Register Allocation

Live Ranges and Live Intervals

- Recall: A variable is **live** at a particular program point if its value may be read later before it is written.
  - Can find this using global liveness analysis.
- The **live range** for a variable is the set of program points at which that variable is live.
- The **live interval** for a variable is the smallest subrange of the IR code containing all a variable's live ranges.
  - A property of the IR code, **not** the CFG.
  - Less precise than live ranges, but simpler to work with.

Computing Live Variables

- To know if a variable will be used at some point, we iterate across the statements in a basic block in reverse order.
- Initially, some small set of values are known to be live (which ones depends on the particular program).
- When we see the statement **a = b + c**:
  - Just before the statement, **a** is not alive, since its value is about to be overwritten.
  - Just before the statement, both **b** and **c** are alive, since we're about to read their values.
  - *(what if we have* **a = a + b***?)*

Liveness Analysis

```
a = b;
c = a;
d = a + b;
e = d;
d = a;
f = e;
```

Liveness Analysis

```
  a = b;

  c = a;

d = a + b;

  e = d;

  d = a;

  f = e;
{ b, d }
```

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;

d = a;
```
`{ b, d, e }`
```
f = e;
```
`{ b, d }`

Liveness Analysis

```
a = b;

c = a;

d = a + b;

e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
a = b;

c = a;

d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
a = b;

c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

Liveness Analysis

```
{ b }
a = b;
{ a, b }
c = a;
{ a, b }
d = a + b;
{ a, b, d }
e = d;
{ a, b, e }
d = a;
{ b, d, e }
f = e;
{ b, d }
```

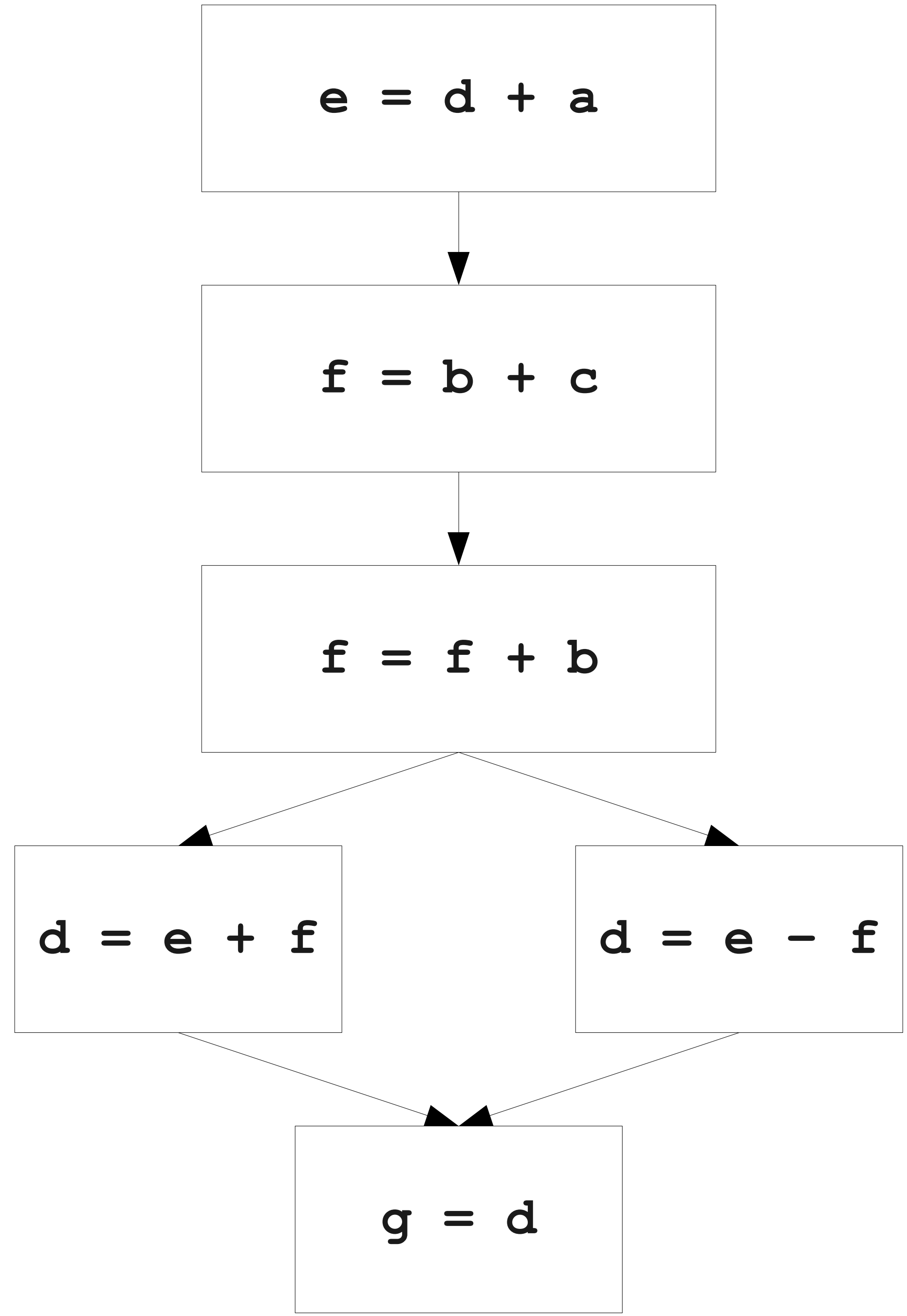
Live Ranges and Live Intervals

Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

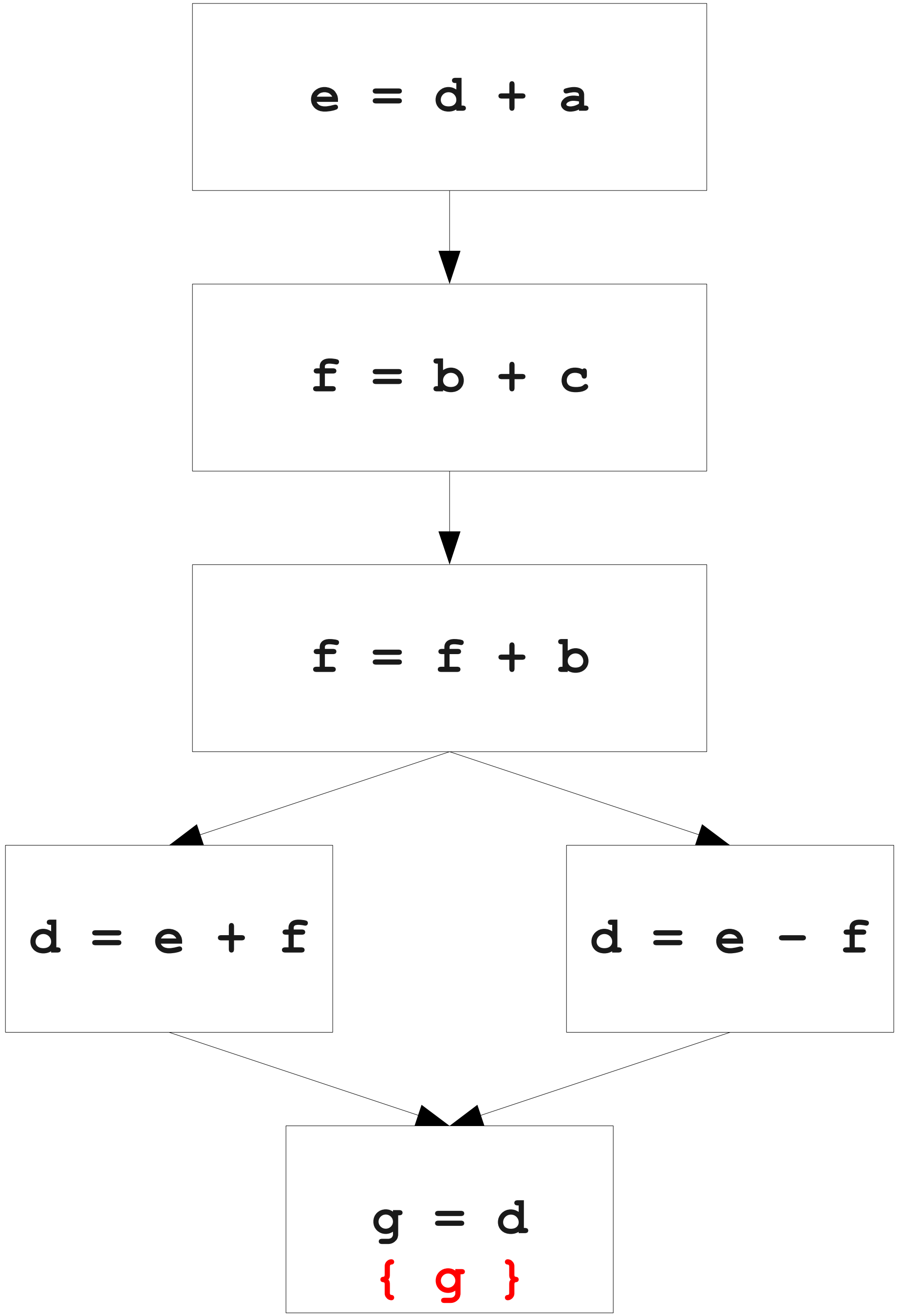
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



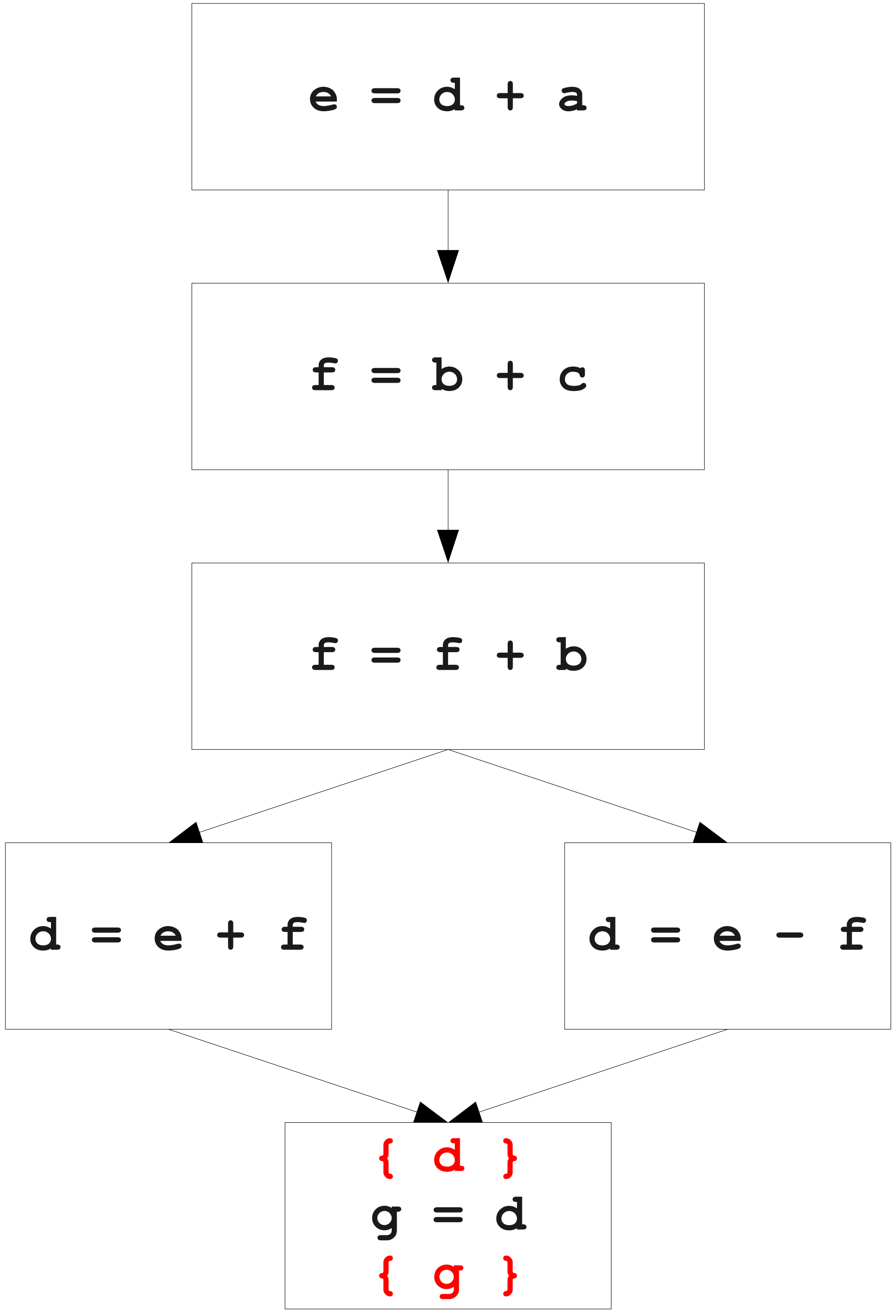
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



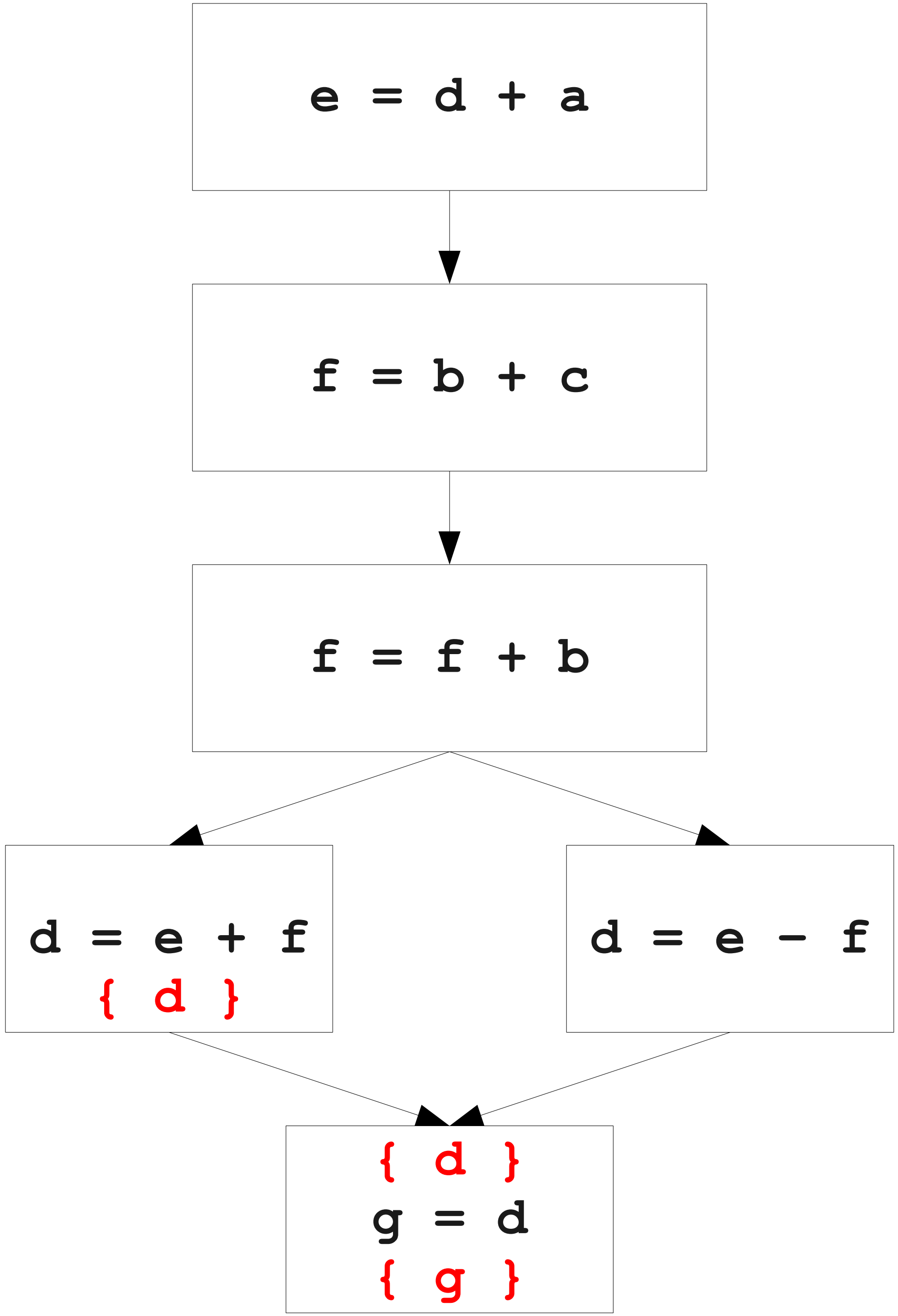
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



