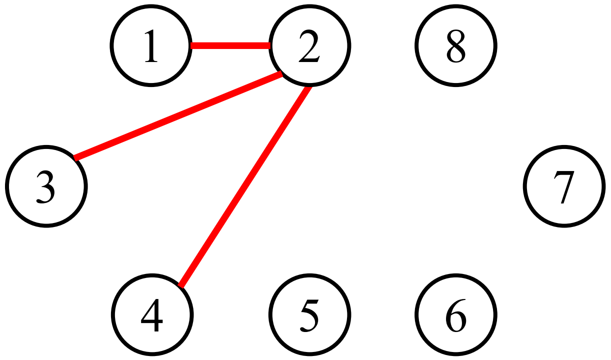

Conflict Graph

# Conflict Graph

- **Vertex**: virtual registers;
- **Edges**: virtual registers that have overlapping live range

If we can use k, e.g., 4, colors, to color the graph, the 8 virtual registers can be replaced by 4 physical registers.

If we have to use 5 colors to color the graph, but only k = 4 physical registers are available, spilling is necessary.




Conflict Graph

# Coloring the Conflict Graph

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable

# Coloring the Conflict Graph

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable

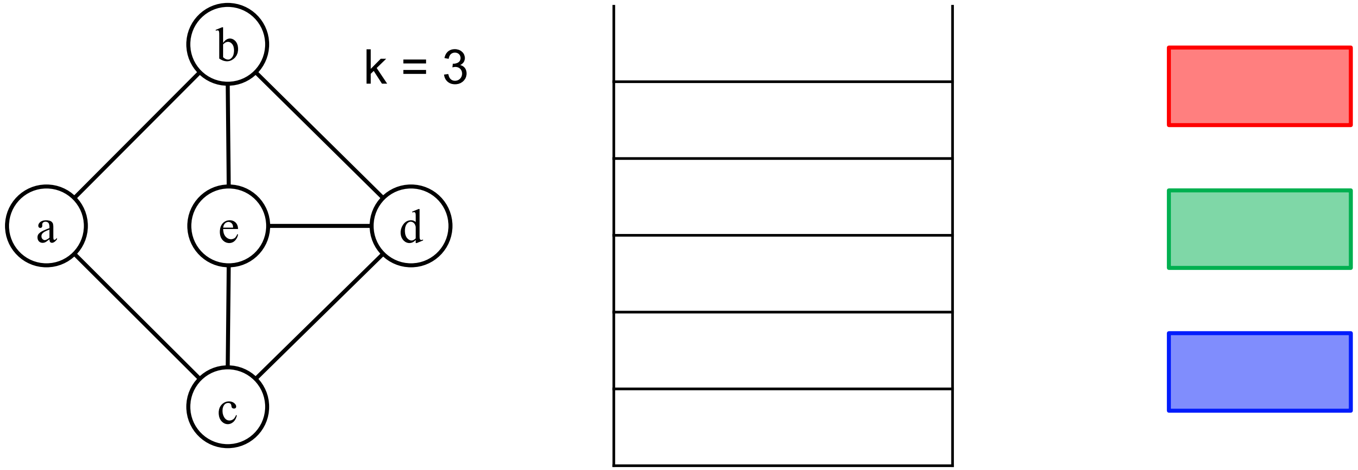
- Chaitin's Algorithm
- ACM SIGPLAN Symposium on Compiler Construction (1982)

# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)

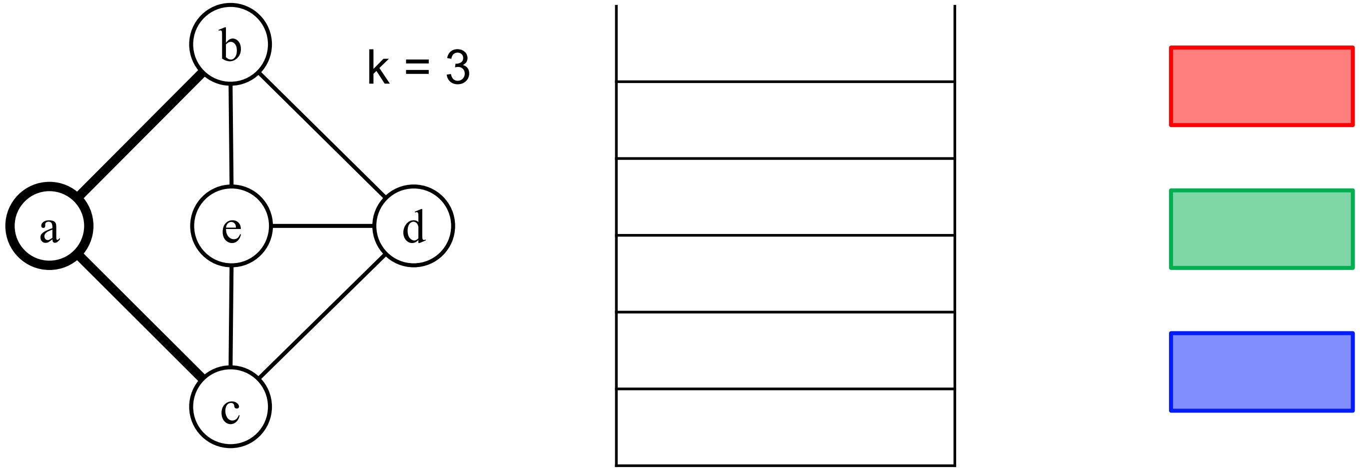
# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



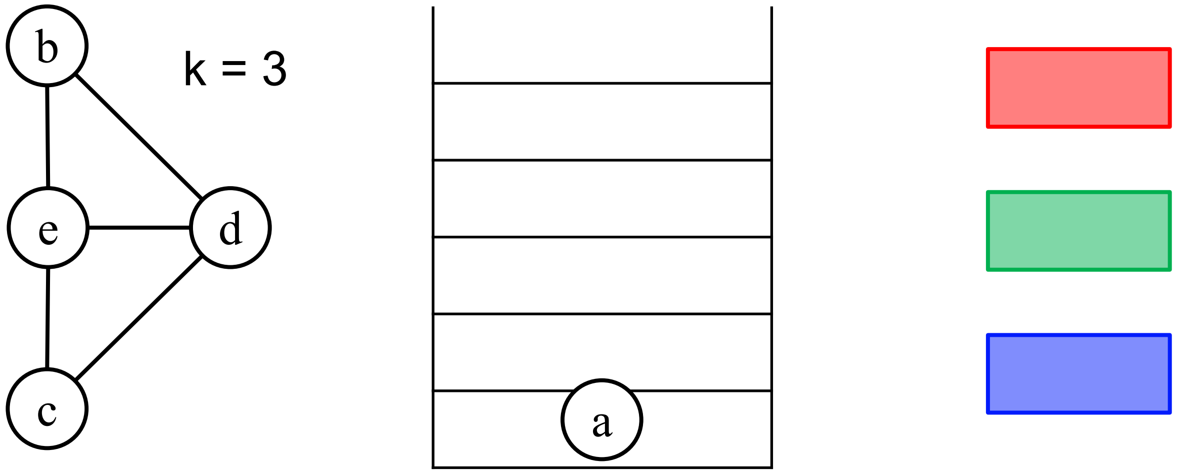
# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



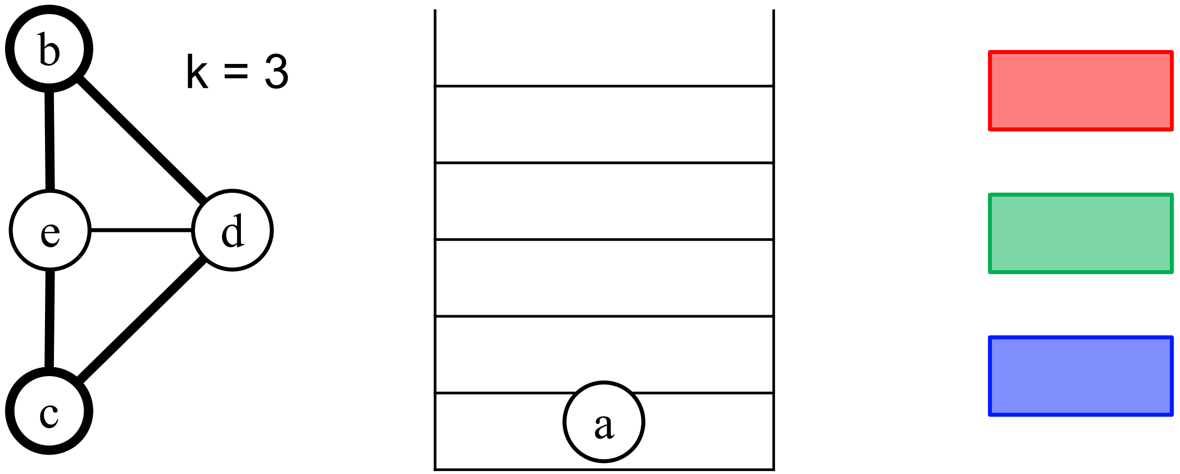
# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



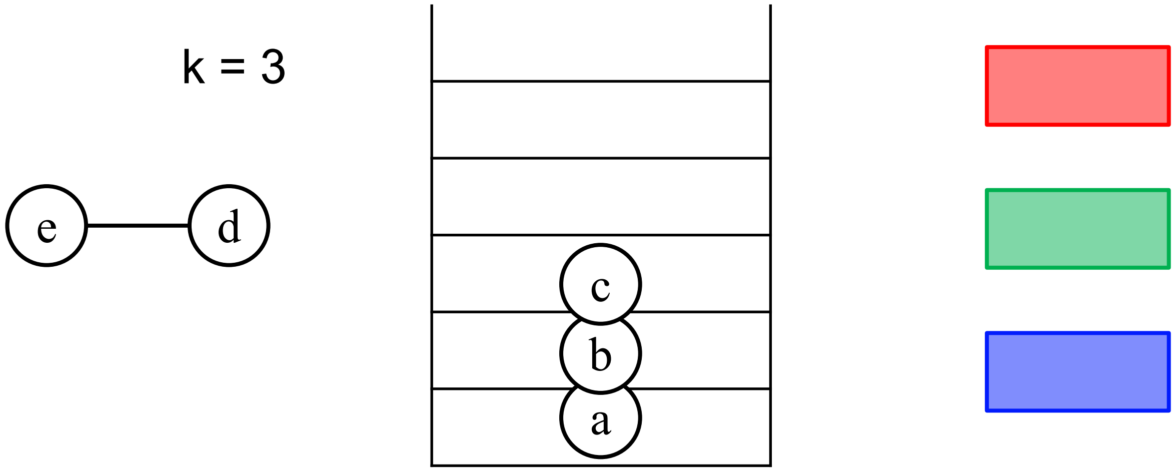
# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



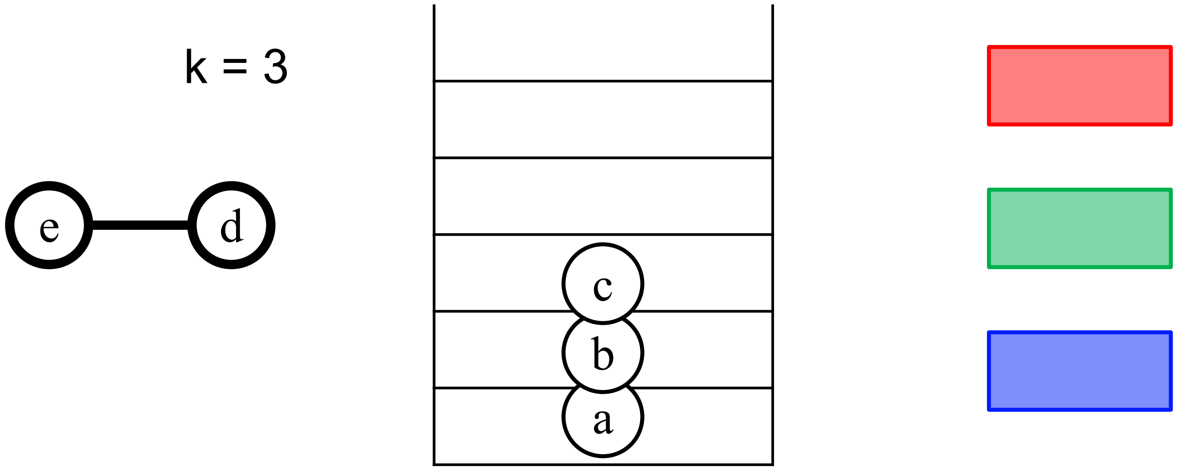
# Chaitin’s Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



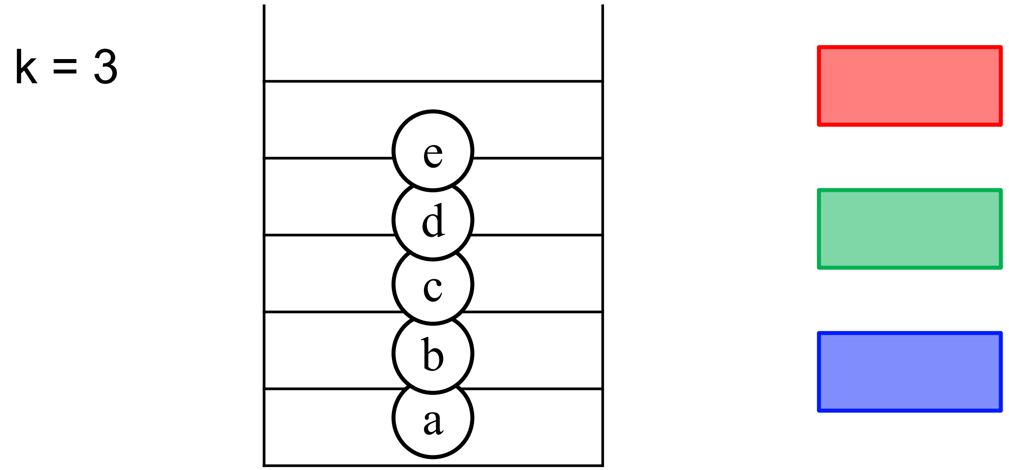
# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