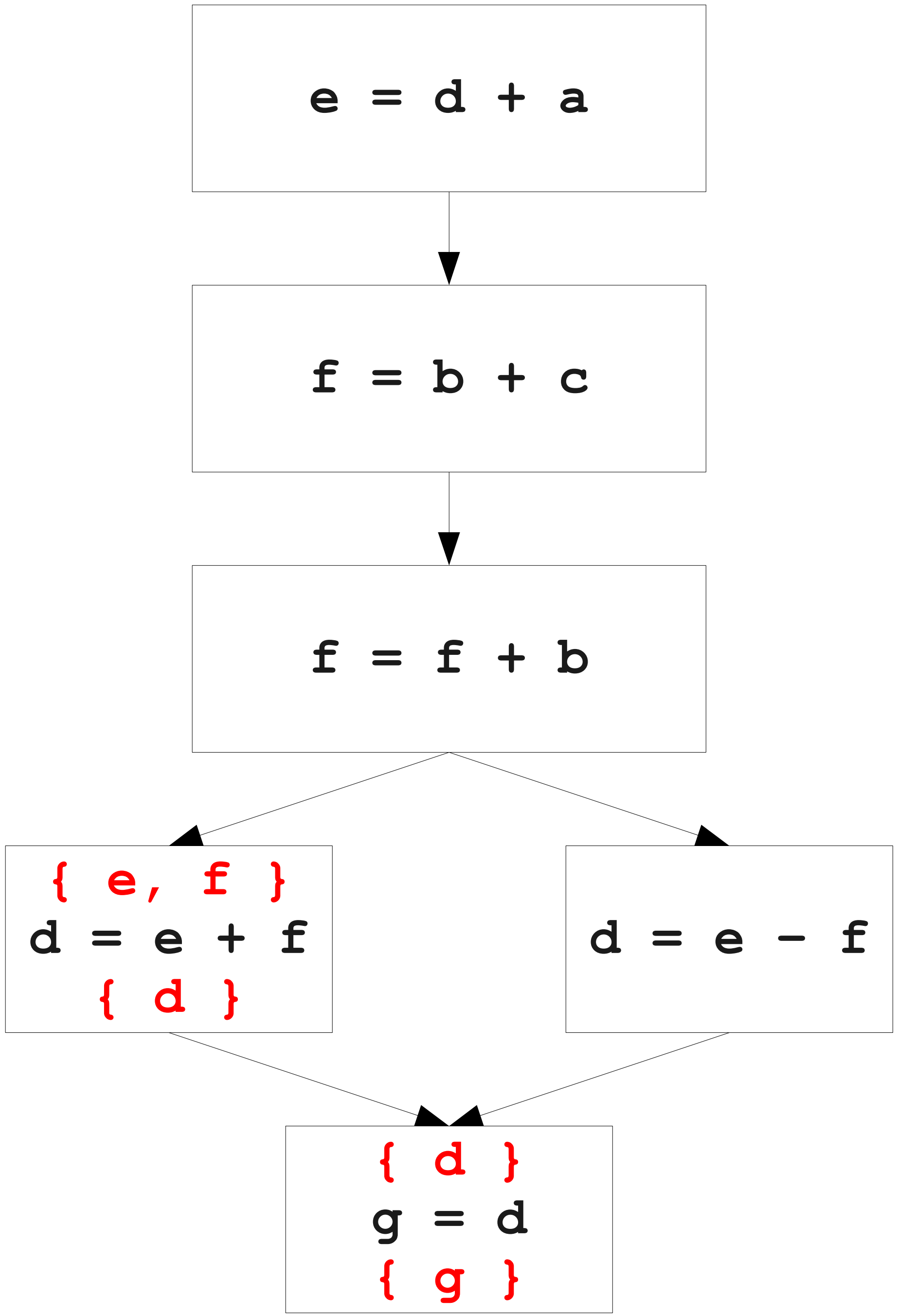
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



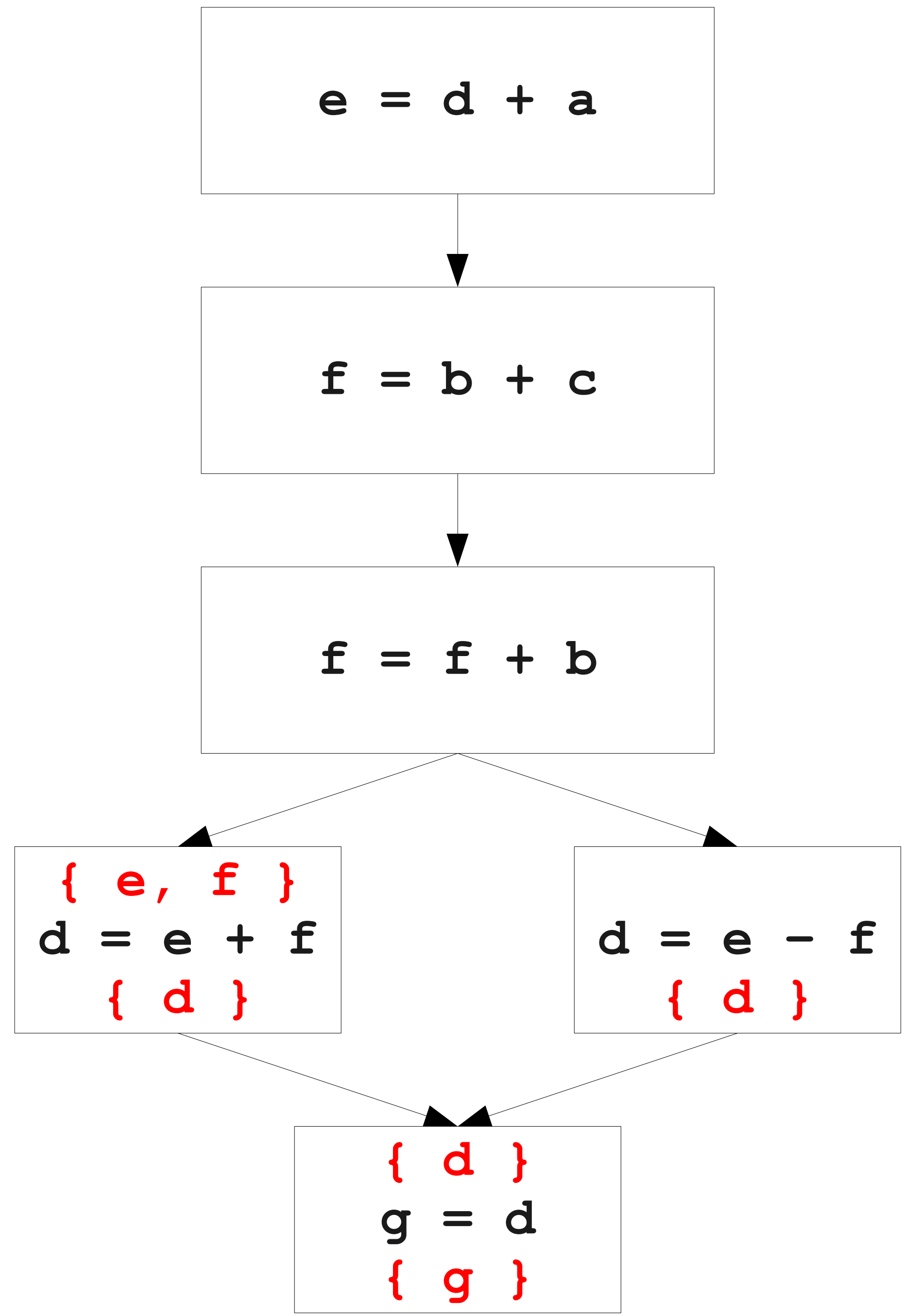
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



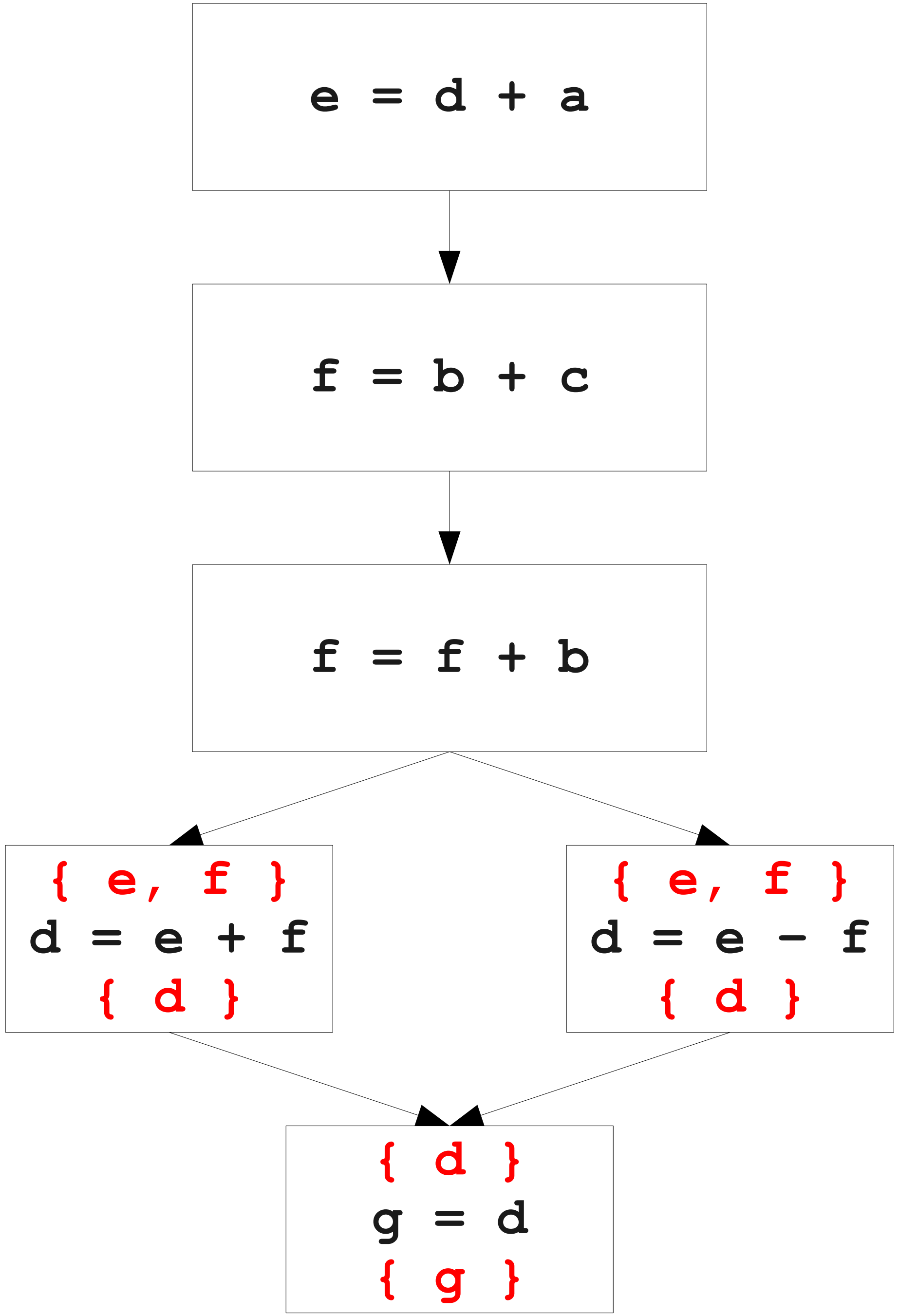
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



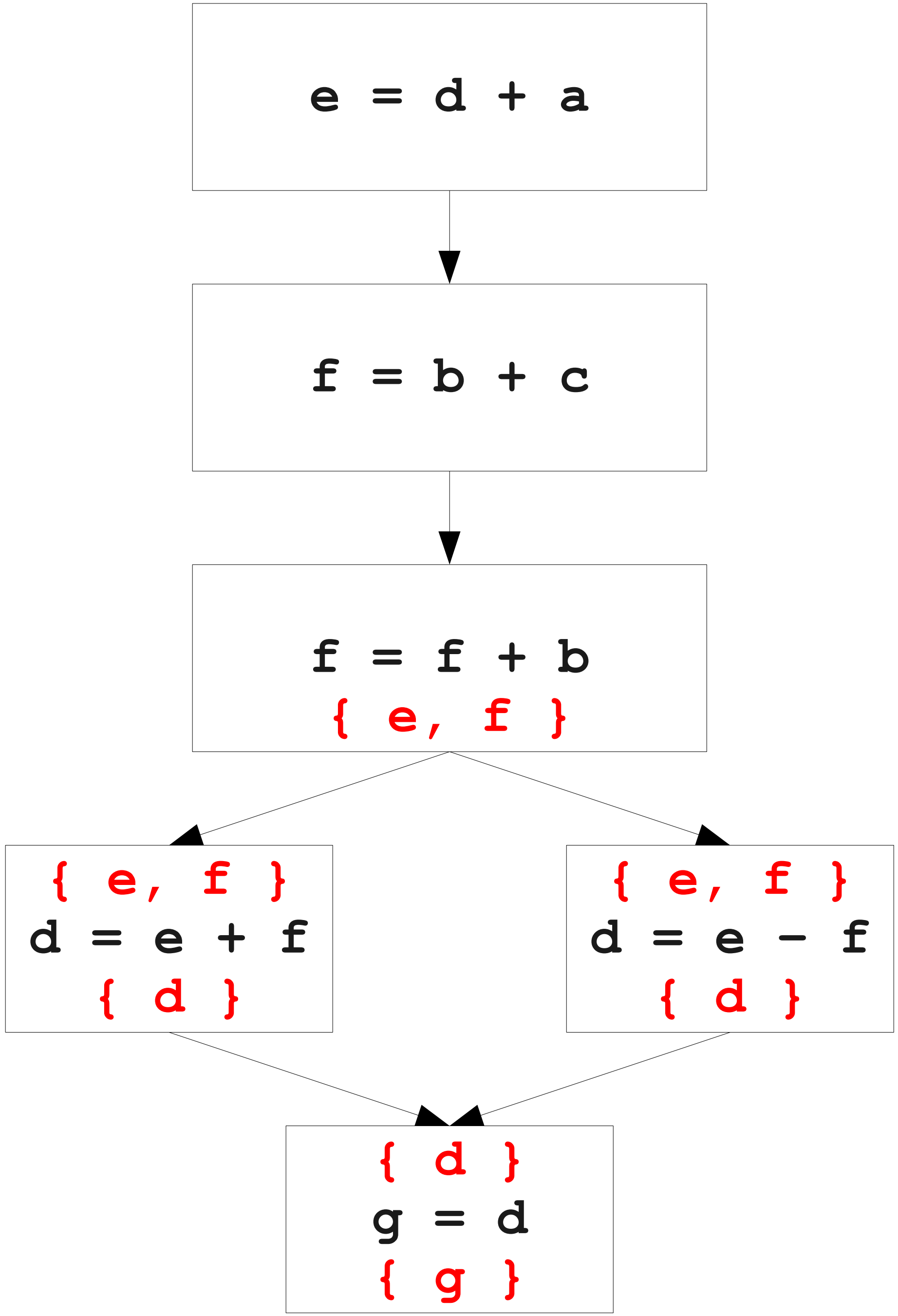
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