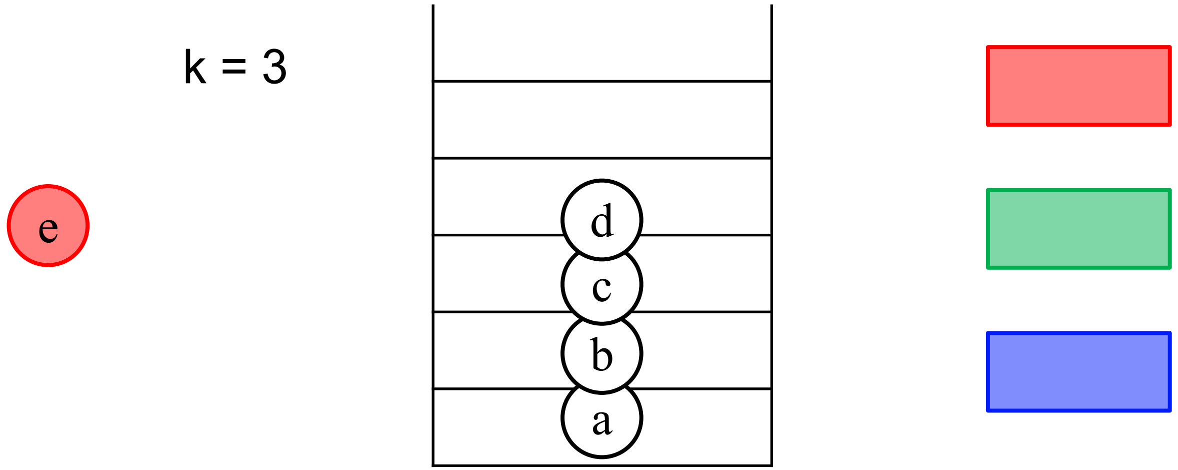
# Chaitin’s Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



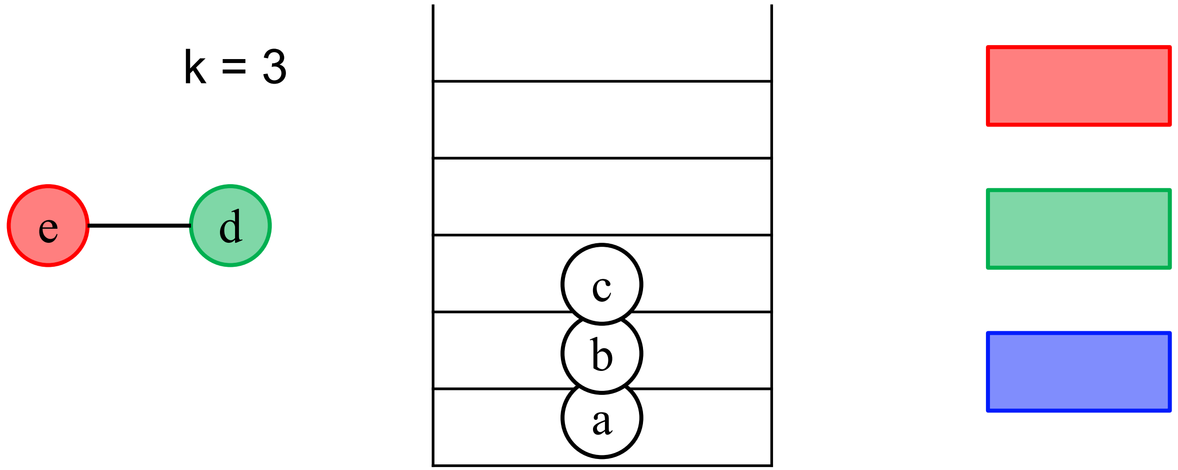
# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



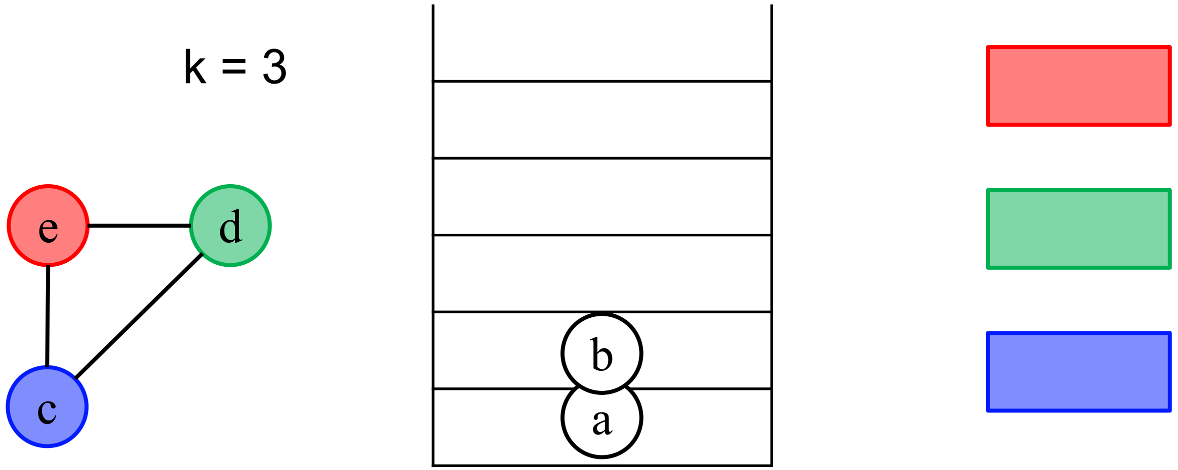
# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



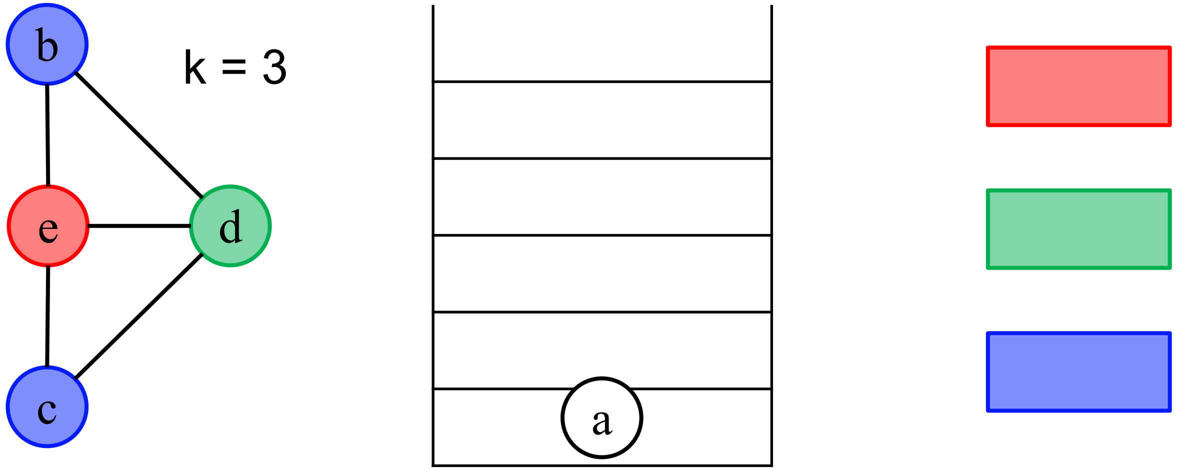
# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



# Chaitin's Algorithm

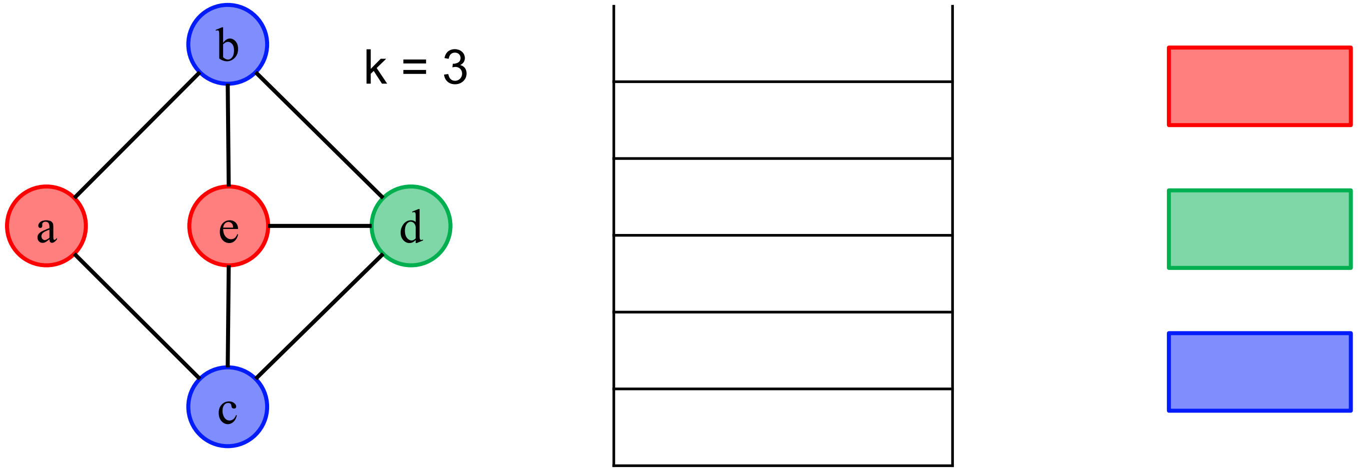
- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



南京大学编译技术
及软件安全研究组

# Chaitin's Algorithm

- Degree of a vertex is a loose upper bound on colorability
- A vertex whose degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty (color them later)



# Chaitin’s Algorithm

- When the algorithm reaches a state where every vertex has a degree ≥ k

# Chaitin's Algorithm

- When the algorithm reaches a state where every vertex has a degree ≥ k
- Choose a vertex immediately to spill
  - put the vertex into a spill list

# Chaitin's Algorithm

- When the algorithm reaches a state where every vertex has a degree ≥ k
- Choose a vertex immediately to spill
  - put the vertex into a spill list
  - remove the vertex from the graph

# Chaitin’s Algorithm

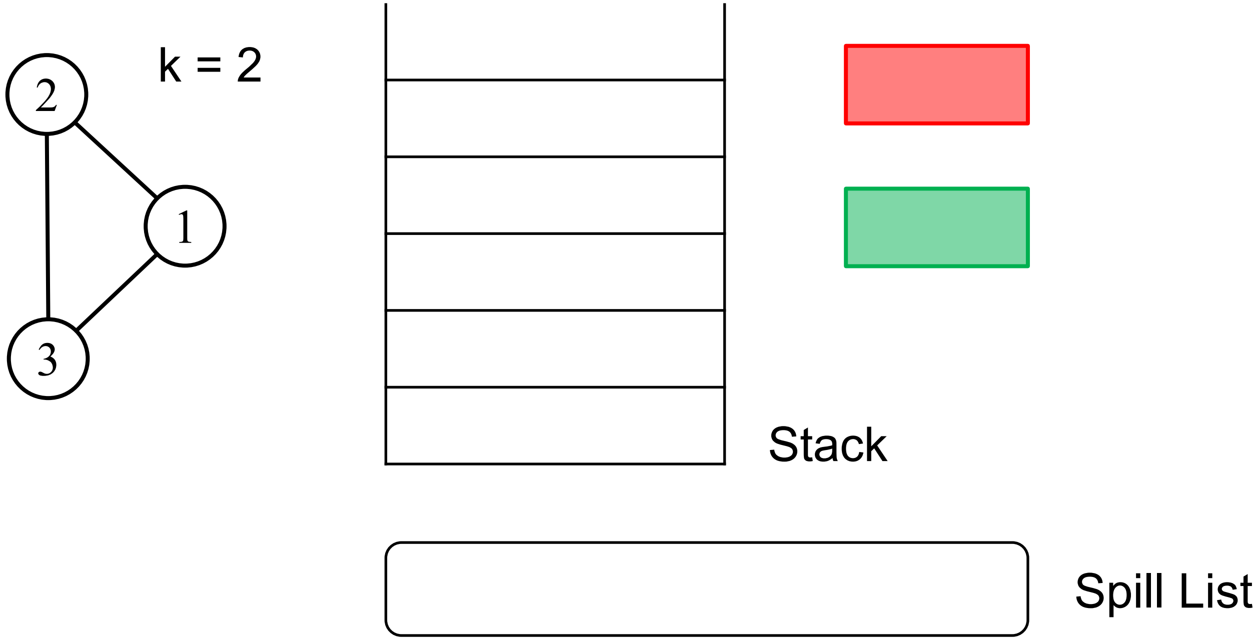
- When the algorithm reaches a state where every vertex has a degree ≥ k
- Choose a vertex immediately to spill
  - put the vertex into a spill list
  - remove the vertex from the graph
  - continue moving vertices < k degree from the graph to the stack

# Chaitin’s Algorithm

- When the algorithm reaches a state where every vertex has a degree ≥ k
- Choose a vertex immediately to spill
  - put the vertex into a spill list
  - remove the vertex from the graph
  - continue moving vertices < k degree from the graph to the stack
- If the spill list is not empty, insert spill code, rebuild conflict graph, and retry allocation.

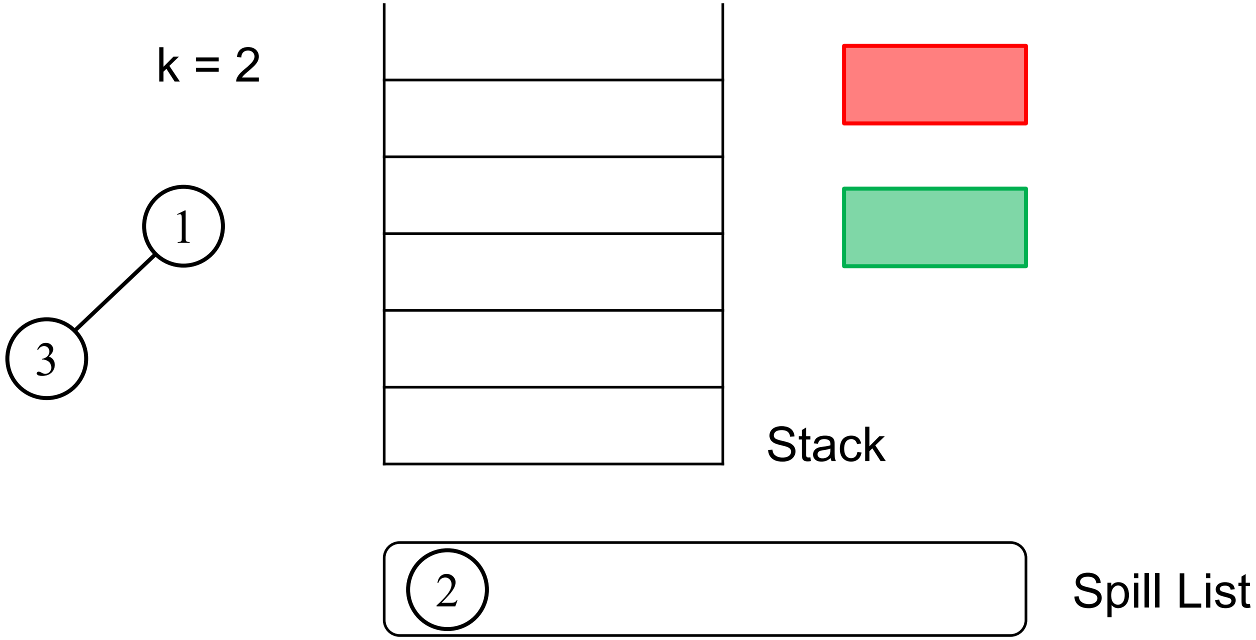
# Chaitin’s Algorithm

```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. MUL  R3  R1    R2    // R1 R2 R3
4. ST   x   R3          // R1 R2
5. ST   y   R2          // R1
6. ST   z   R1          //
```



# Chaitin's Algorithm

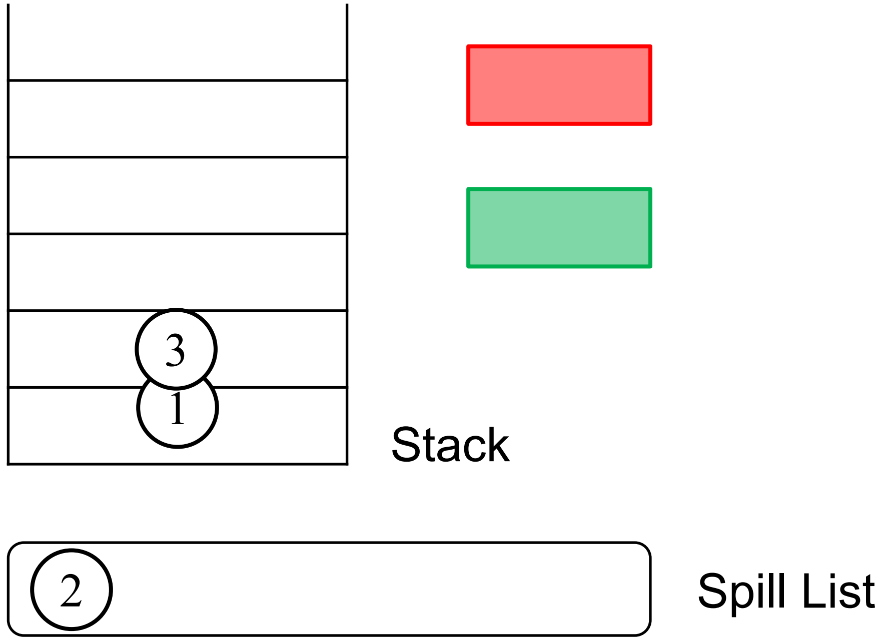
```
1. LD   R1  #1028        // R1
2. LD   R2  *R1          // R1 R2
3. MUL  R3  R1     R2    // R1 R2 R3
4. ST   x   R3           // R1 R2
5. ST   y   R2           // R1
6. ST   z   R1           //
```



# Chaitin's Algorithm

```
1. LD    R1   #1028        // R1
2. LD    R2   *R1          // R1 R2
3. MUL   R3   R1     R2    // R1 R2 R3
4. ST    x    R3           // R1 R2
5. ST    y    R2           // R1
6. ST    z    R1           //
```

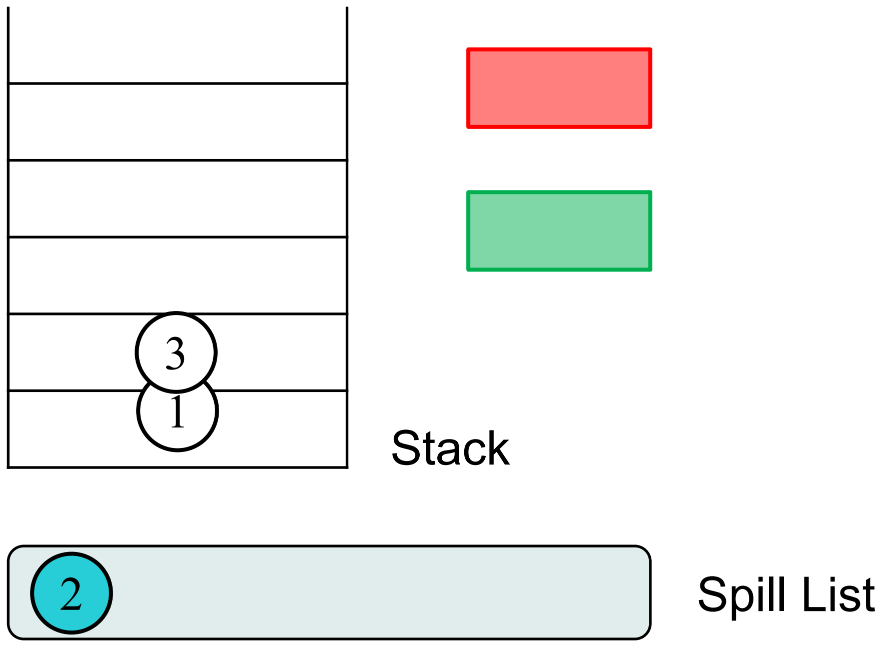
k = 2



# Chaitin’s Algorithm

```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. MUL  R3  R1    R2    // R1 R2 R3
4. ST   x   R3          // R1 R2
5. ST   y   R2          // R1
6. ST   z   R1          //
```

k = 2

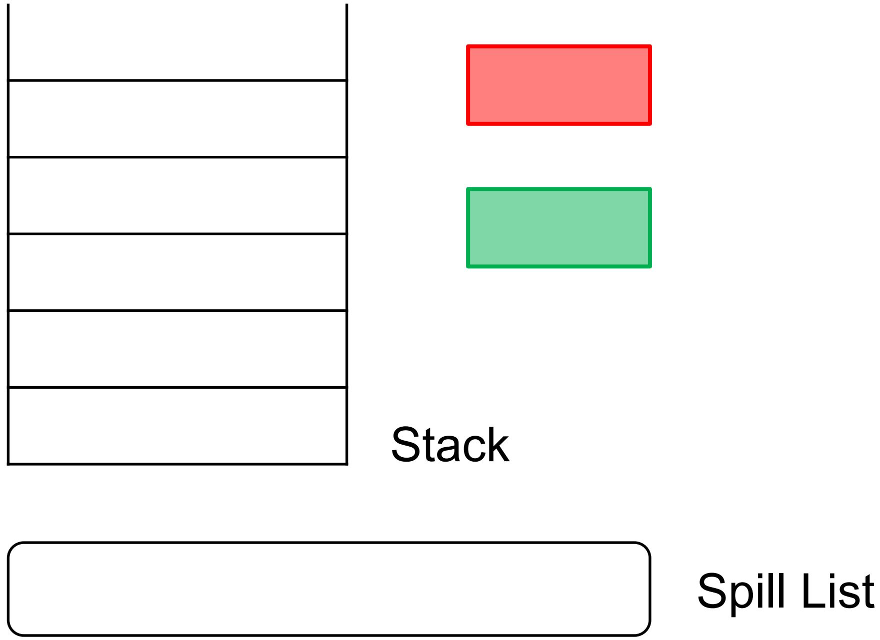



Spill R2!

# Chaitin's Algorithm

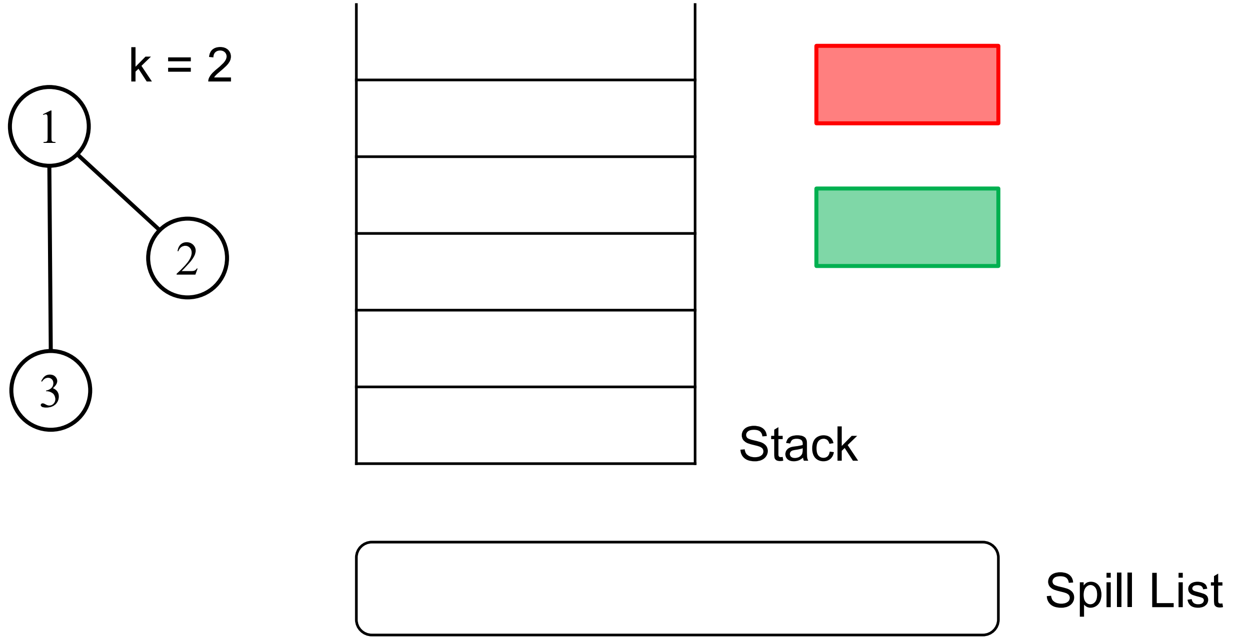
```
1. LD   R1  #1028        // R1
2. LD   R2  *R1          // R1 R2
3. ST   a   R2           // R1
4. MUL  R3  R1     R1    // R1      R3
5. ST   x   R3           // R1
6. LD   R2  a            // R1 R2
7. ST   y   R2           // R1
8. ST   z   R1           //
```