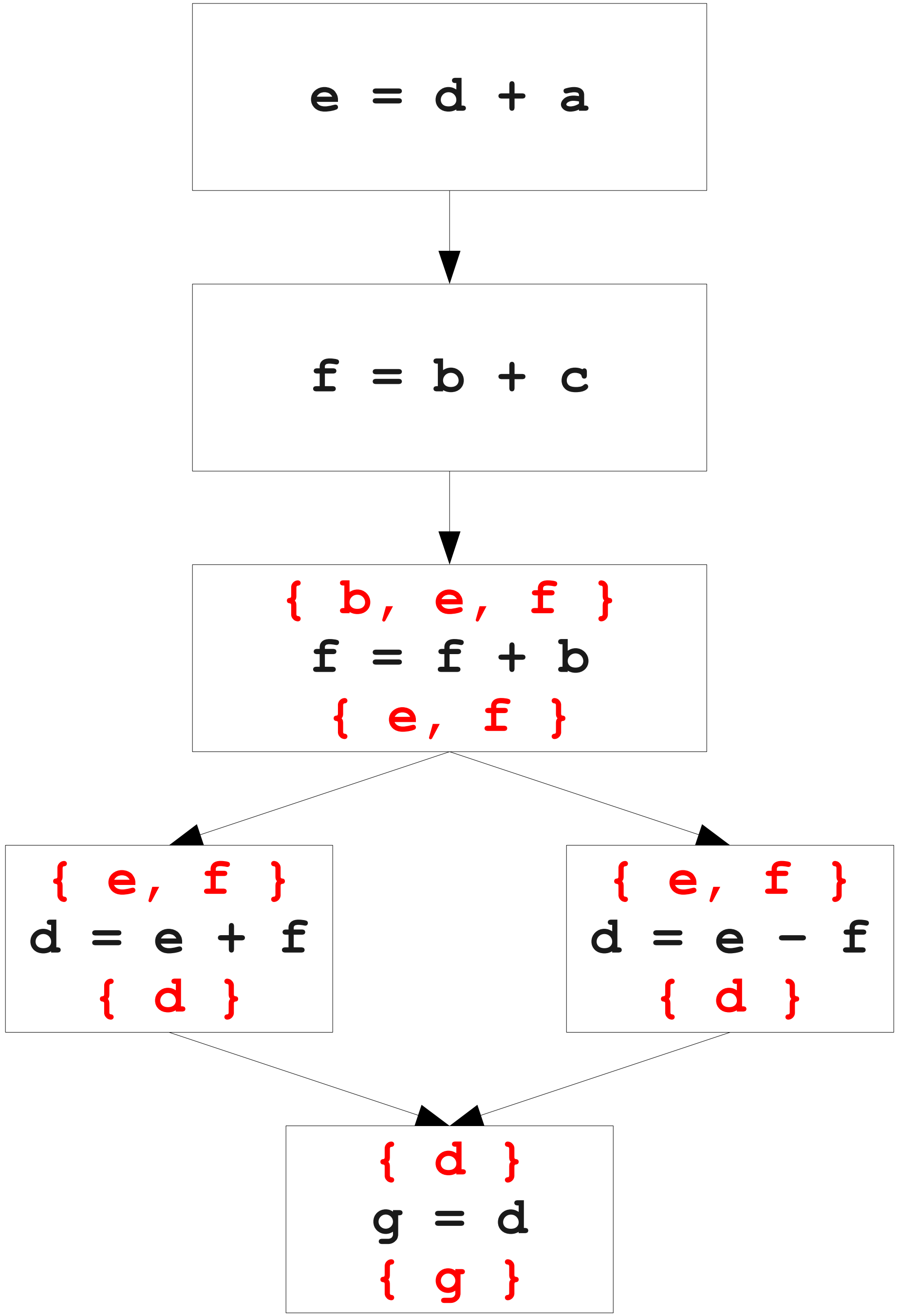
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



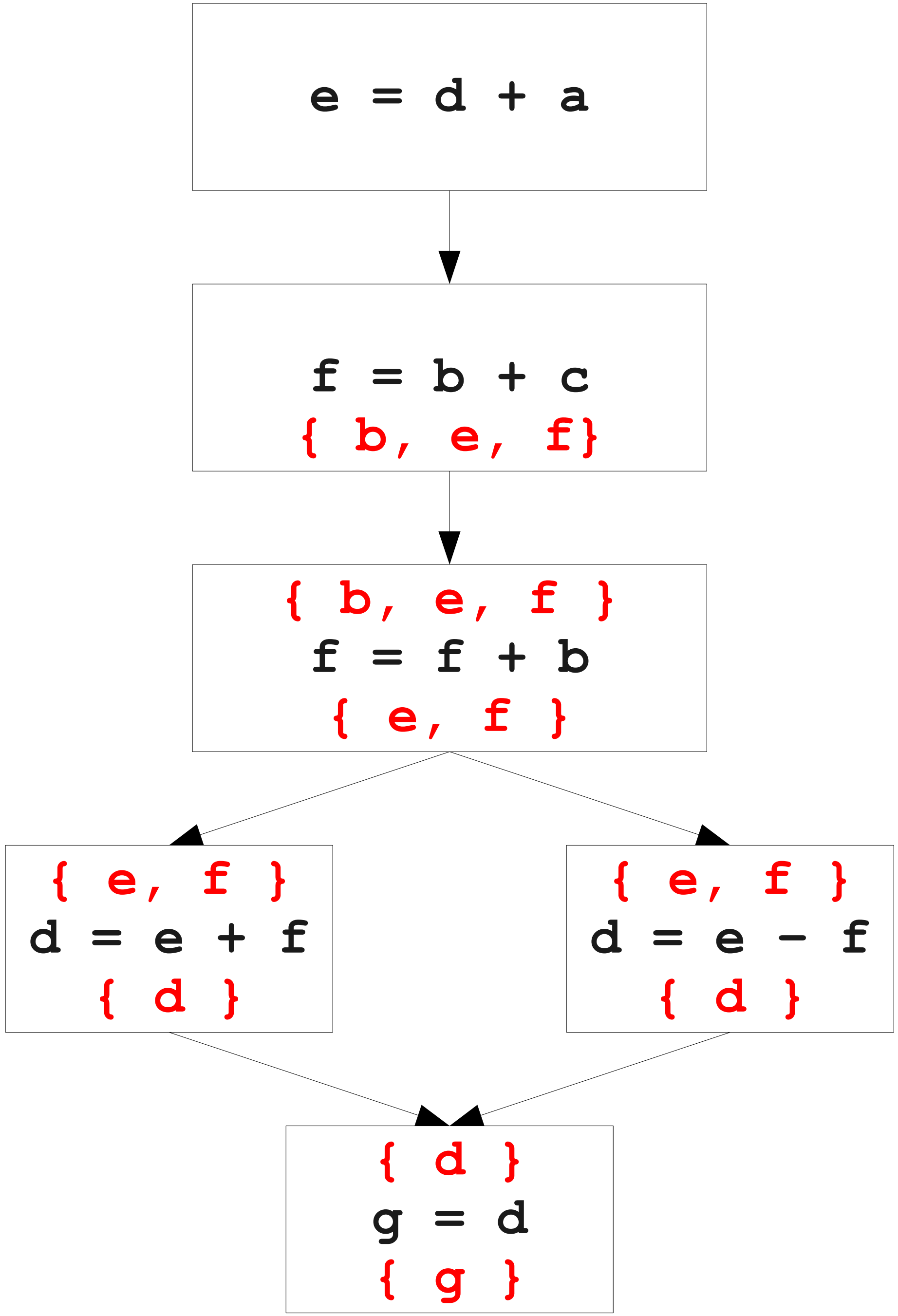
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



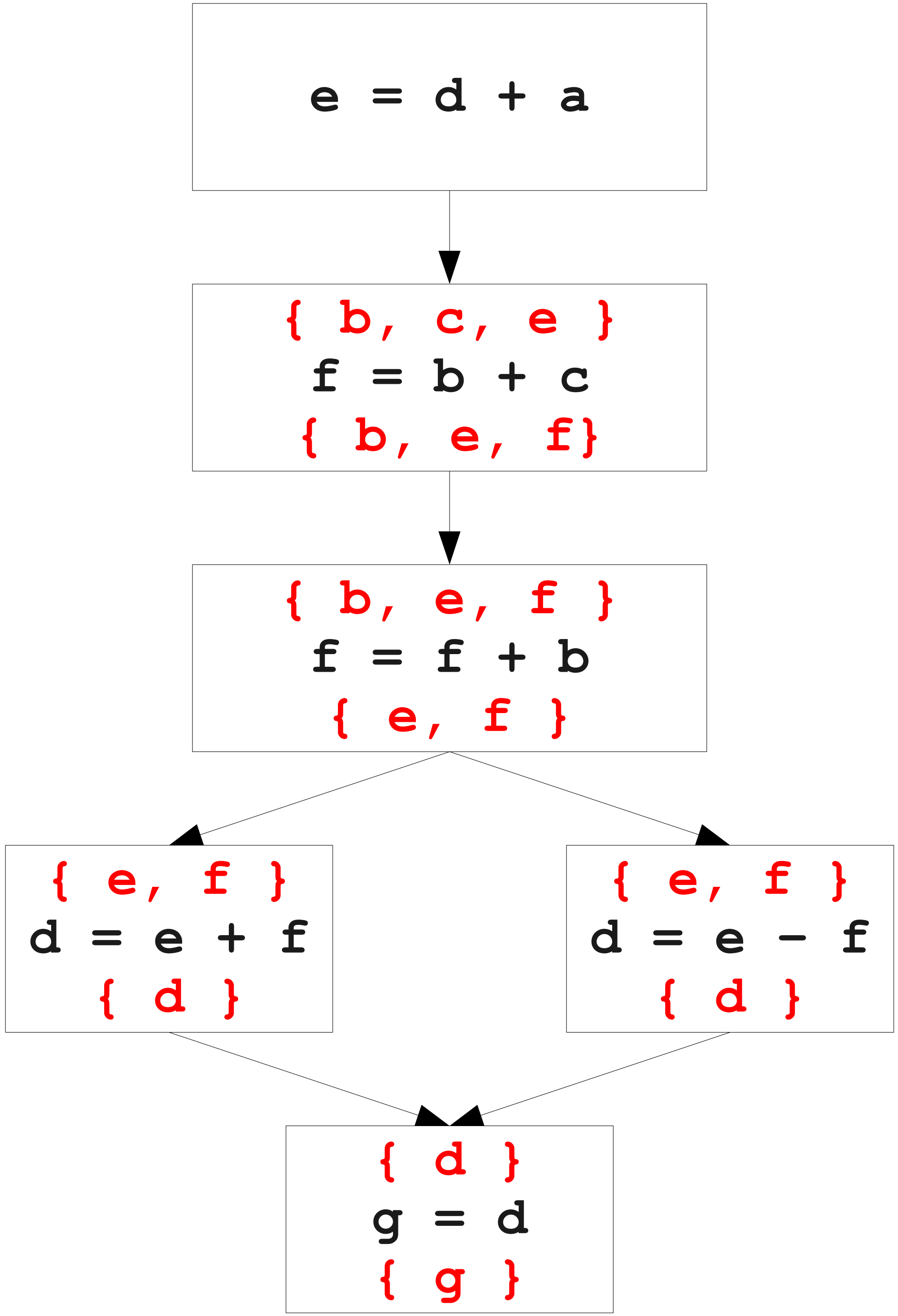
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



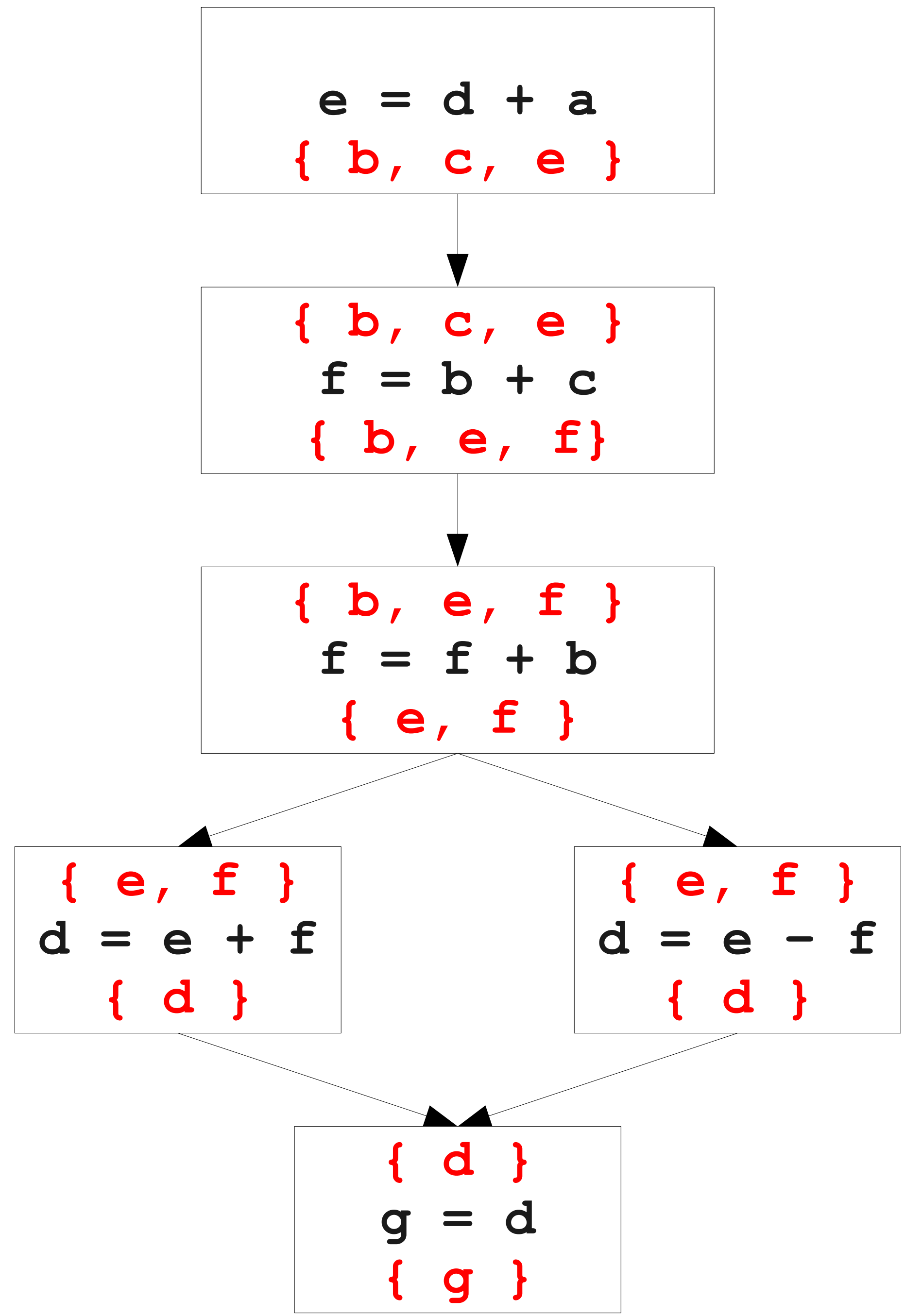
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