k = 2



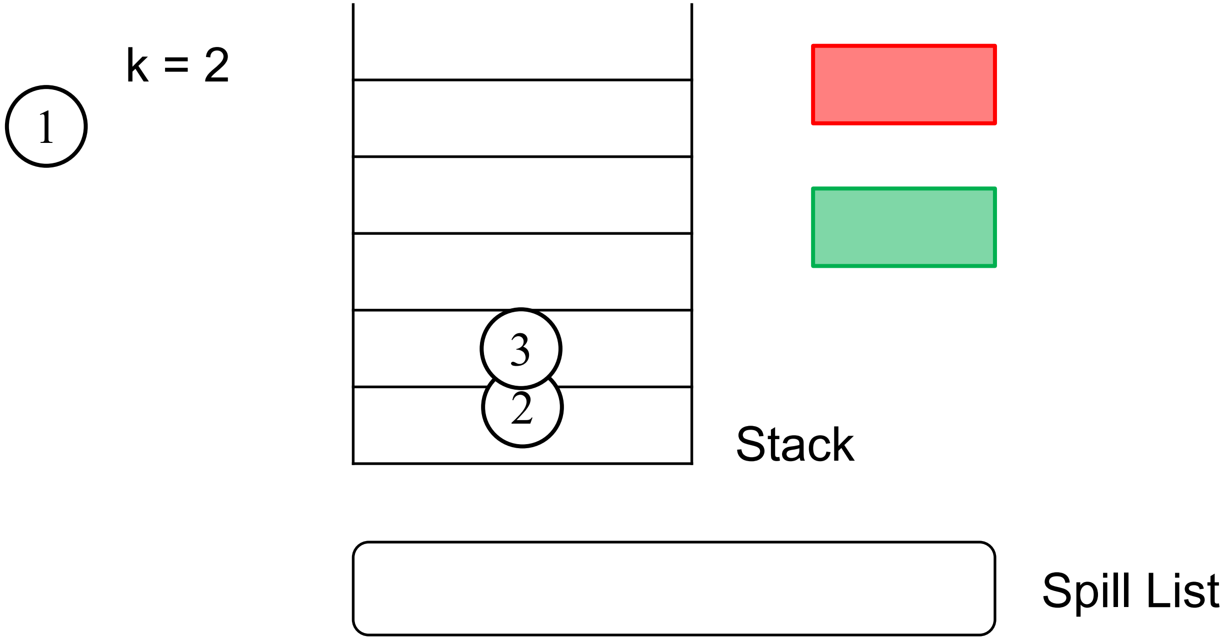
# Chaitin's Algorithm

```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```



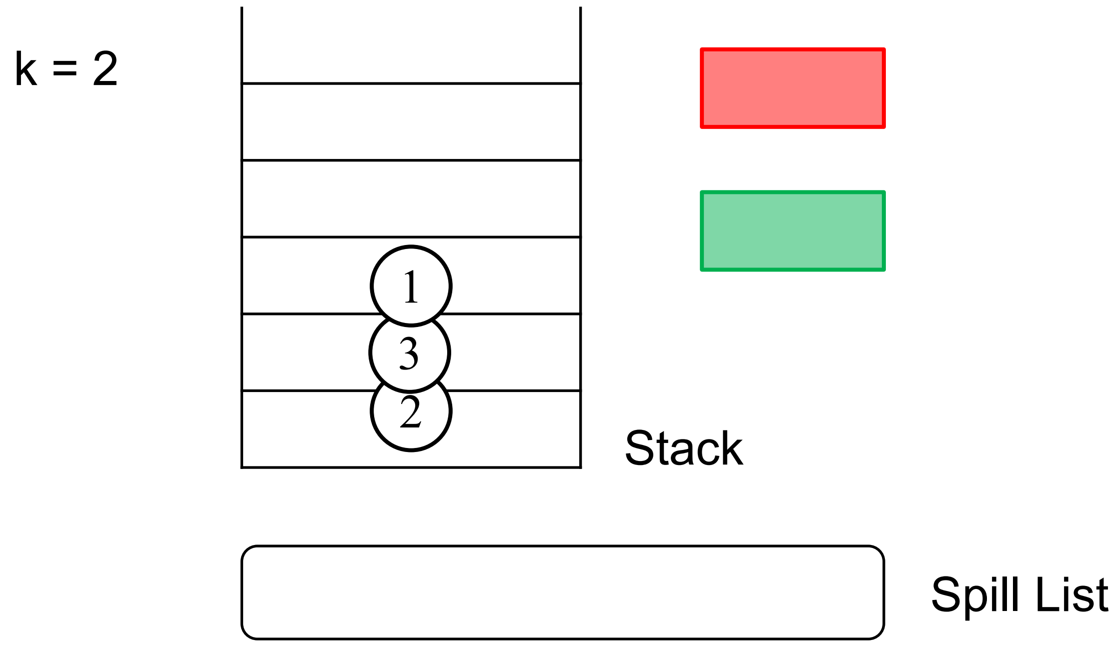
# Chaitin's Algorithm

```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1       R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```



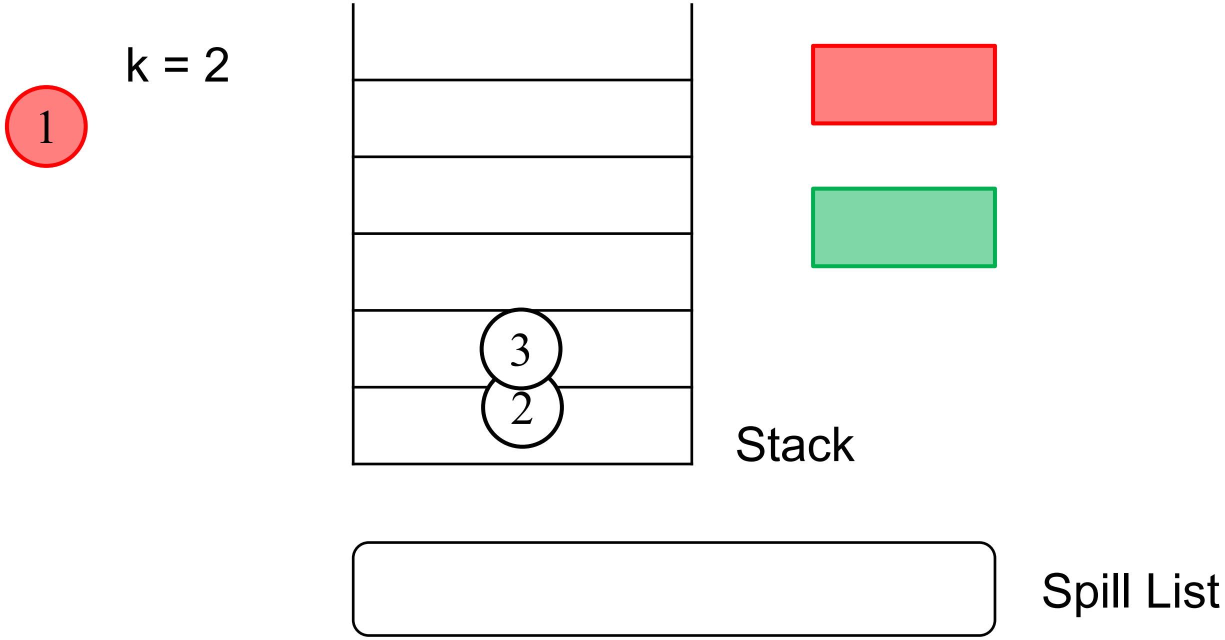
# Chaitin’s Algorithm

```
1.  LD    R1   #1028       // R1
2.  LD    R2   *R1         // R1 R2
3.  ST    a    R2          // R1
4.  MUL   R3   R1    R1    // R1       R3
5.  ST    x    R3          // R1
6.  LD    R2   a           // R1 R2
7.  ST    y    R2          // R1
8.  ST    z    R1          //
```



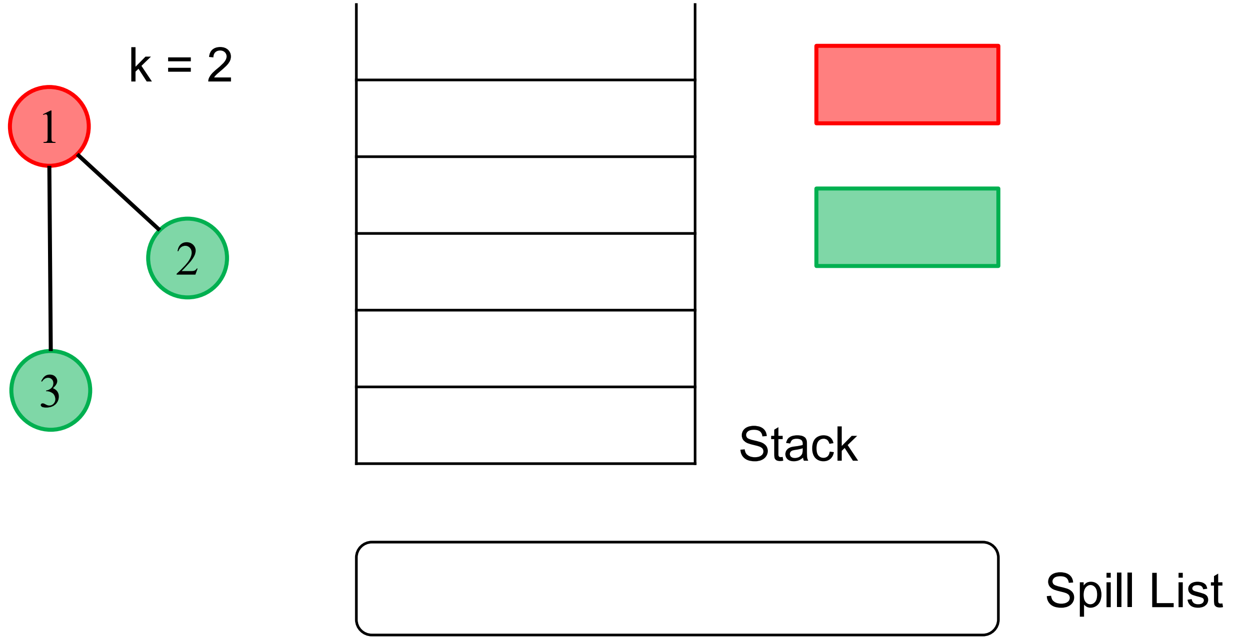
# Chaitin’s Algorithm

```
1.  LD    R1   #1028          // R1
2.  LD    R2   *R1            // R1 R2
3.  ST    a    R2             // R1
4.  MUL   R3   R1     R1      // R1     R3
5.  ST    x    R3             // R1
6.  LD    R2   a              // R1 R2
7.  ST    y    R2             // R1
8.  ST    z    R1             //
```



# Chaitin's Algorithm

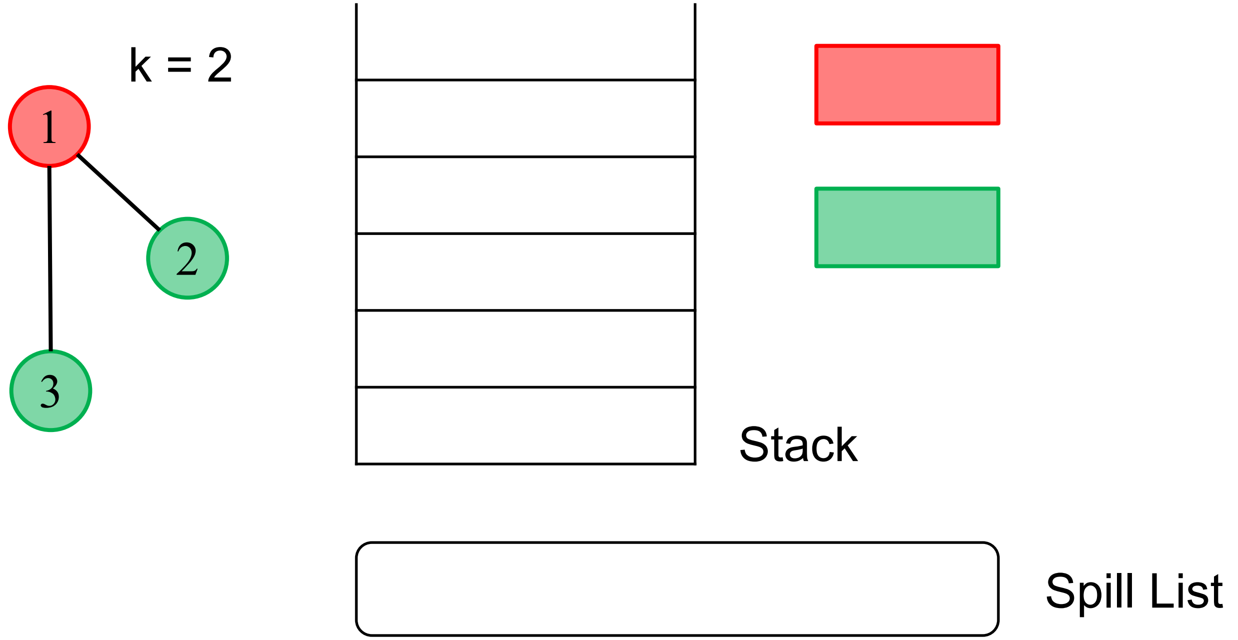
```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1     R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```



# Chaitin's Algorithm

Replace R1 with $R_{red}$
Replace R2/R3 with $R_{green}$

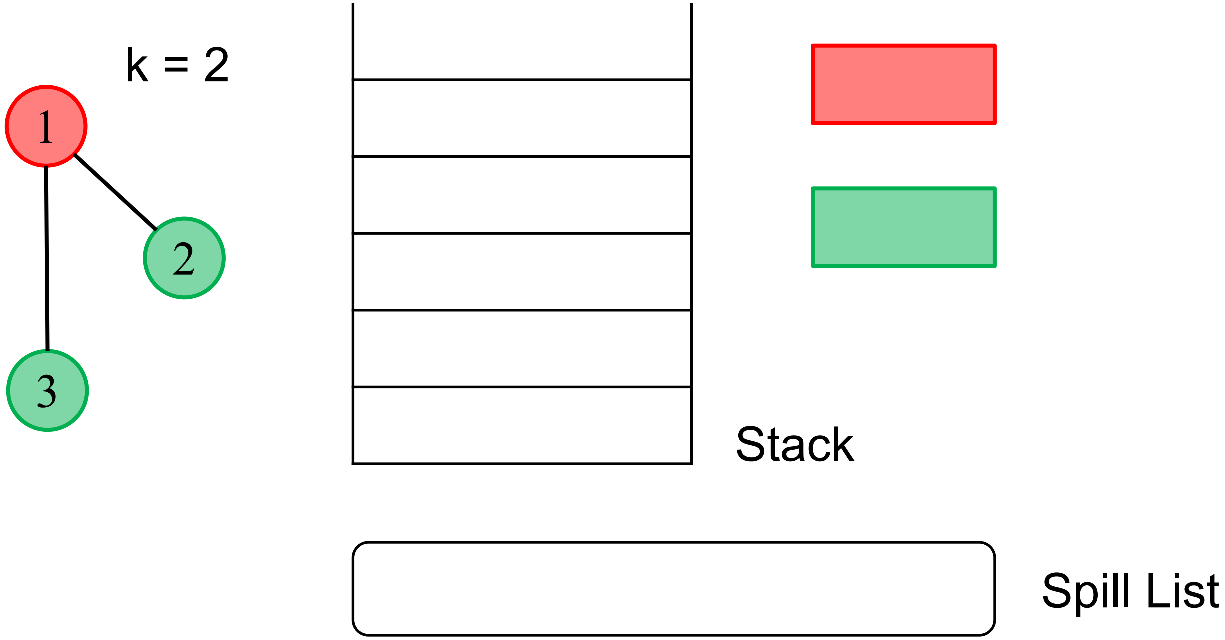
```
1. LD   R1  #1028       // R1
2. LD   R2  *R1         // R1 R2
3. ST   a   R2          // R1
4. MUL  R3  R1    R1    // R1      R3
5. ST   x   R3          // R1
6. LD   R2  a           // R1 R2
7. ST   y   R2          // R1
8. ST   z   R1          //
```



# Chaitin's Algorithm

Replace R1 with $R_{red}$
Replace R2/R3 with $R_{green}$