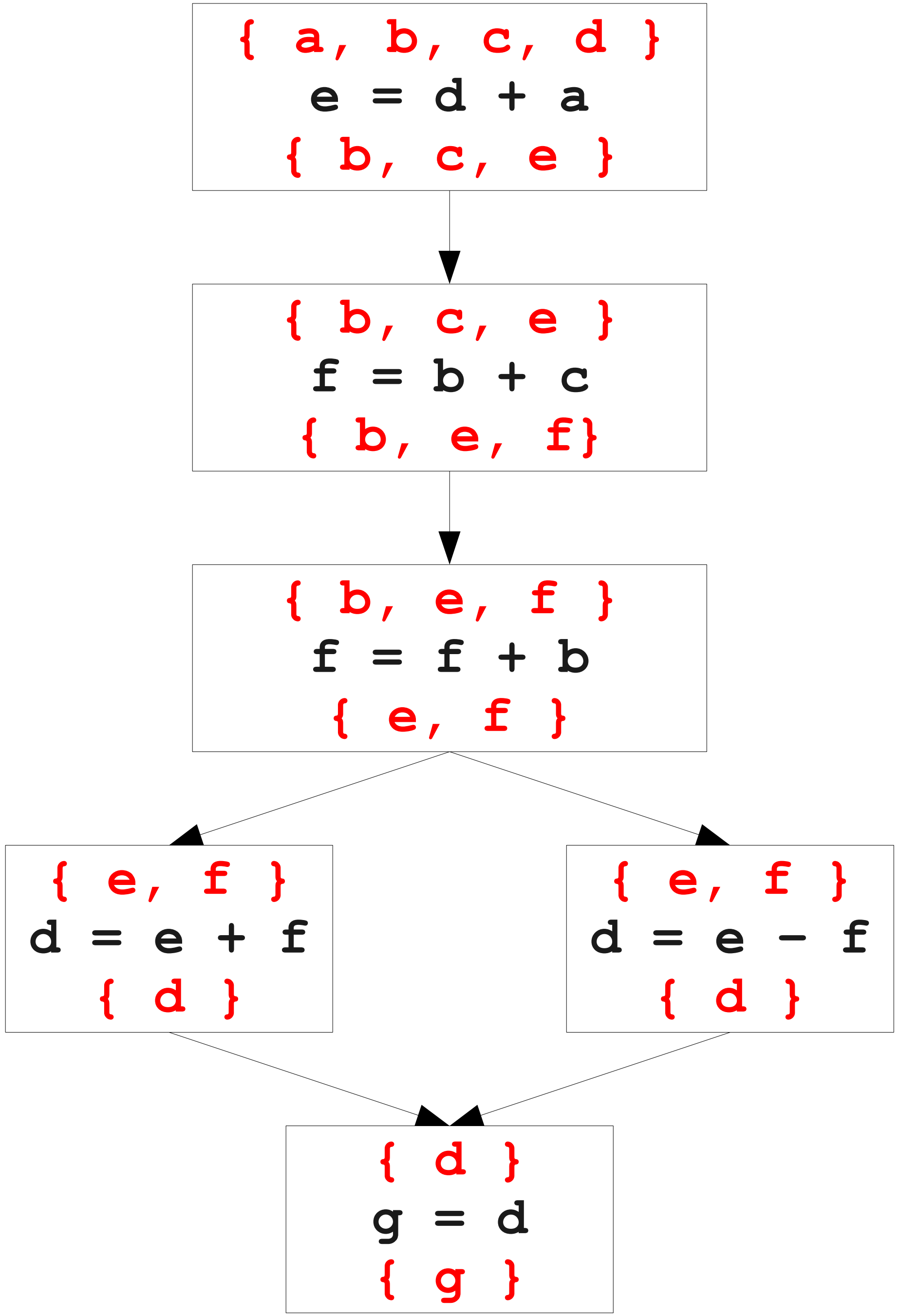
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



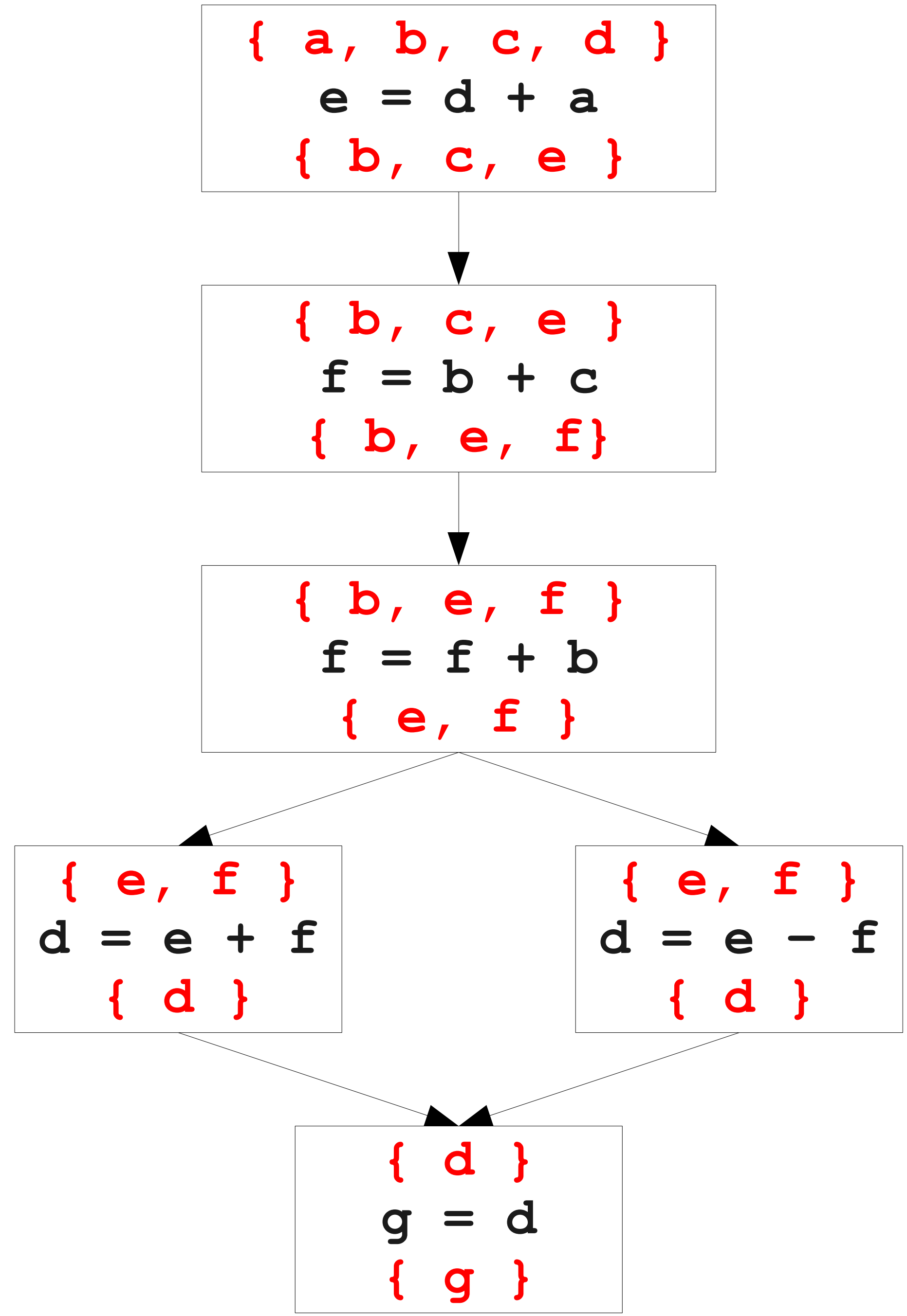
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



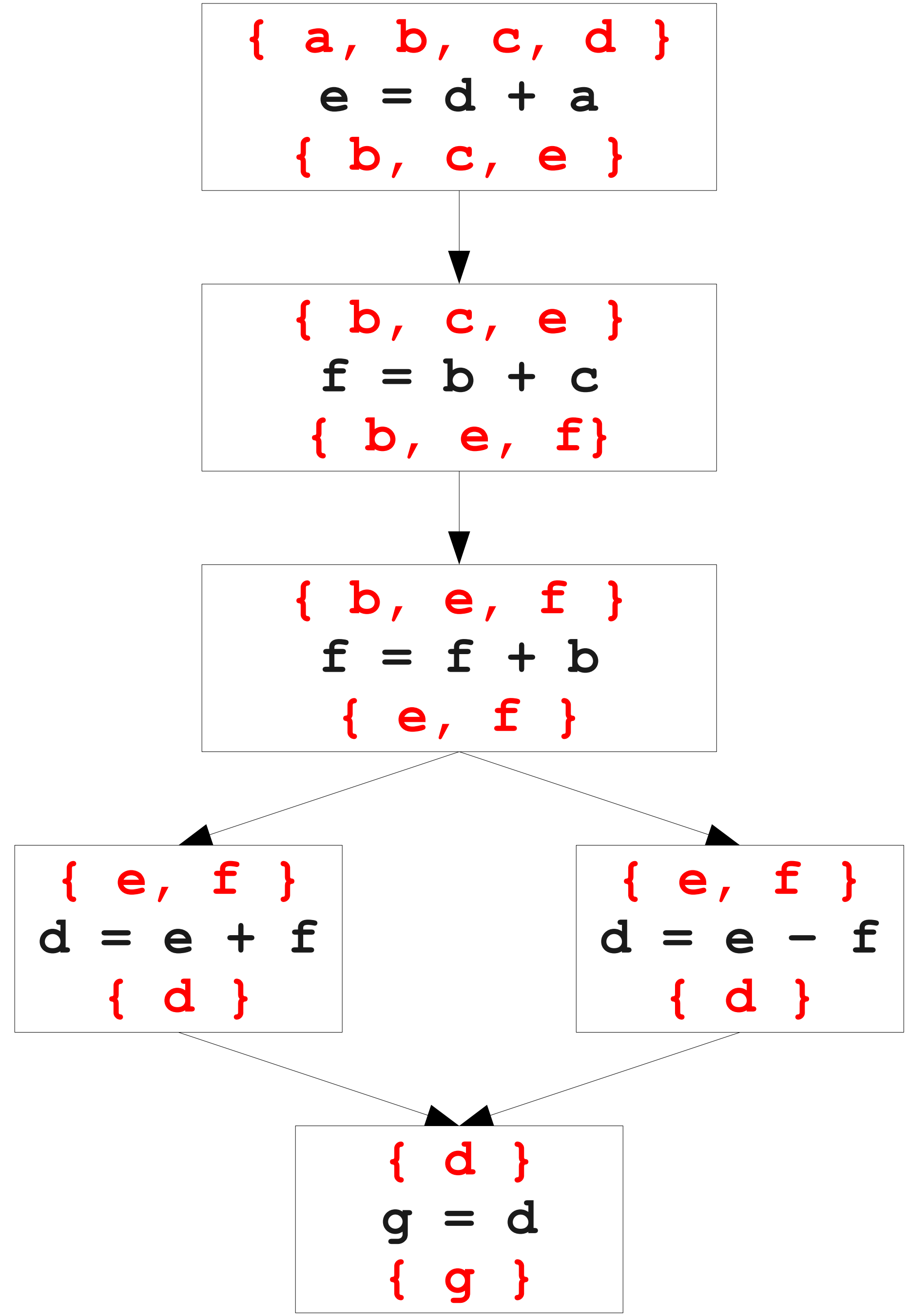
Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



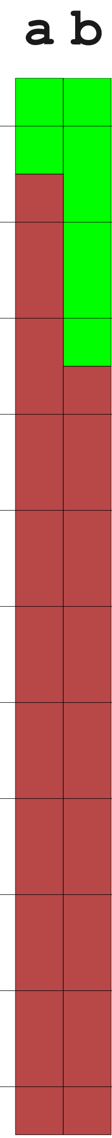
Live Ranges and Live Intervals

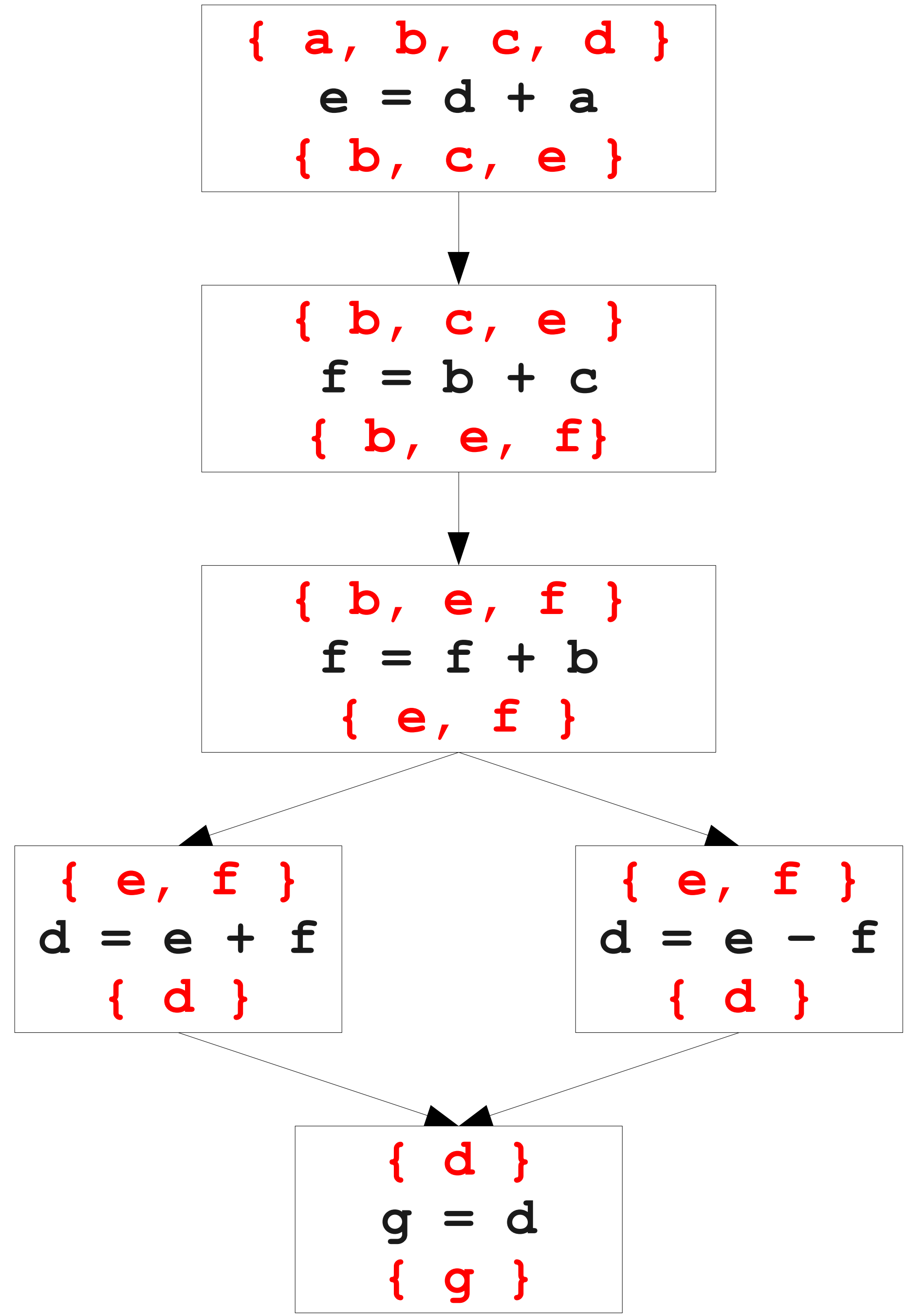
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



Live Ranges and Live Intervals

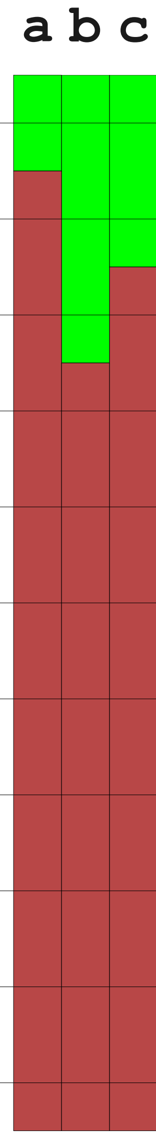
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