```
1. LD   Rred     #1028
2. LD   Rgreen   *Rred
3. ST   a        Rgreen
4. MUL  Rgreen   Rred     Rred
5. ST   x        Rgreen
6. LD   Rgreen   a
7. ST   y        Rgreen
8. ST   z        Rred
```



# Chaitin-Briggs Algorithm

- When Chaitin’s algorithm reaches a state where every vertex has a degree ≥ k
- Choose a vertex immediately to spill
  - put the vertex into a spill list
  - …

# Chaitin-Briggs Algorithm

- When Chaitin’s algorithm reaches a state where every vertex has a degree ≥ k
- Choose a vertex immediately to spill
  - put the vertex into a spill list
  - …
- **Briggs**:
  - Degree of a vertex is a **loose** upper bound on colorability
  - A vertex whose degree < k is **always** k-colorable

# Chaitin-Briggs Algorithm

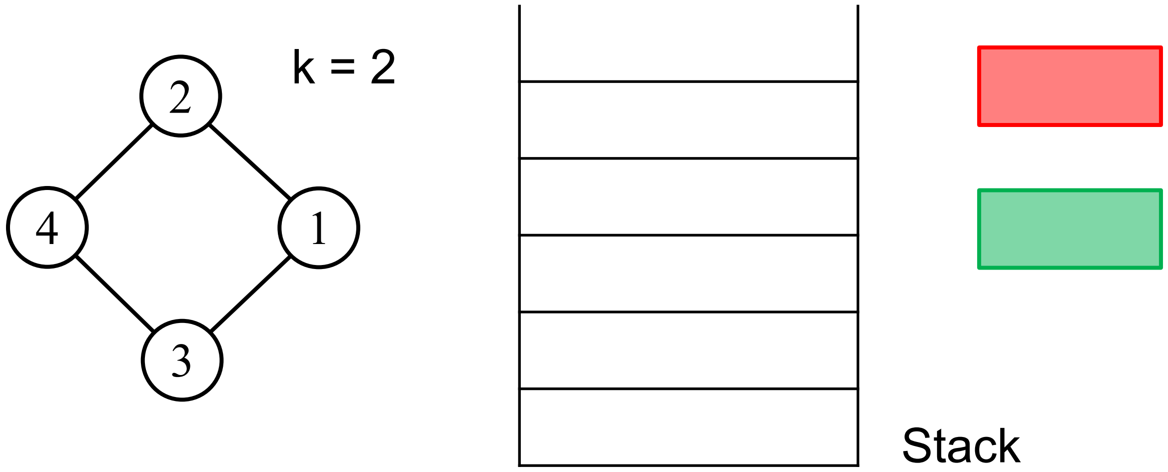
- When Chaitin's algorithm reaches a state where every vertex has a degree ≥ k
- Choose a vertex immediately to spill
  - put the vertex into a spill list
  - …

- **Briggs**:
  - Degree of a vertex is a **loose** upper bound on colorability
  - A vertex whose degree < k is **always** k-colorable
  - A vertex whose degree ≥ k **may also be** k-colorable

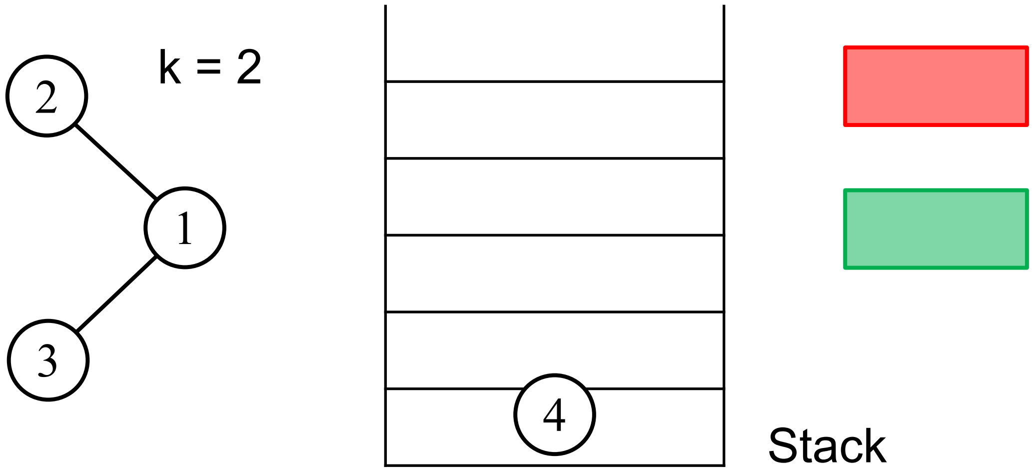
# Chaitin-Briggs Algorithm

- When Chaitin's algorithm reaches a state where every vertex has a degree ≥ k
- Choose a vertex immediately to spill
  - ~~put the vertex into a spill list~~
  - ~~…~~
- **Briggs**:
  - Push the vertex (to spill) to the stack.
  - We may have a color available when it is popped.

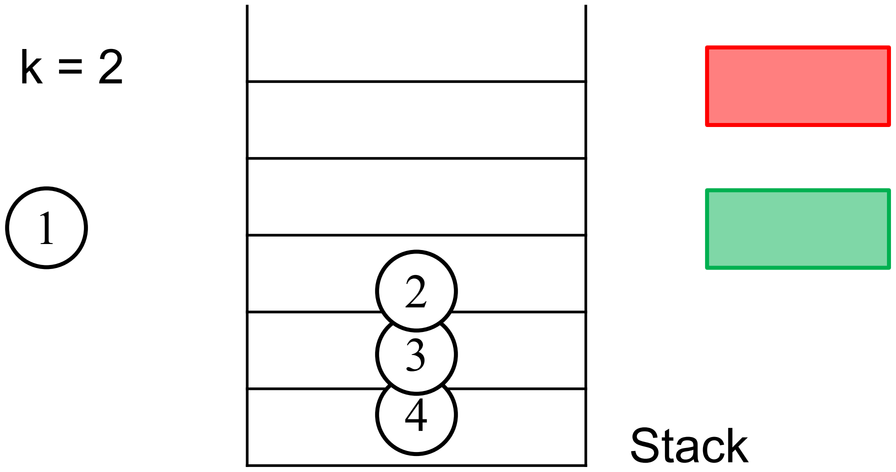
# Chaitin-Briggs Algorithm



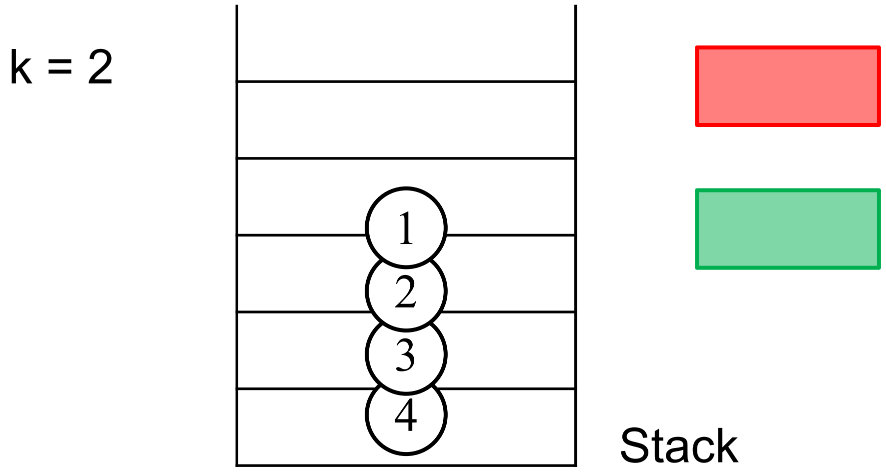
# Chaitin-Briggs Algorithm



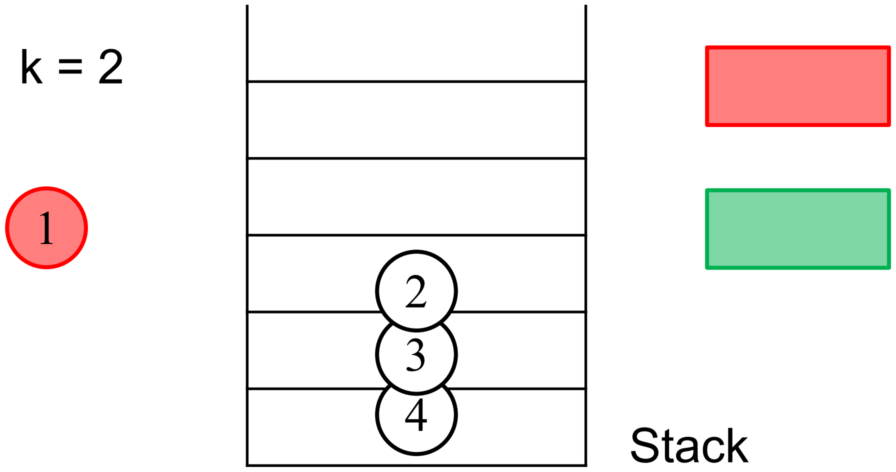
# Chaitin-Briggs Algorithm



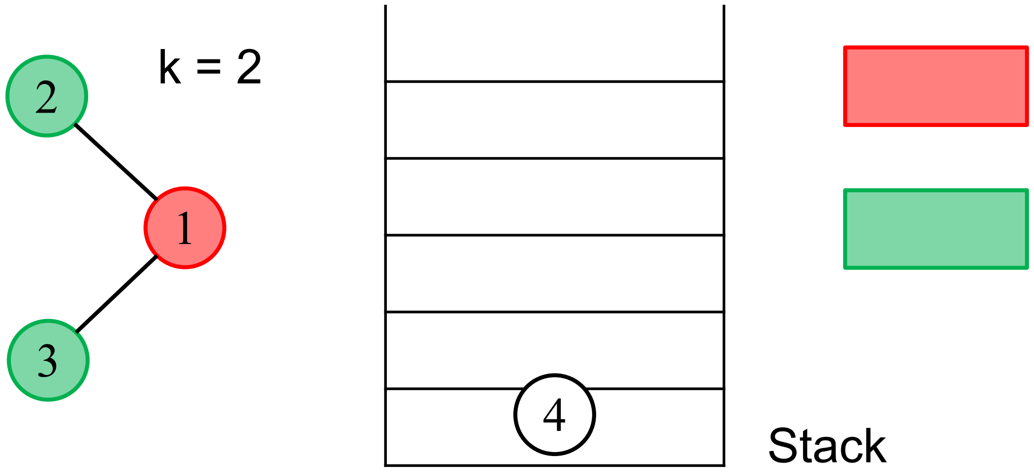
# Chaitin-Briggs Algorithm



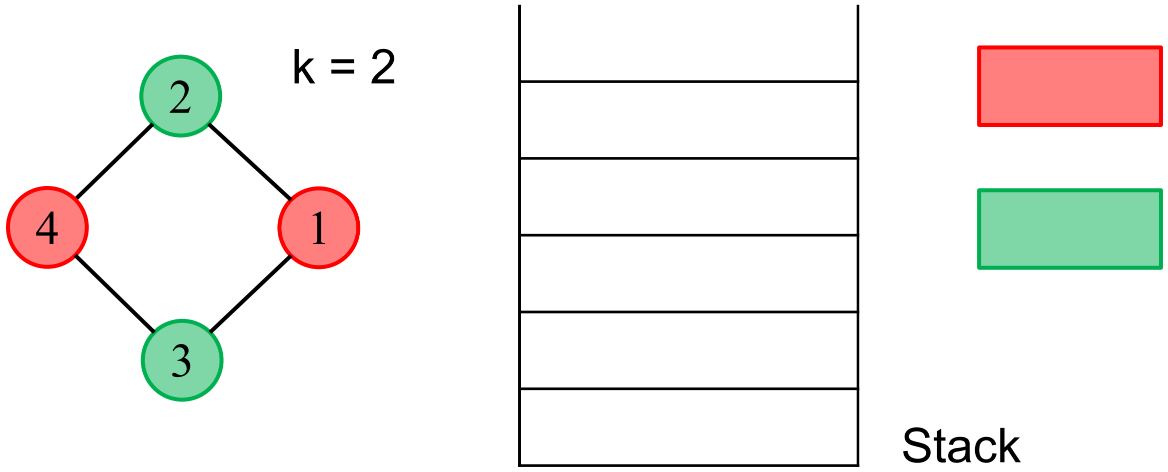
# Chaitin-Briggs Algorithm



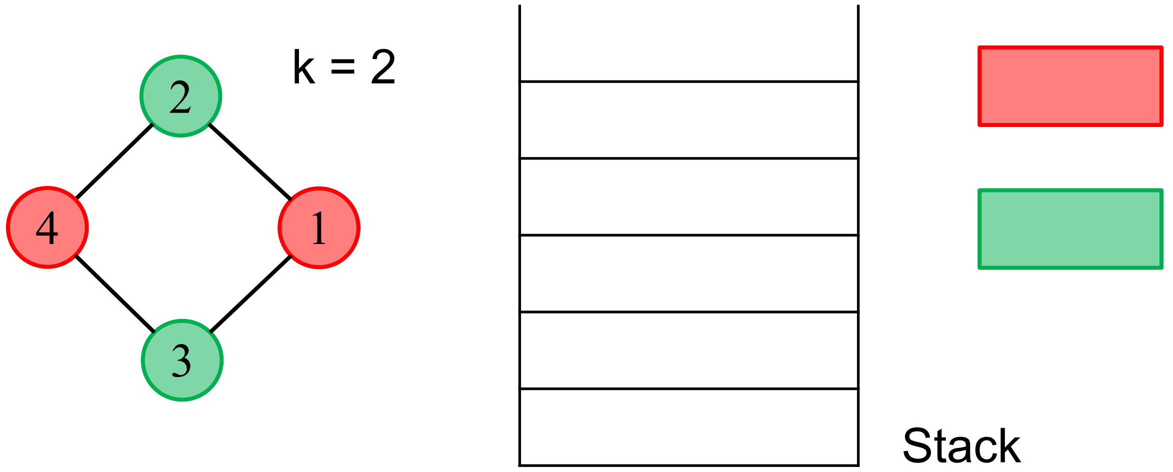
# Chaitin-Briggs Algorithm



# Chaitin-Briggs Algorithm

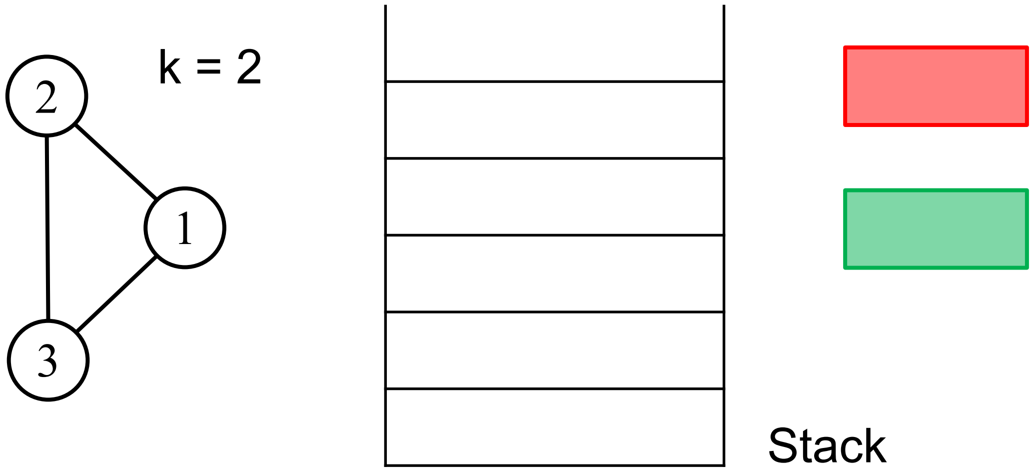


# Chaitin-Briggs Algorithm

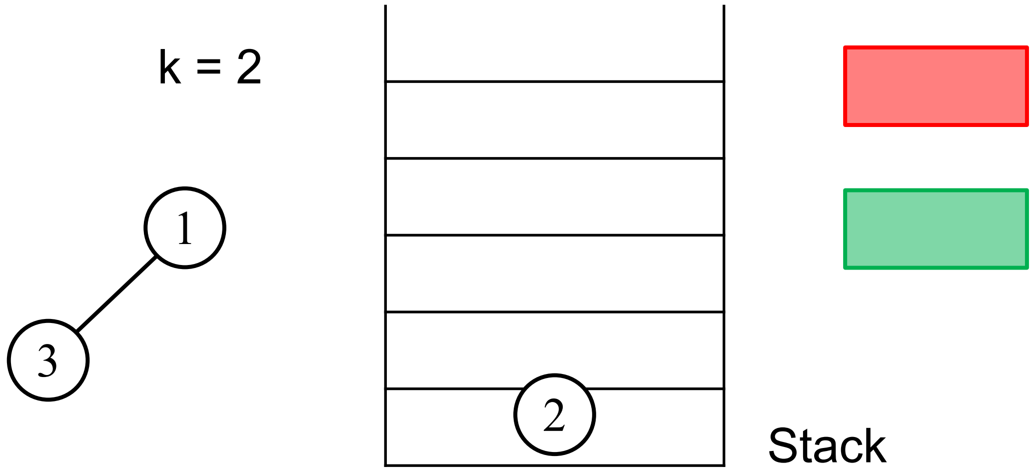



Sometimes, it is not necessary to spill!

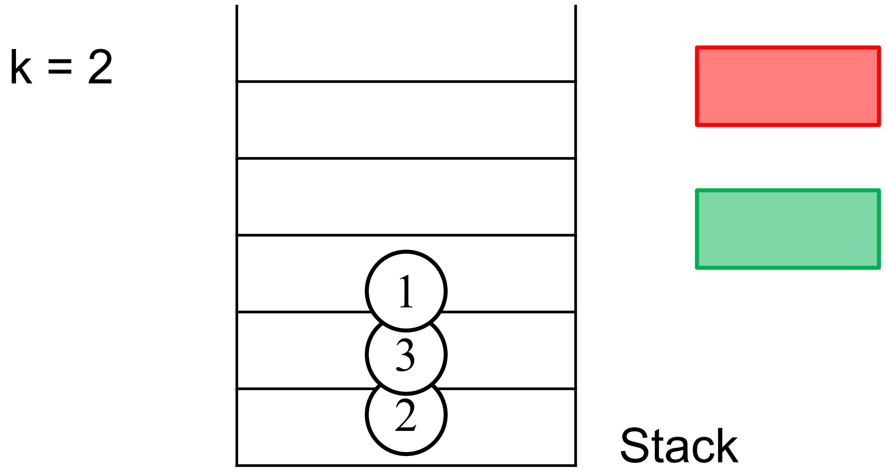
# Chaitin-Briggs Algorithm



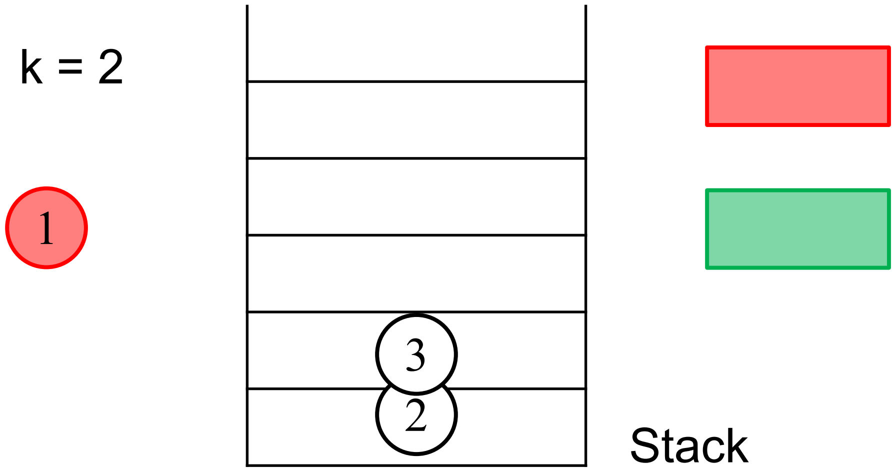
# Chaitin-Briggs Algorithm



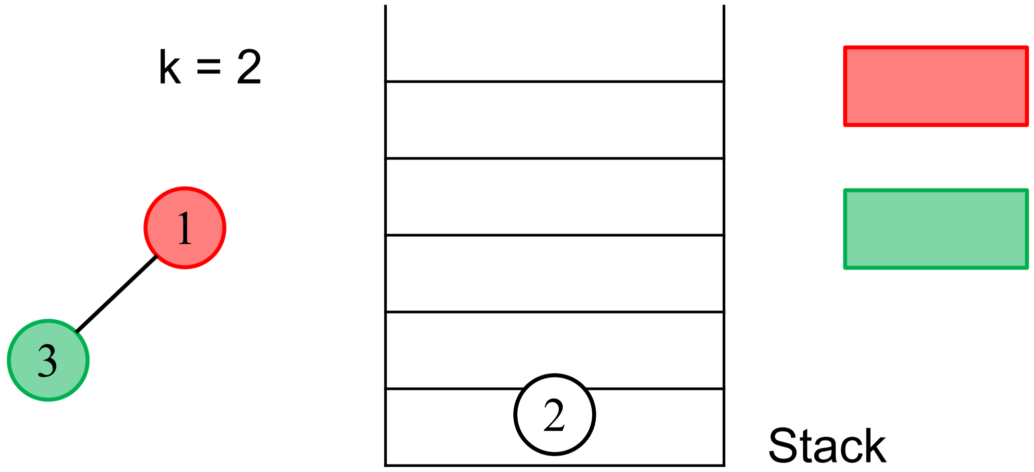
# Chaitin-Briggs Algorithm



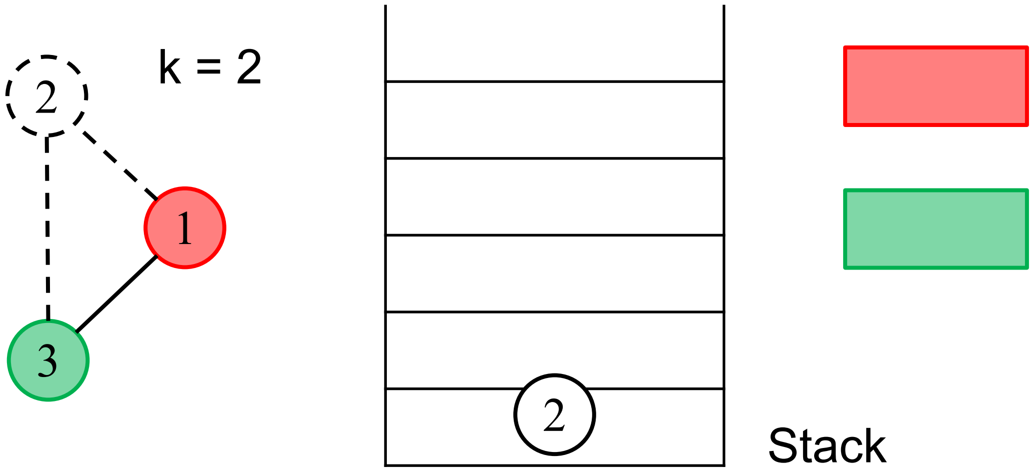
# Chaitin-Briggs Algorithm



# Chaitin-Briggs Algorithm



# Chaitin-Briggs Algorithm




When there is not any available color, spill it.

# Chaitin-Briggs Algorithm

- Compared to Chaitin’s algorithm, Chitin-Briggs algorithm delays the spill operation
  - saves a spill list
  - reduces spill code

# Spill Candidates

- Minimize spill cost, *i.e.*, the loads/stores needed
  - *e.g.*, avoid inner loops

# Spill Candidates

- Minimize spill cost, *i.e.*, the loads/stores needed