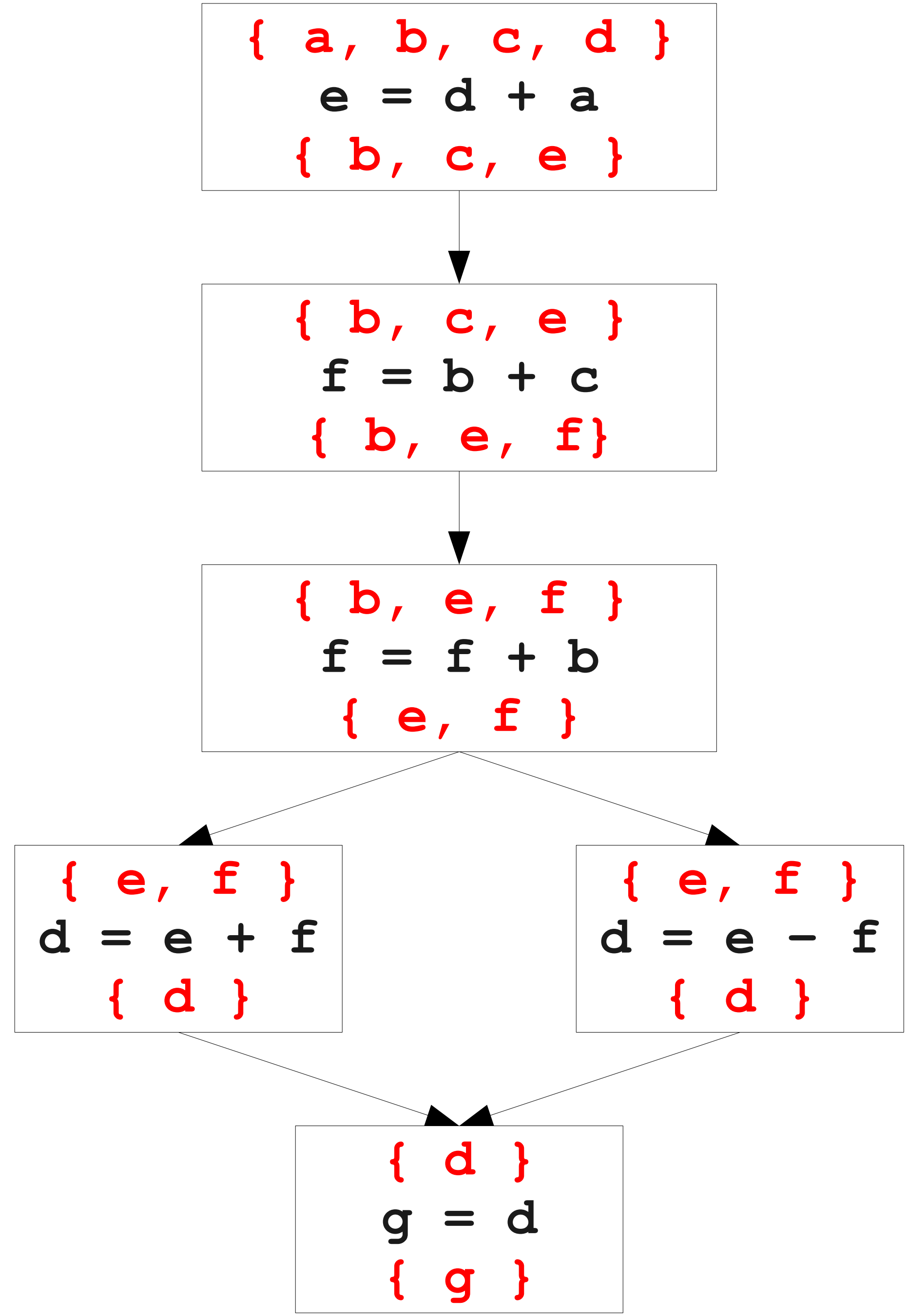




Live Ranges and Live Intervals

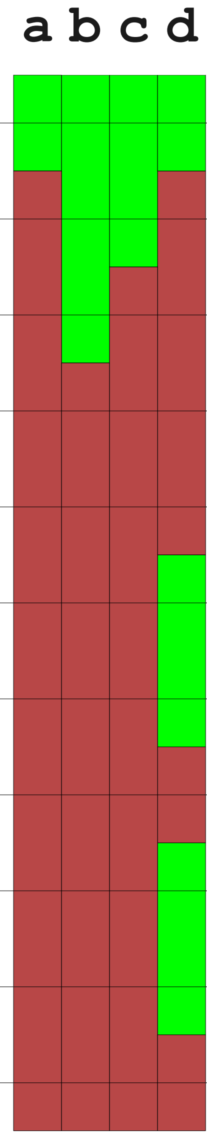
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

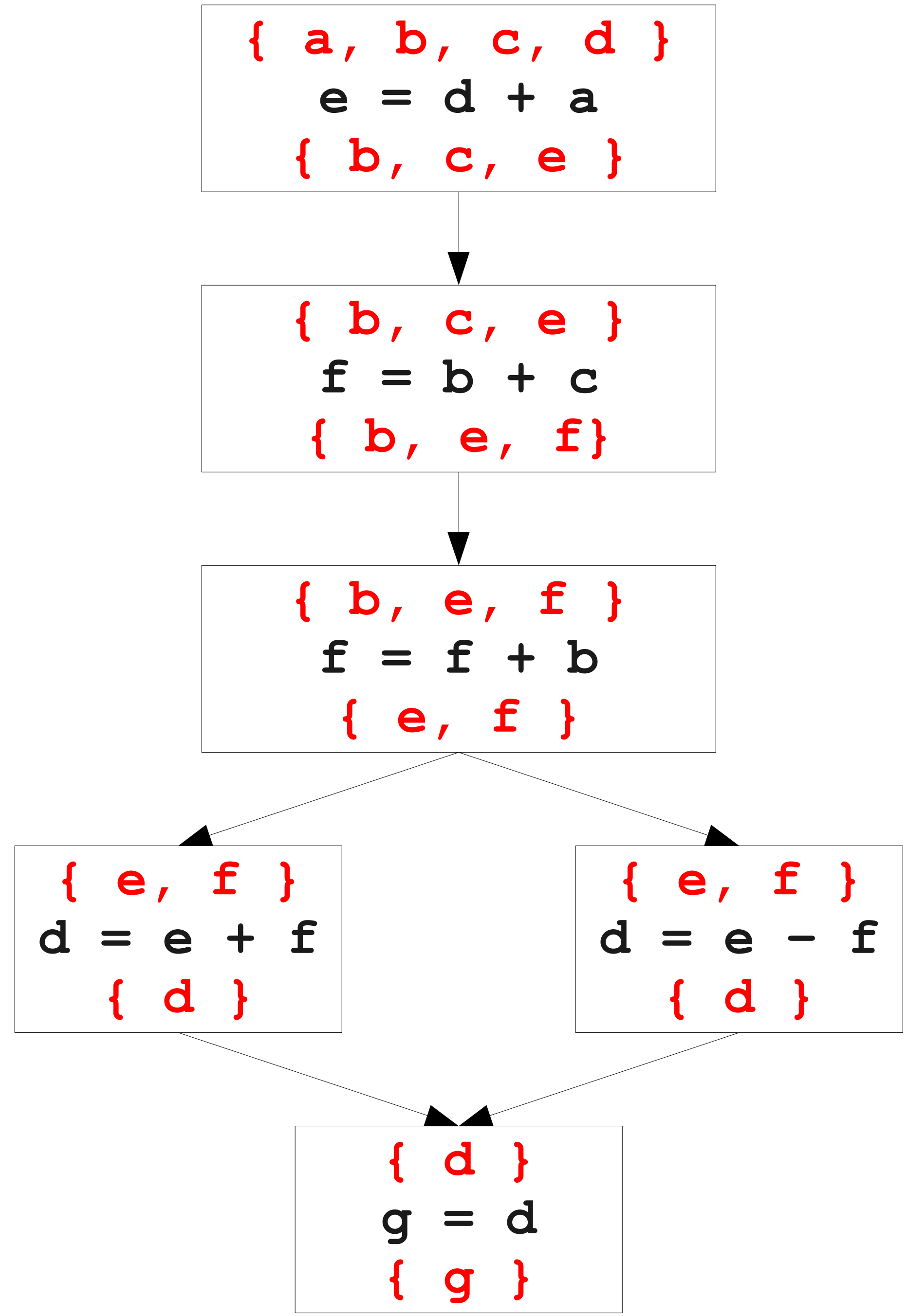




Live Ranges and Live Intervals

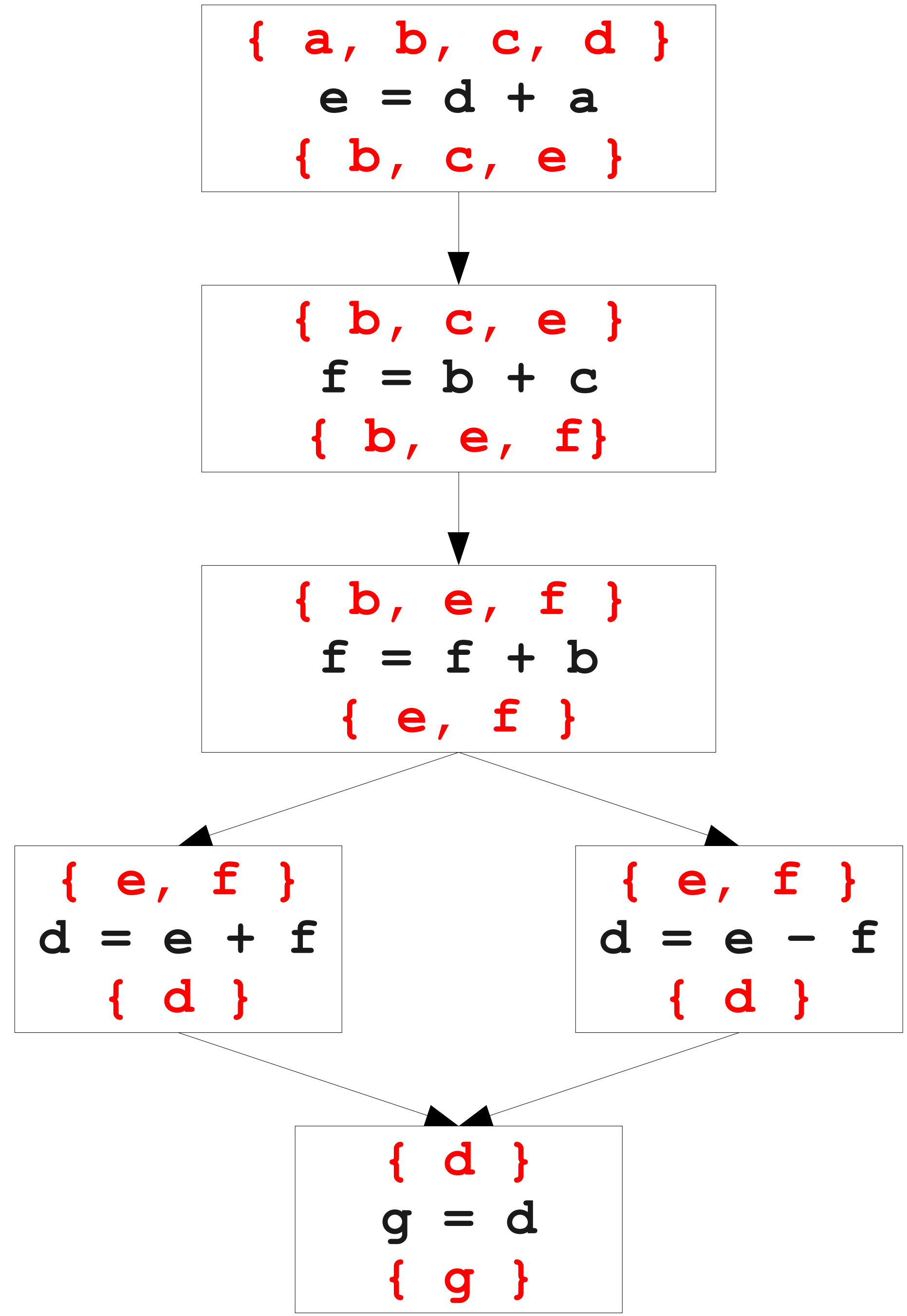
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



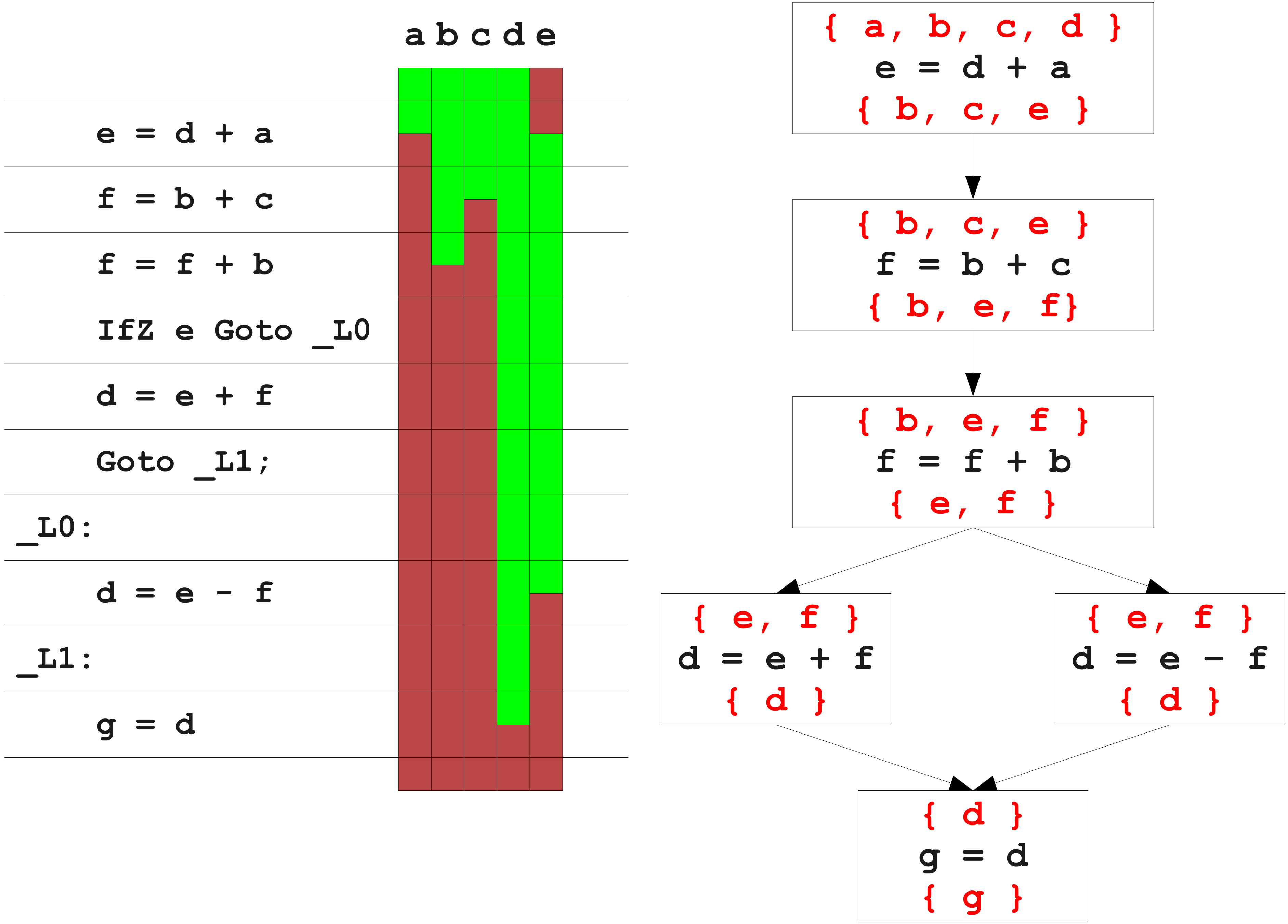


Live Ranges and Live Intervals

```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

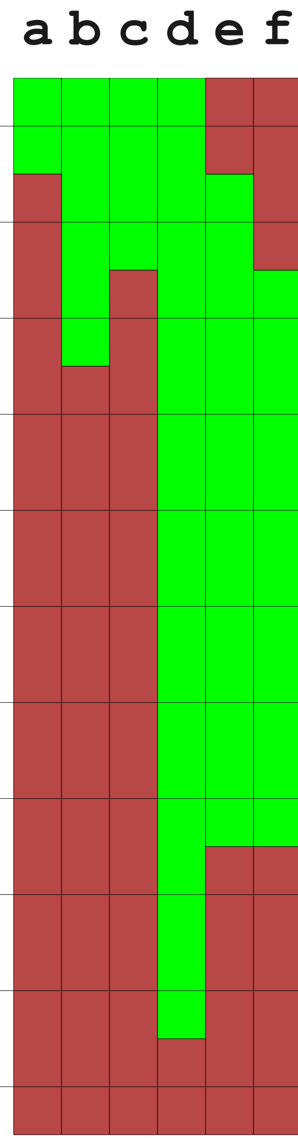


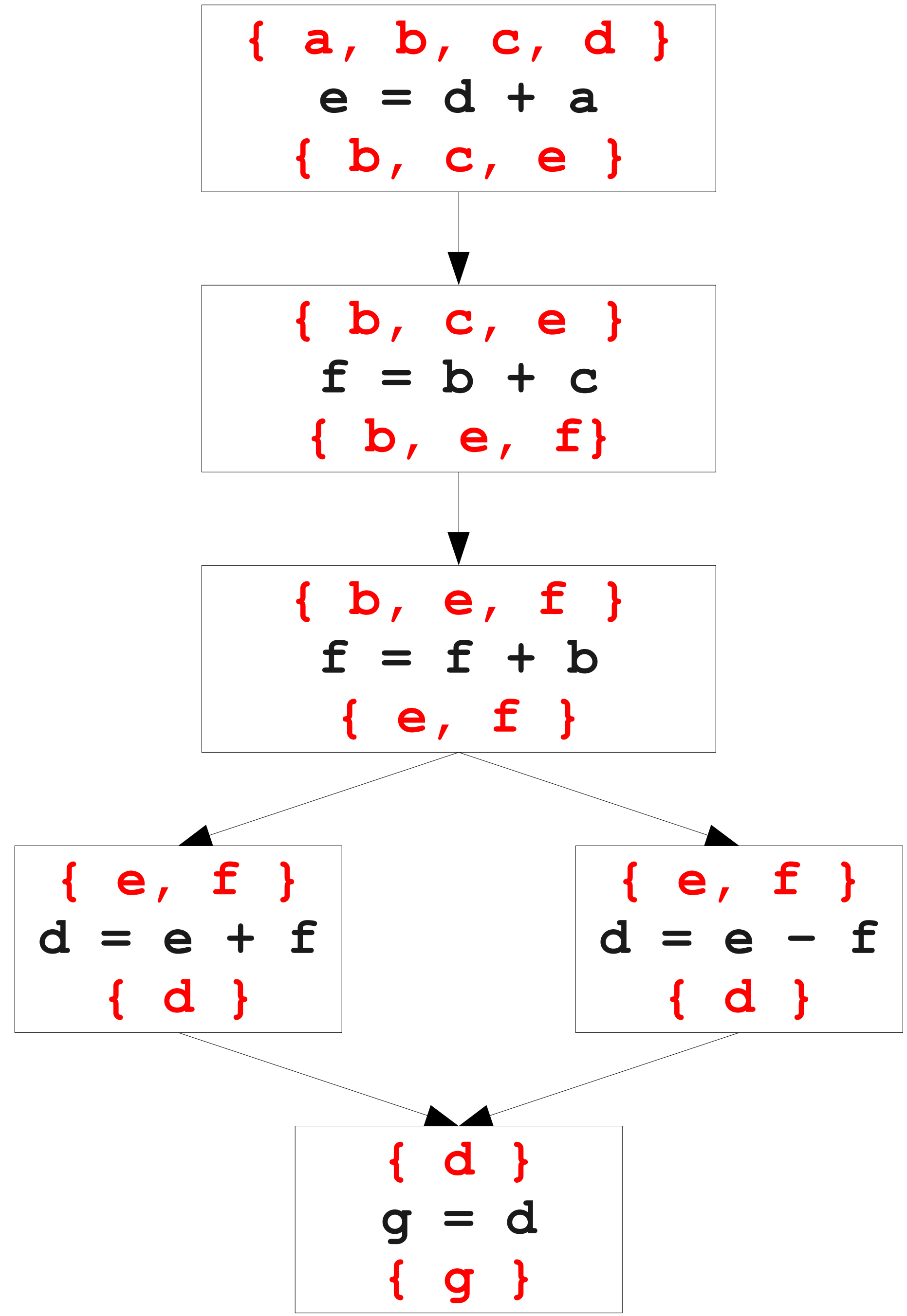
Live Ranges and Live Intervals



Live Ranges and Live Intervals

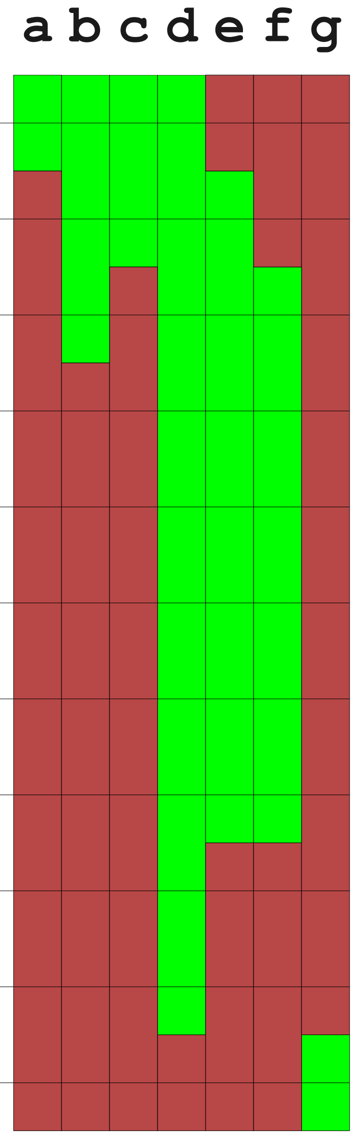
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

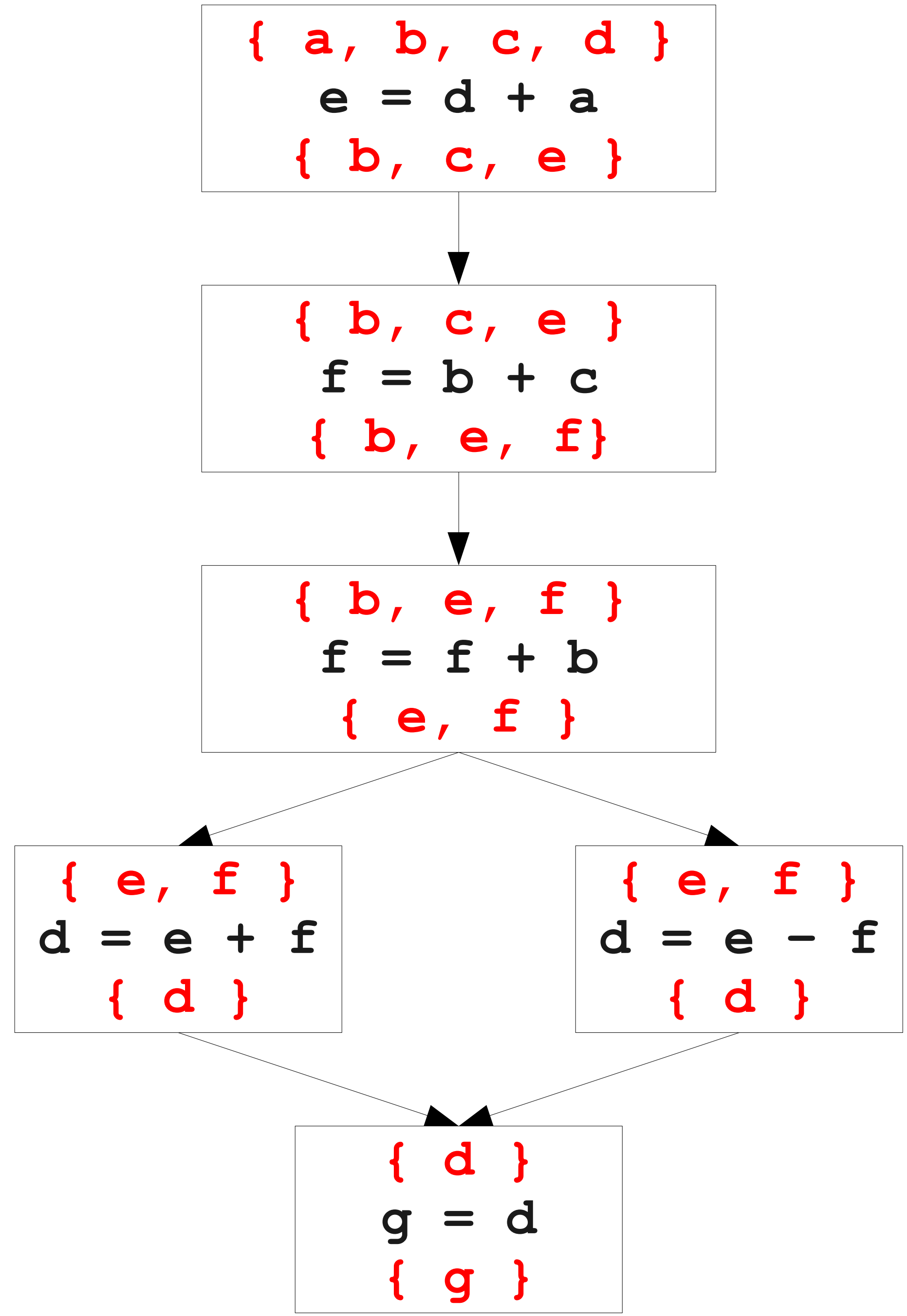




Live Ranges and Live Intervals

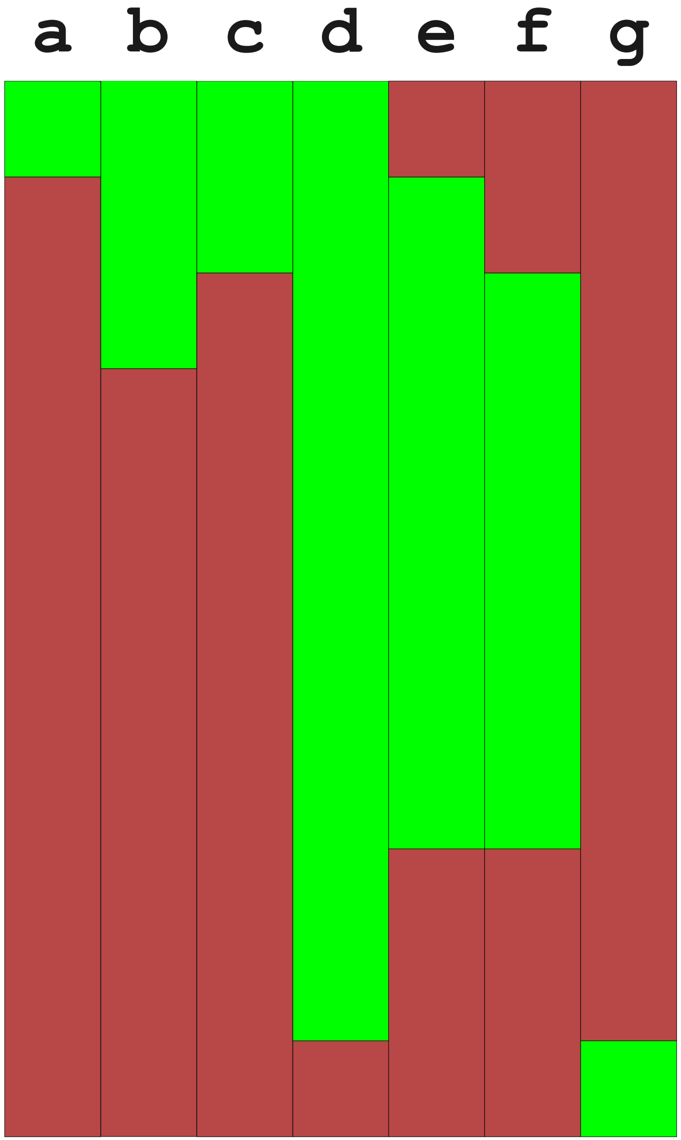
```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```



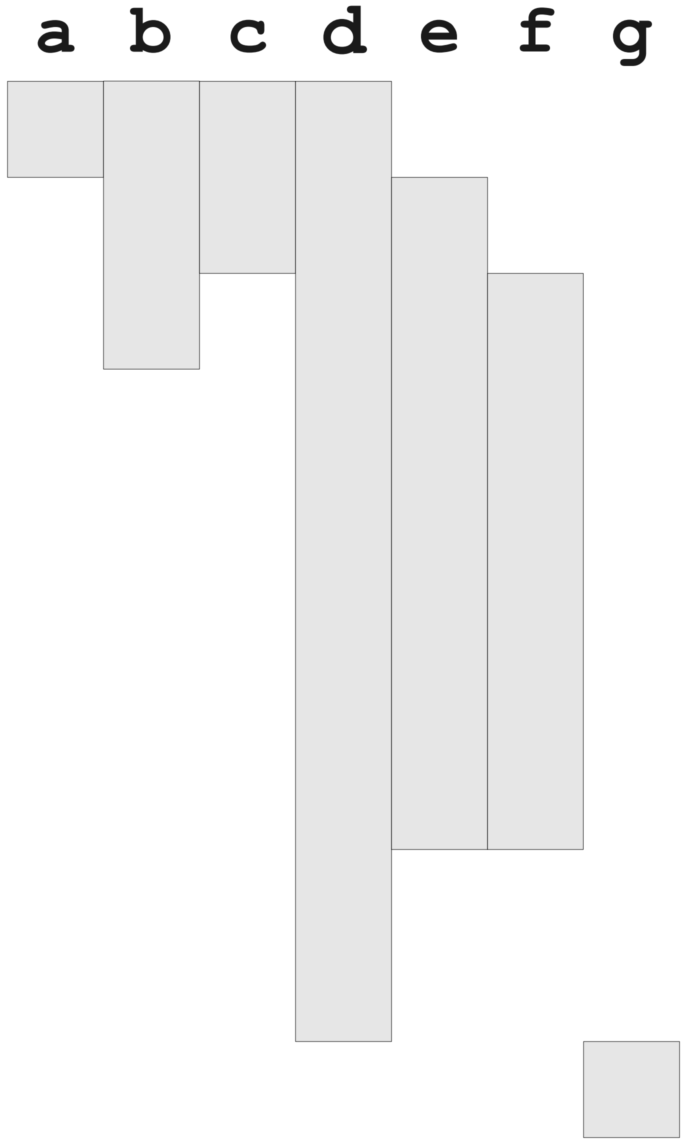


Register Allocation with Live Intervals

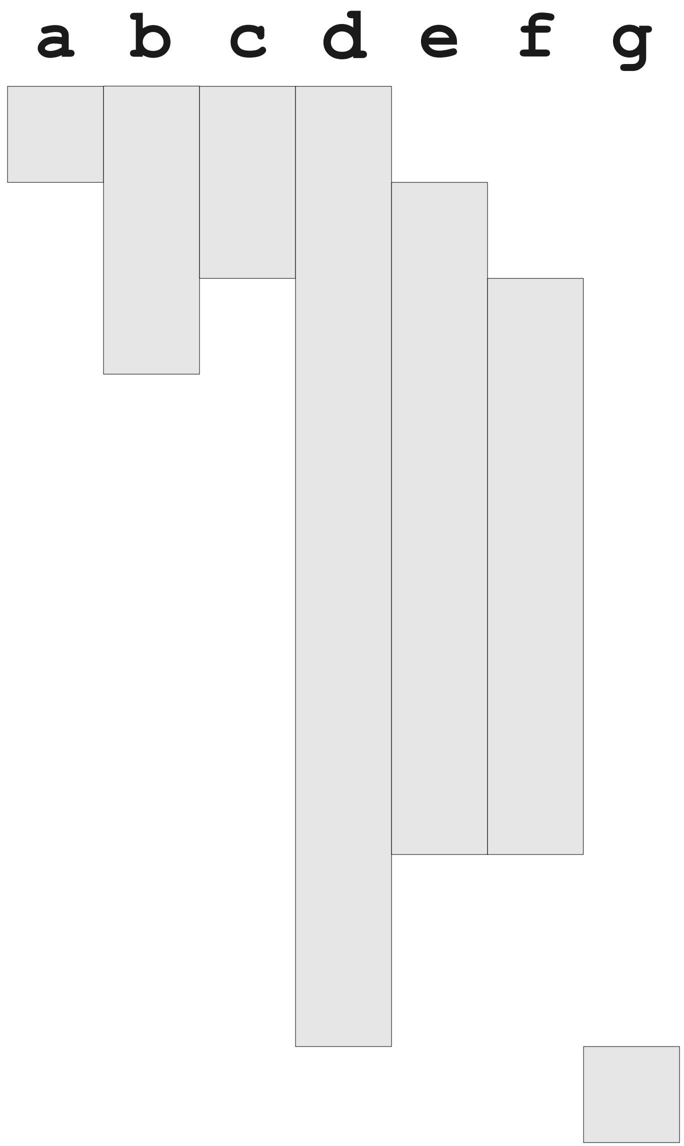
- Given the live intervals for all the variables in the program, we can allocate registers using a simple greedy algorithm.
- Idea: Track which registers are free at each point.
- When a live interval begins, give that variable a free register.
- When a live interval ends, the register is once again free.
- We can't always fit everything into a register; we'll see what do to in a minute.