  - *e.g.*, avoid inner loops
- The higher the degree of a vertex
  - The greater the chance that spilling it will help coloring

# Spill Candidates

- Minimize spill cost, *i.e.*, the loads/stores needed
  - *e.g.*, avoid inner loops
- The higher the degree of a vertex
  - The greater the chance that spilling it will help coloring
- Don’t spill a value which is defined immediately followed by use
  - Spilling does not decrease live range

# Alternative Spilling

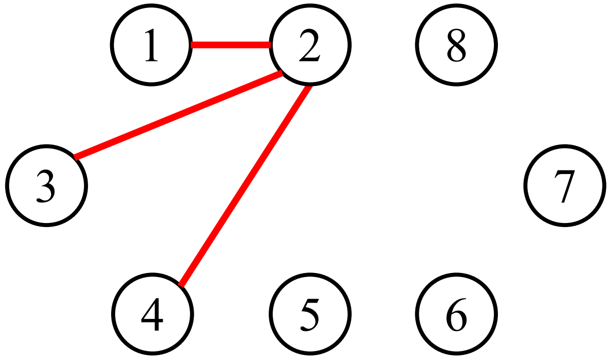
- Splitting Live Ranges, Coalescing Virtual Registers, etc.

# Alternative Spilling

- Splitting Live Ranges, Coalescing Virtual Registers, etc.

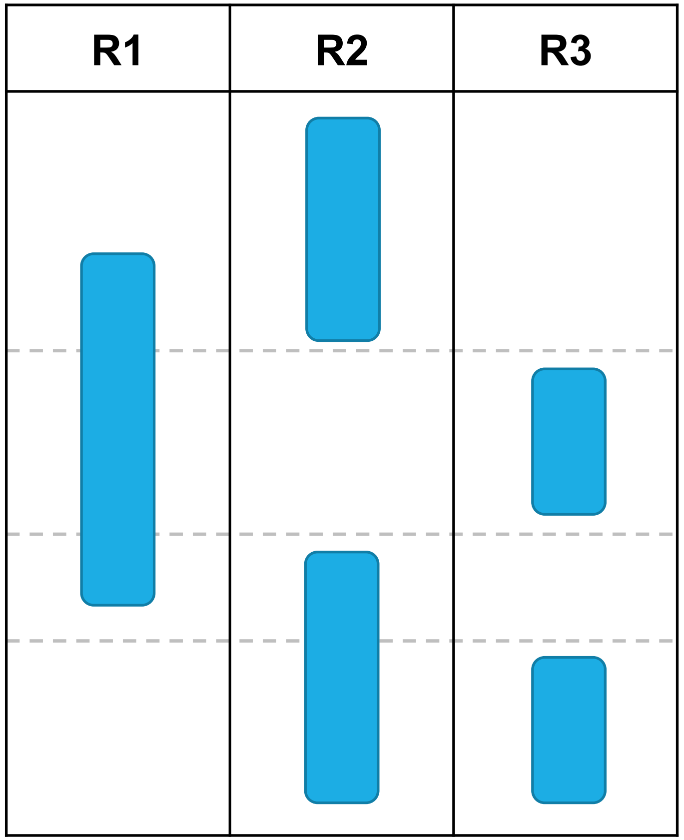
- Recall the **conflict graph**:

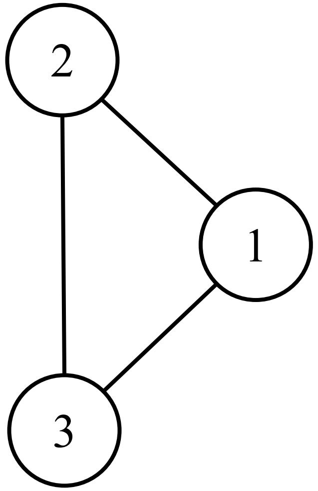
```
1. LD    R1    #1028        // R1
2. LD    R2    *R1          // R1 R2
3. MUL   R3    R1     R2    // R1 R2 R3
4. LD    R4    x            // R1 R2 R3 R4
5. SUB   R5    R4     R2    // R1    R3    R5
6. LD    R6    z            // R1    R3    R5 R6
7. MUL   R7    R5     R6    // R1    R3          R7
8. SUB   R8    R7     R3    // R1                   R8
9. ST    *R1   R8           //
```




**Conflict Graph**

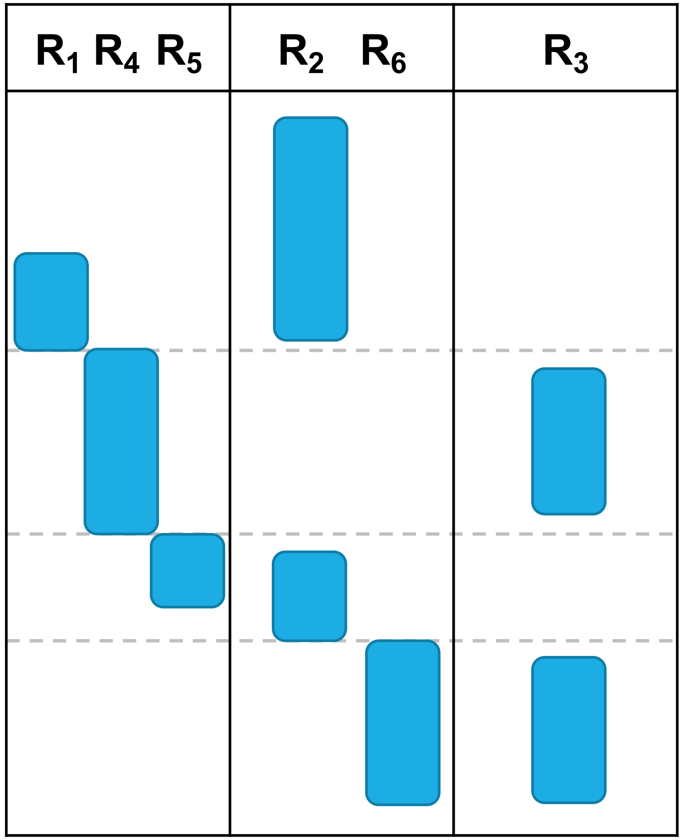
# Splitting Live Range

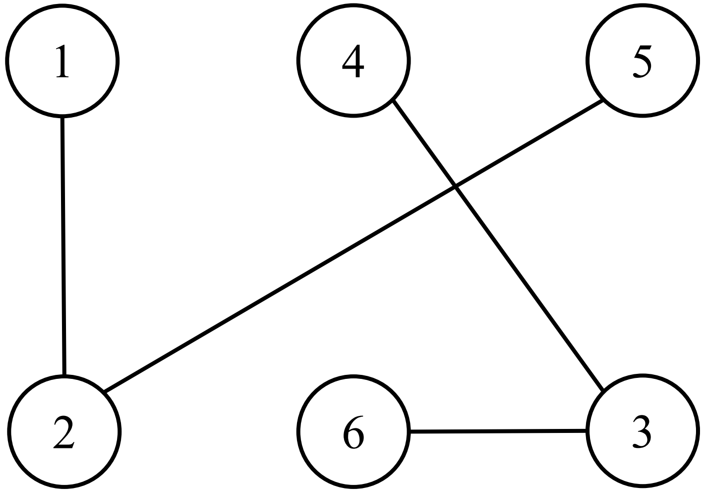



The conflict graph is not 2-colorable!

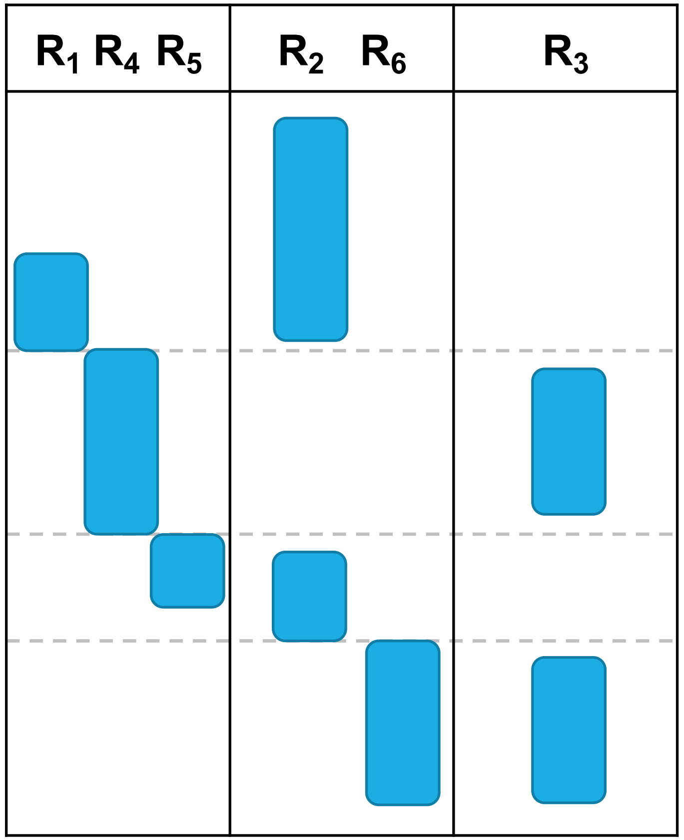
# Splitting Live Range

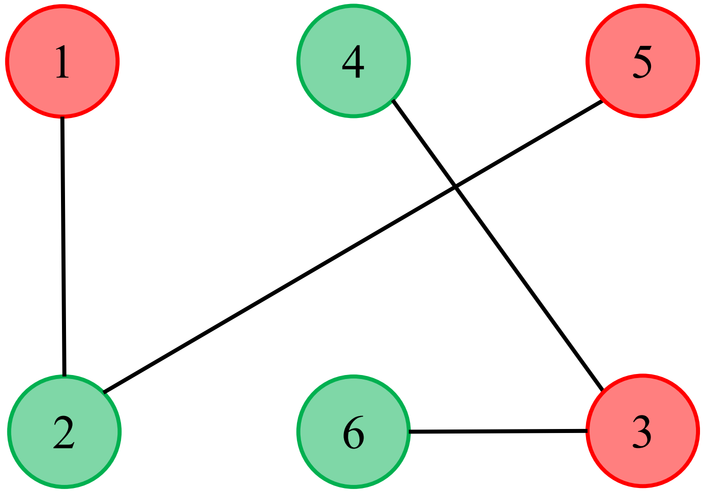