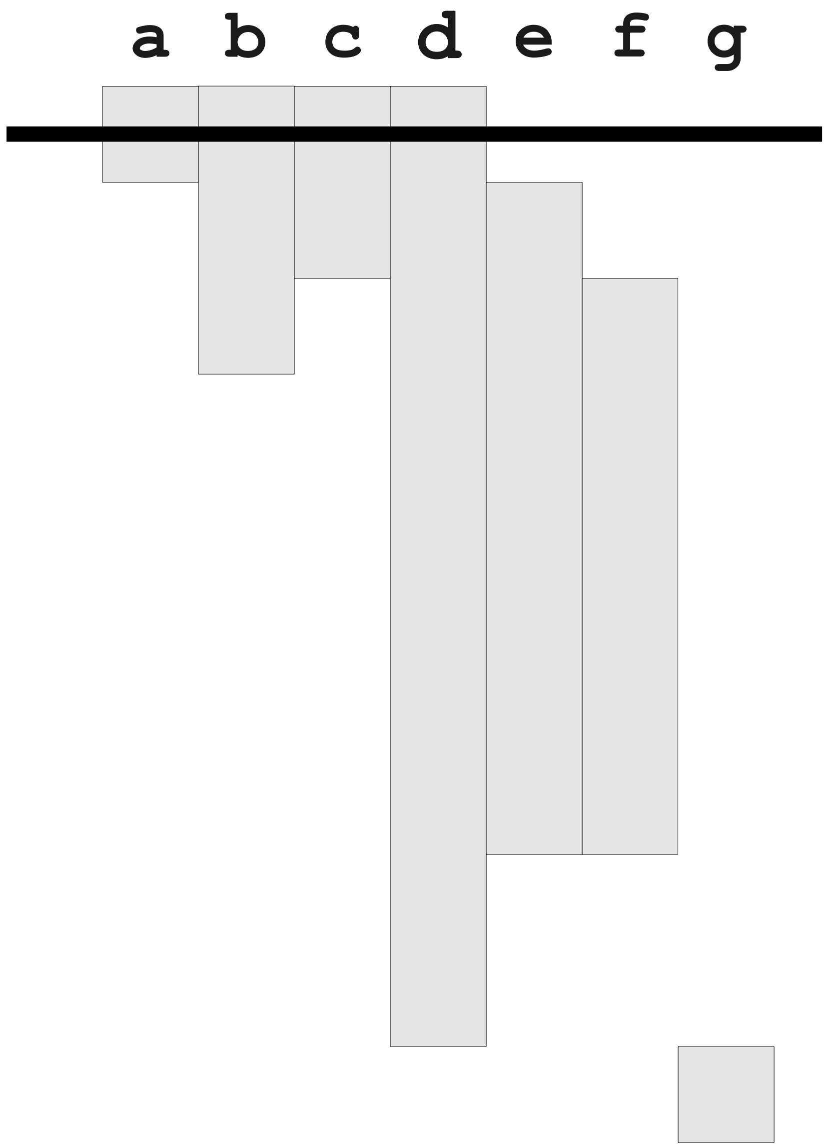
Register Allocation with Live Intervals



Register Allocation with Live Intervals



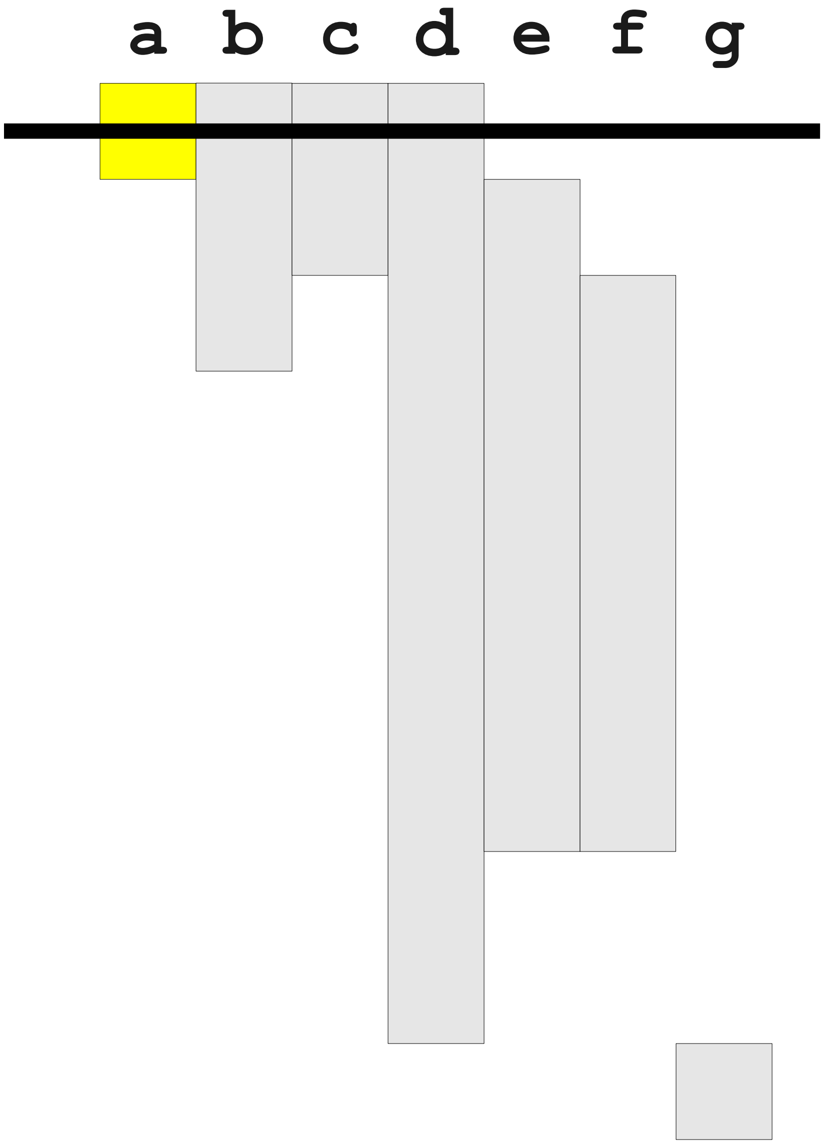
Register Allocation with Live Intervals



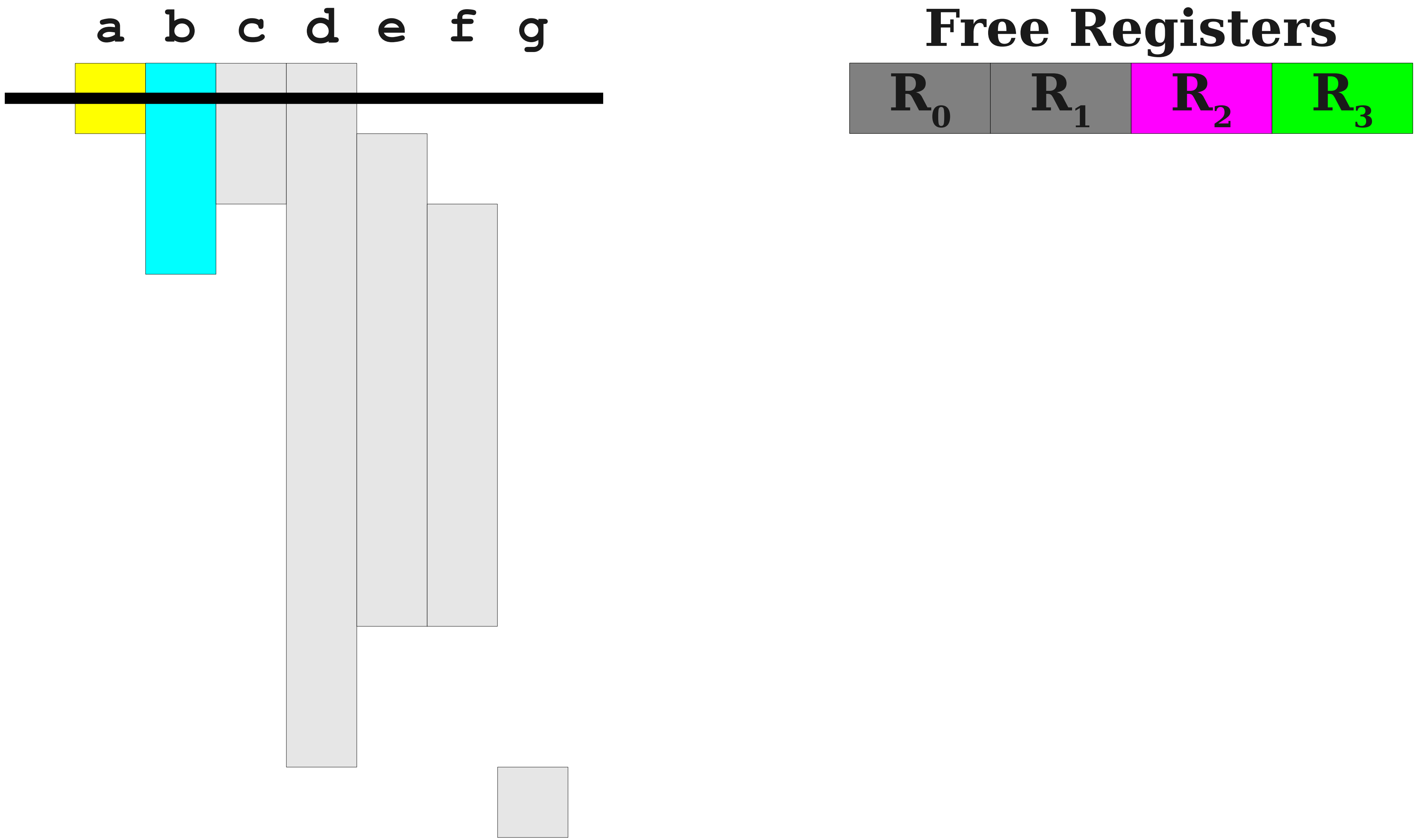
Free Registers

R_0	R_1	R_2	R_3

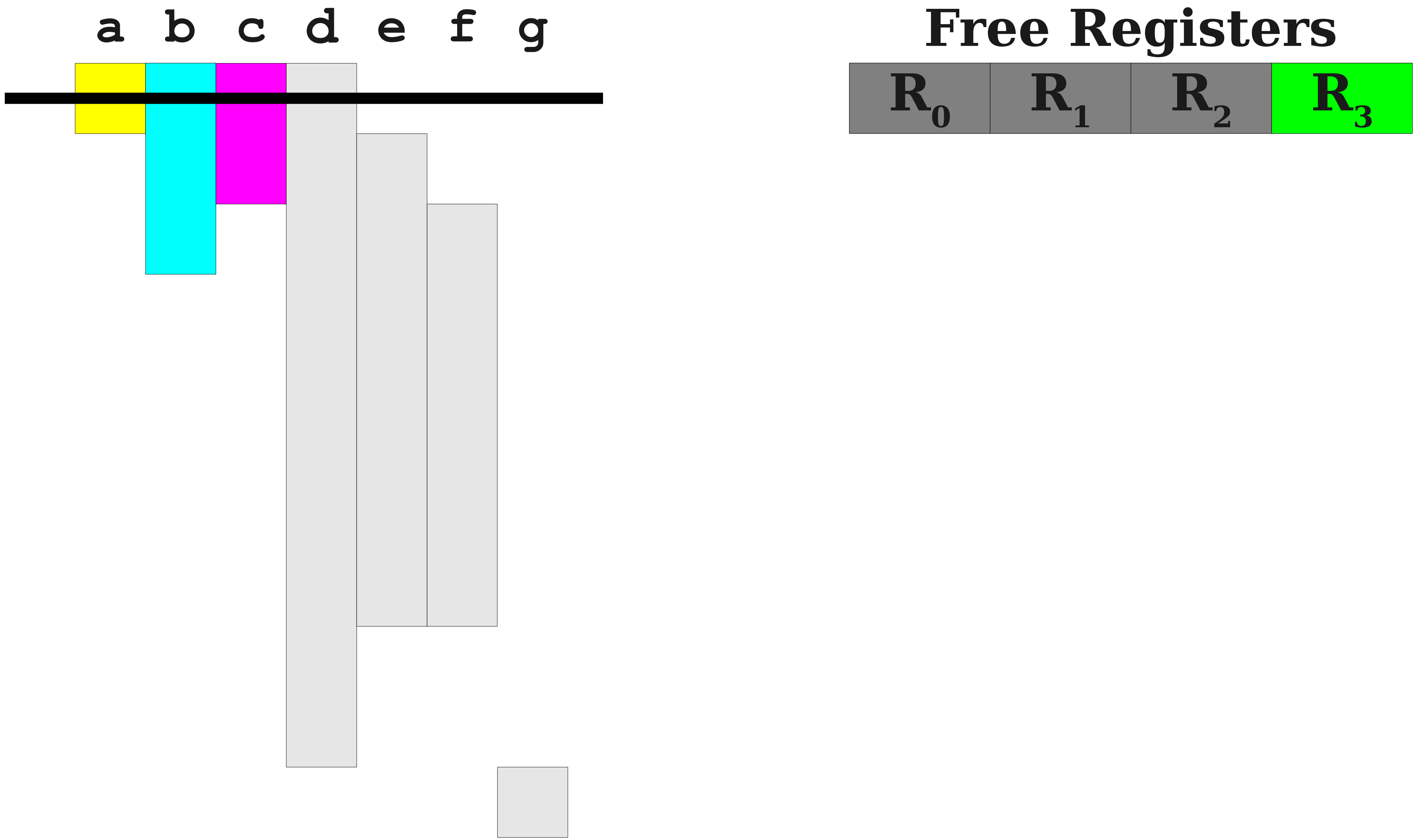
Register Allocation with Live Intervals



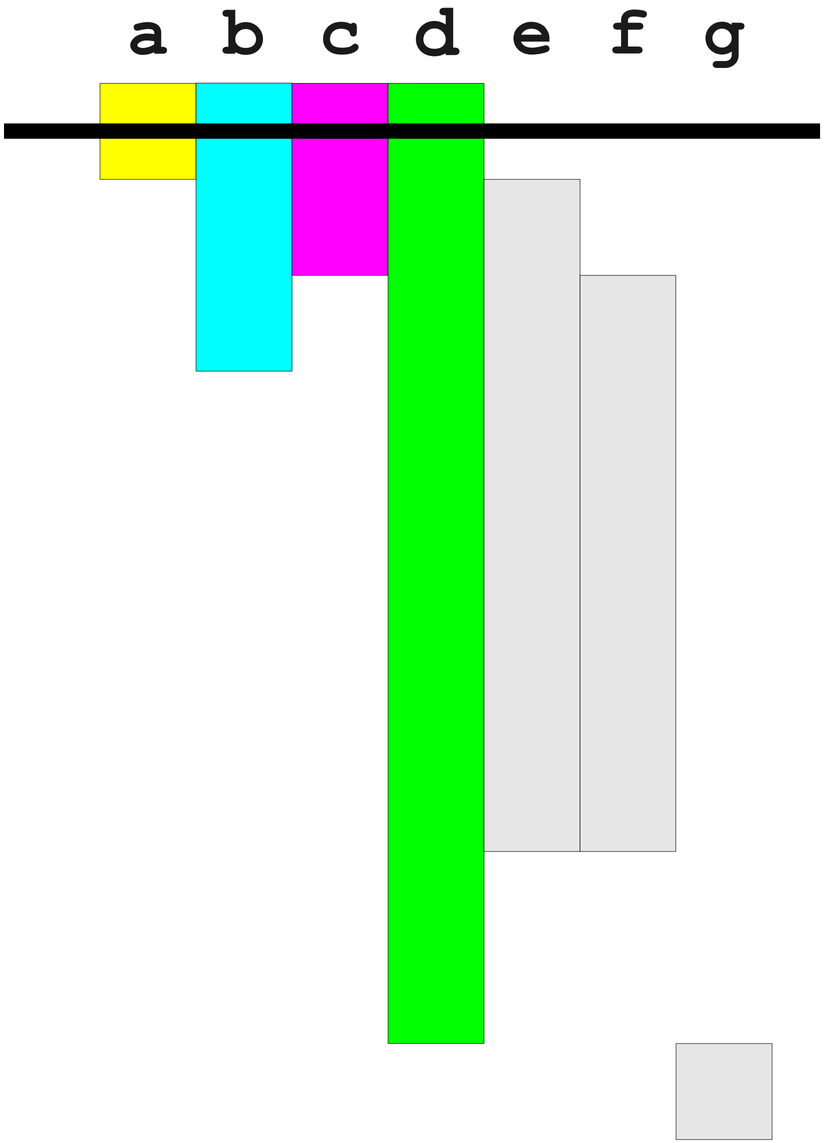
Register Allocation with Live Intervals