The conflict graph is 2-colorable!

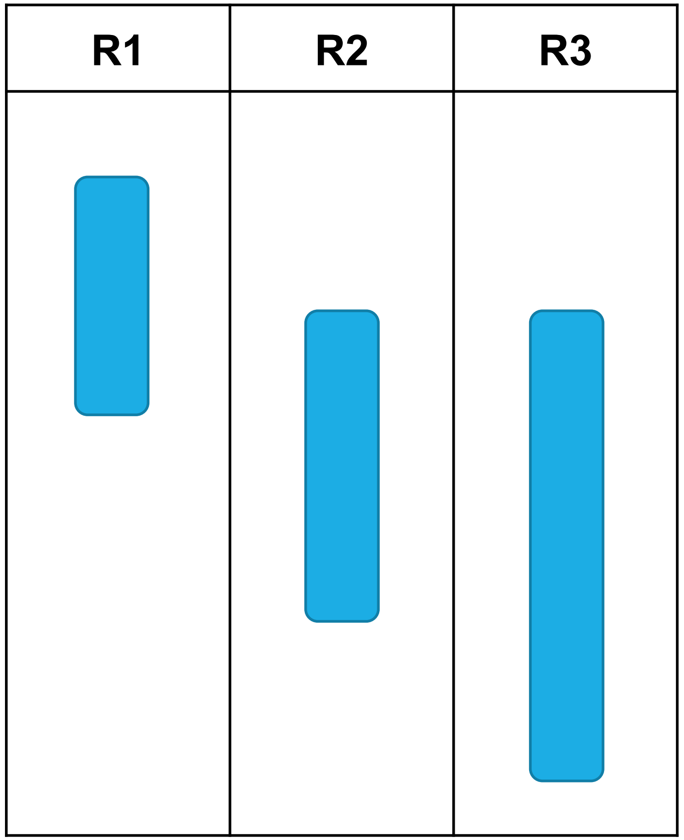
# Splitting Live Range

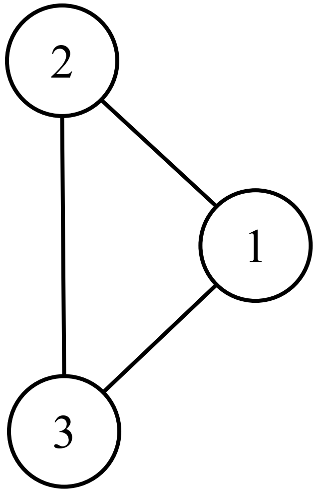


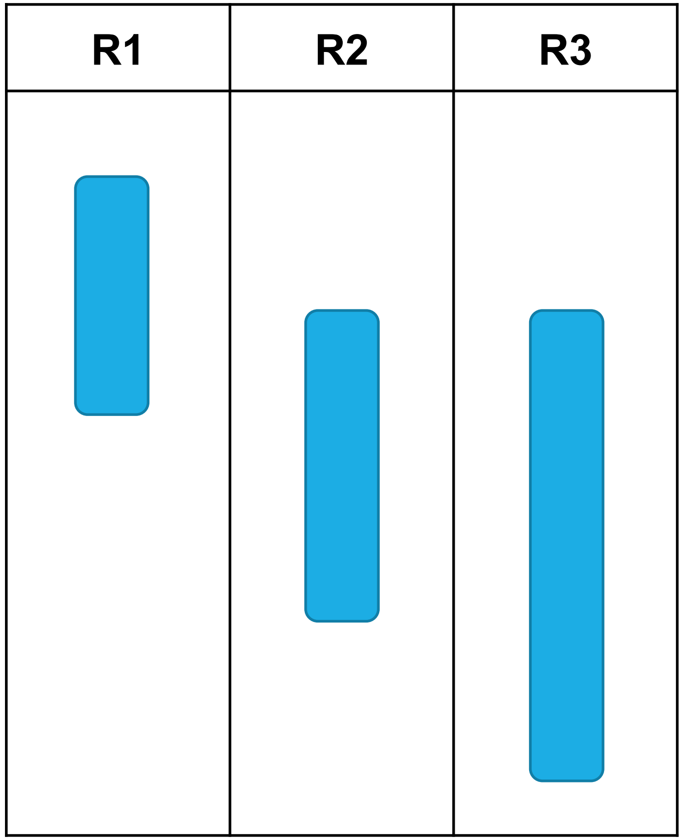

The conflict graph is 2-colorable!

# Coalescing Virtual Registers

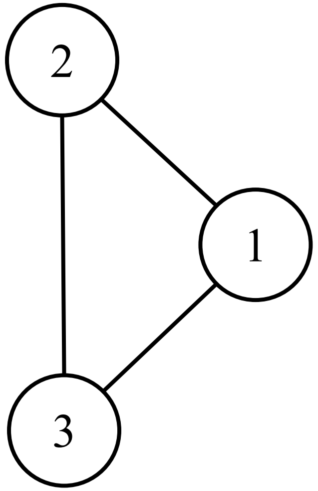


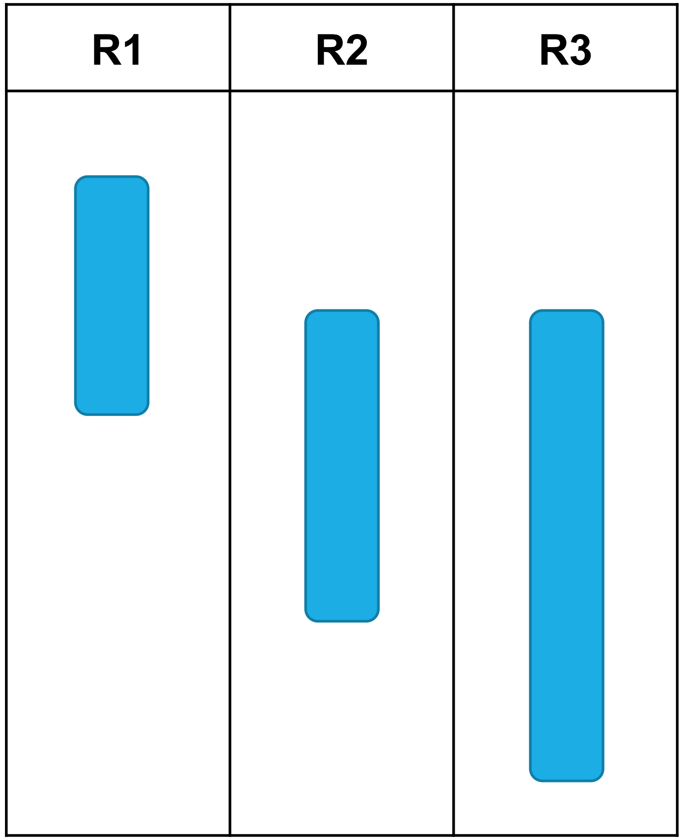
# Coalescing Virtual Registers



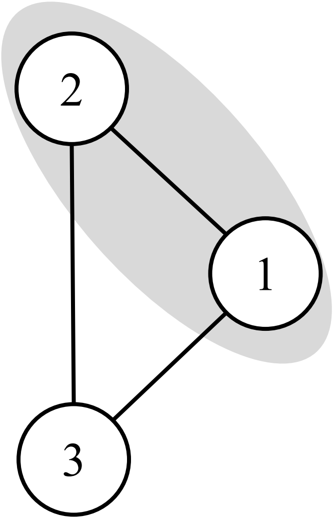

Assume R1 = R2 by LD R2 R1



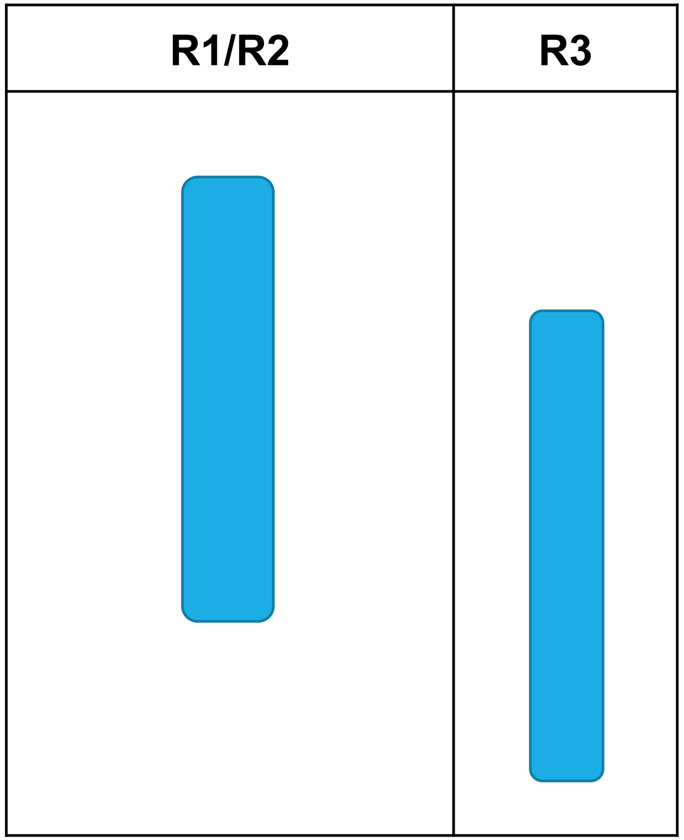
# Coalescing Virtual Registers



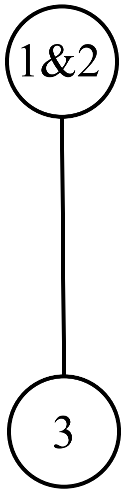

Assume R1 = R2 by LD R2 R1



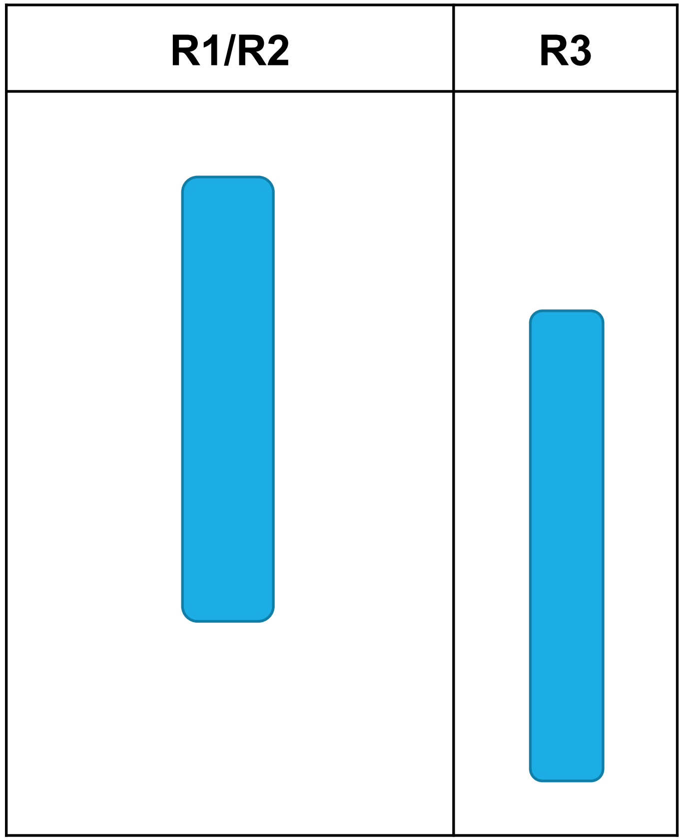
# Coalescing Virtual Registers



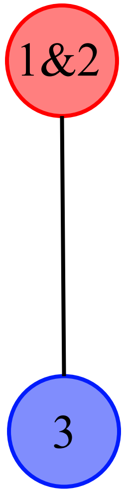

Assume R1 = R2 by LD R2 R1



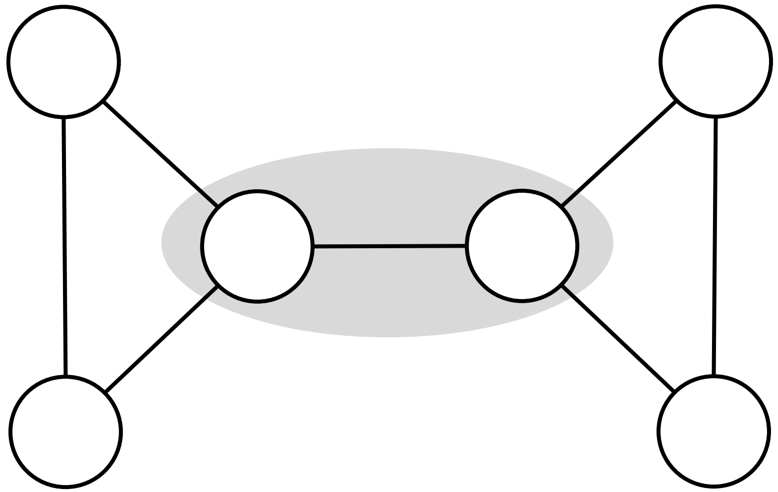
# Coalescing Virtual Registers



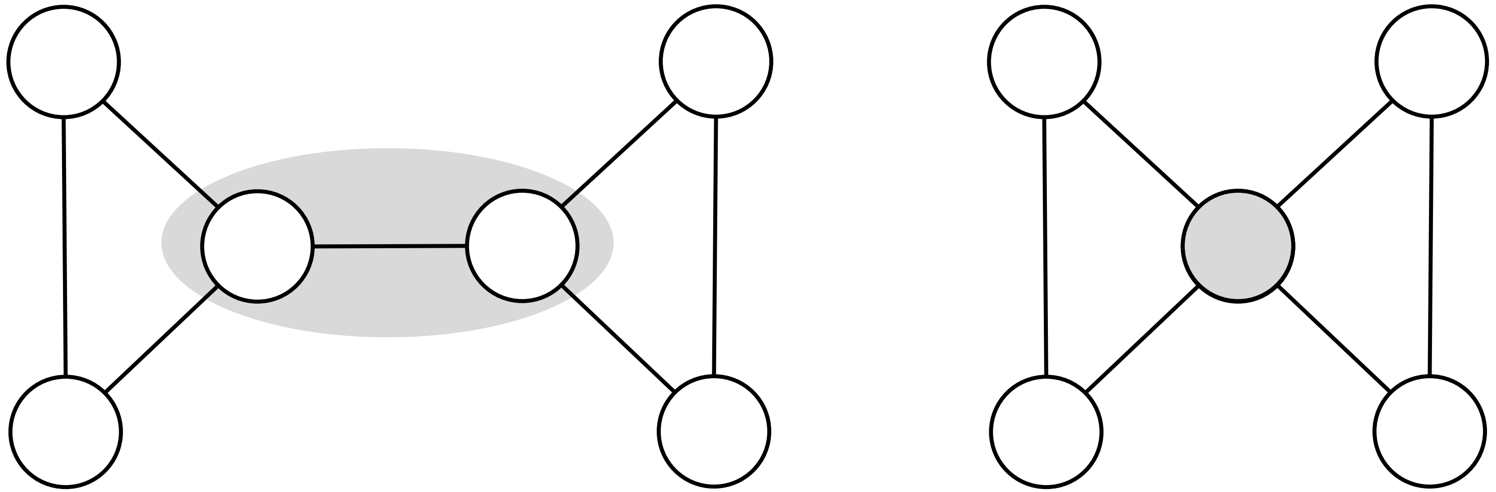

**Assume R1 = R2 by LD R2 R1**



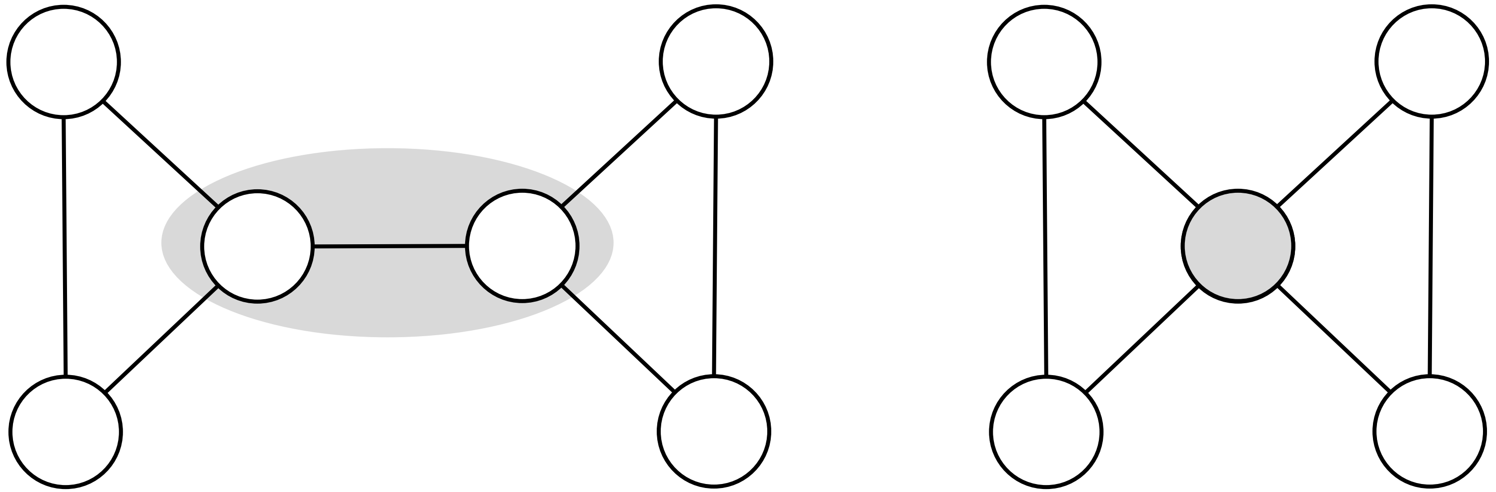
# Coalescing Doesn’t Always Work



# Coalescing Doesn't Always Work

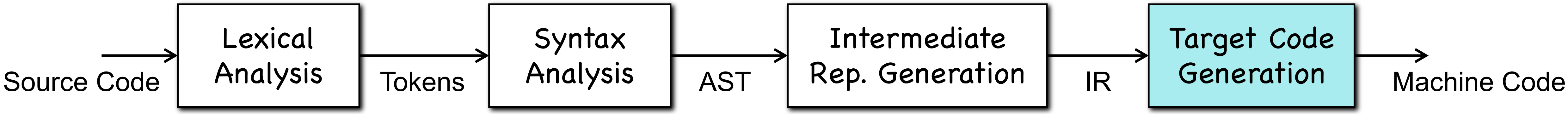


# Coalescing Doesn't Always Work



A vertex such that its degree < k is always k-colorable → Degree: 3 → 4
Harder to color

# Summary




- Speed: Registers > Memory
- Physical machines have limited number of registers
- **Register allocation**: ∞ virtual registers ➞ k physical registers
  - Part I – Local Register Allocation
  - Part II – Global Register Allocation

# THANKS!