Register Allocation with Live Intervals



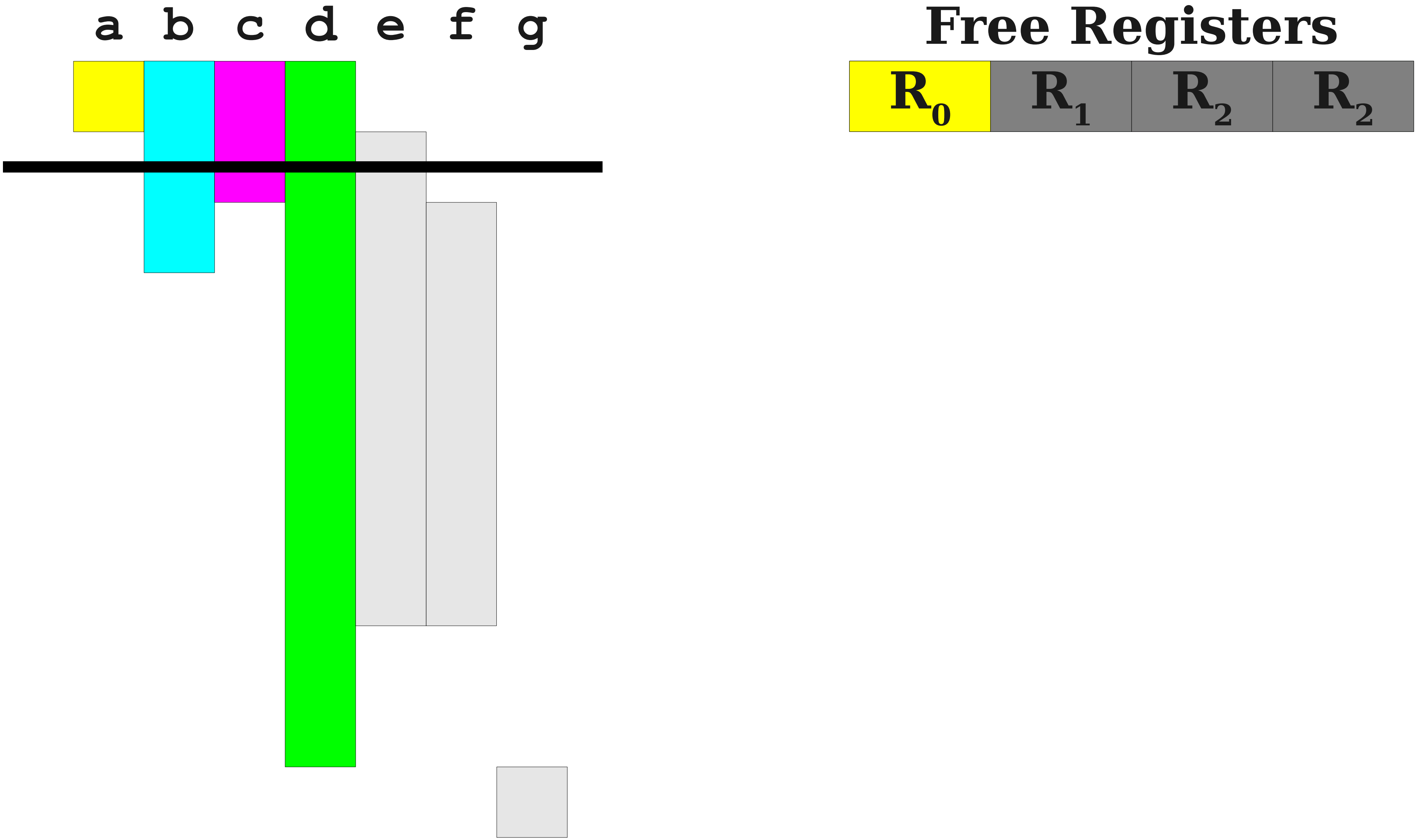
Register Allocation with Live Intervals



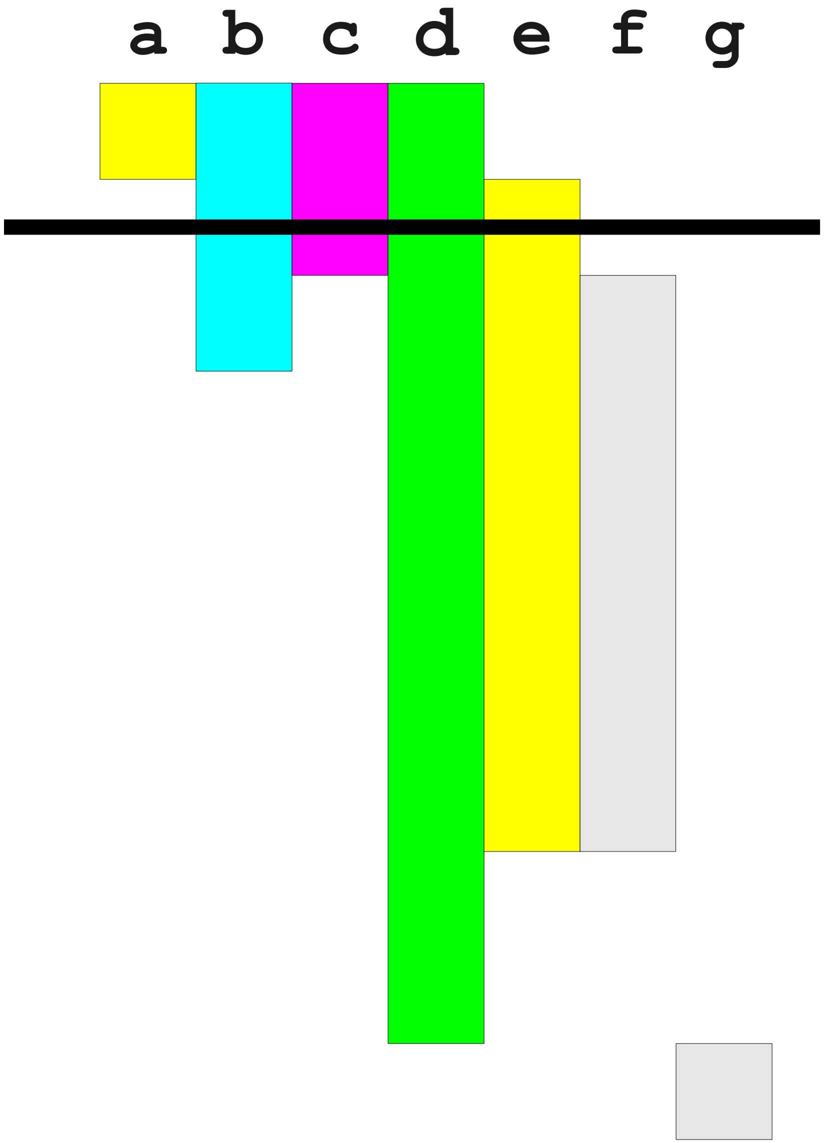
Free Registers

R_0	R_1	R_2	R_2

Register Allocation with Live Intervals



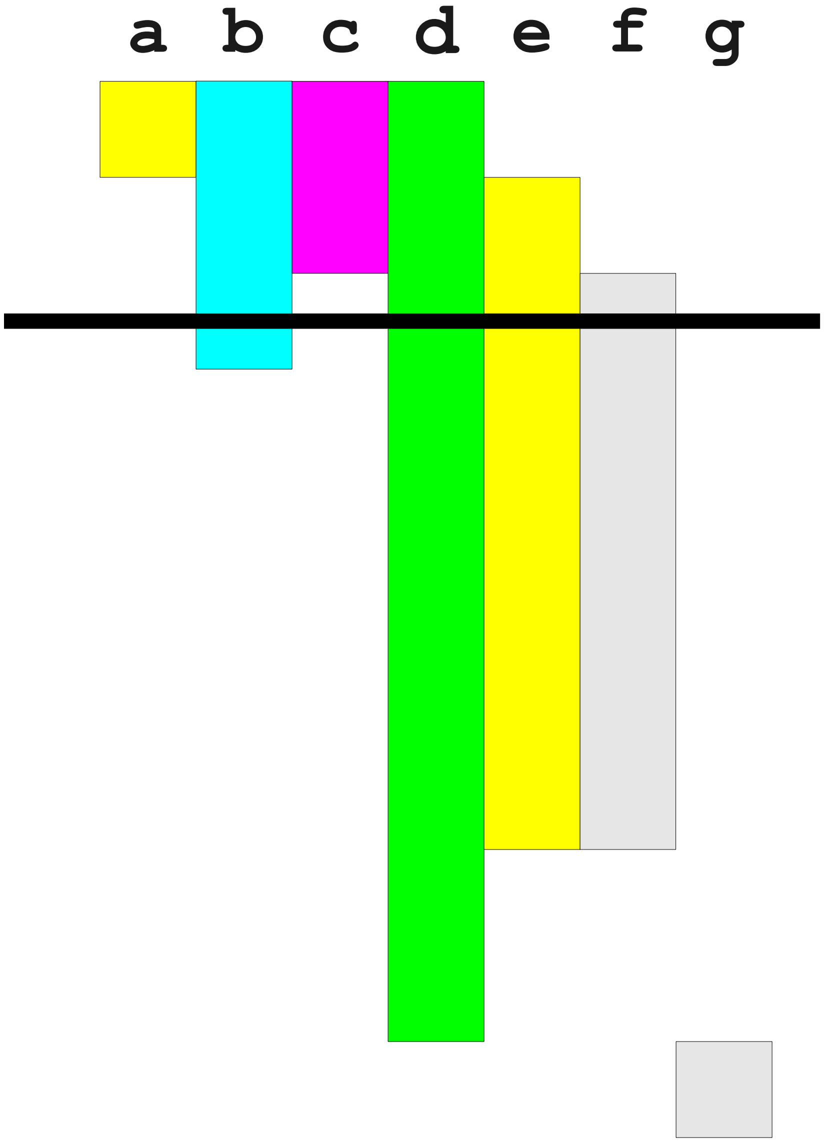
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2	R_2

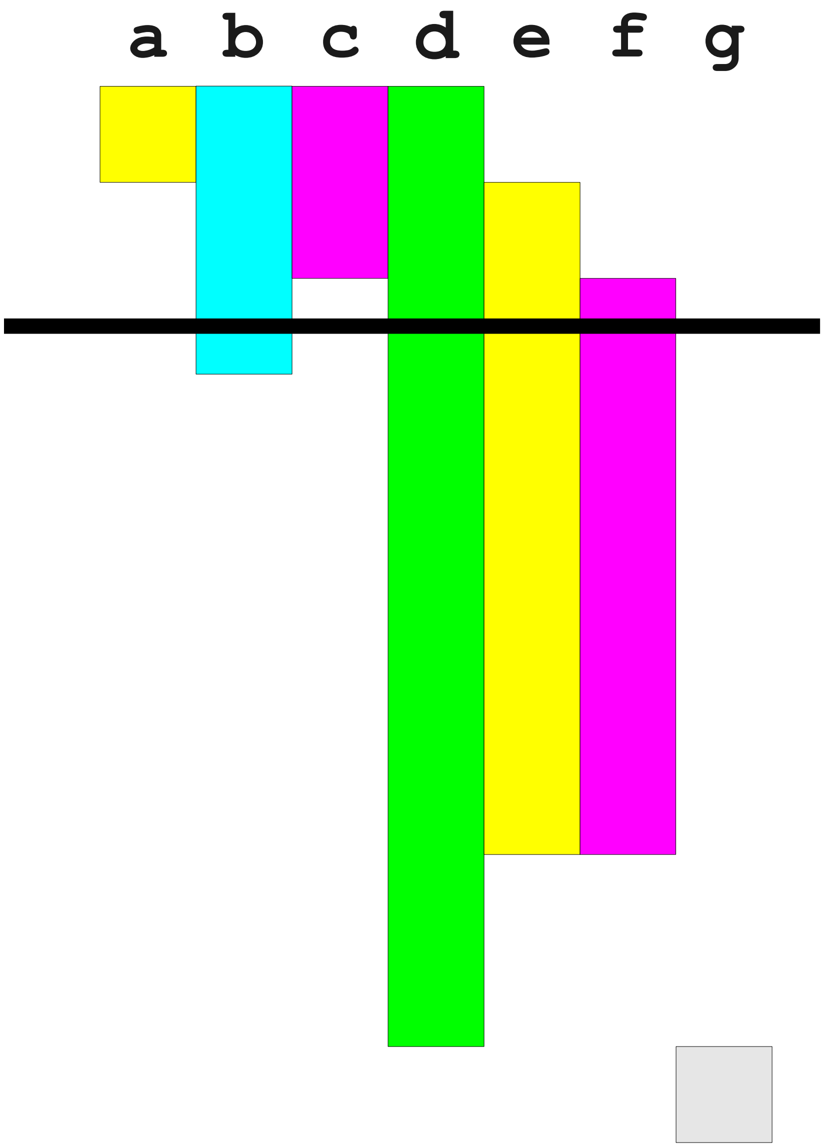
Register Allocation with Live Intervals



Free Registers

R_0	R_1	R_2	R_2

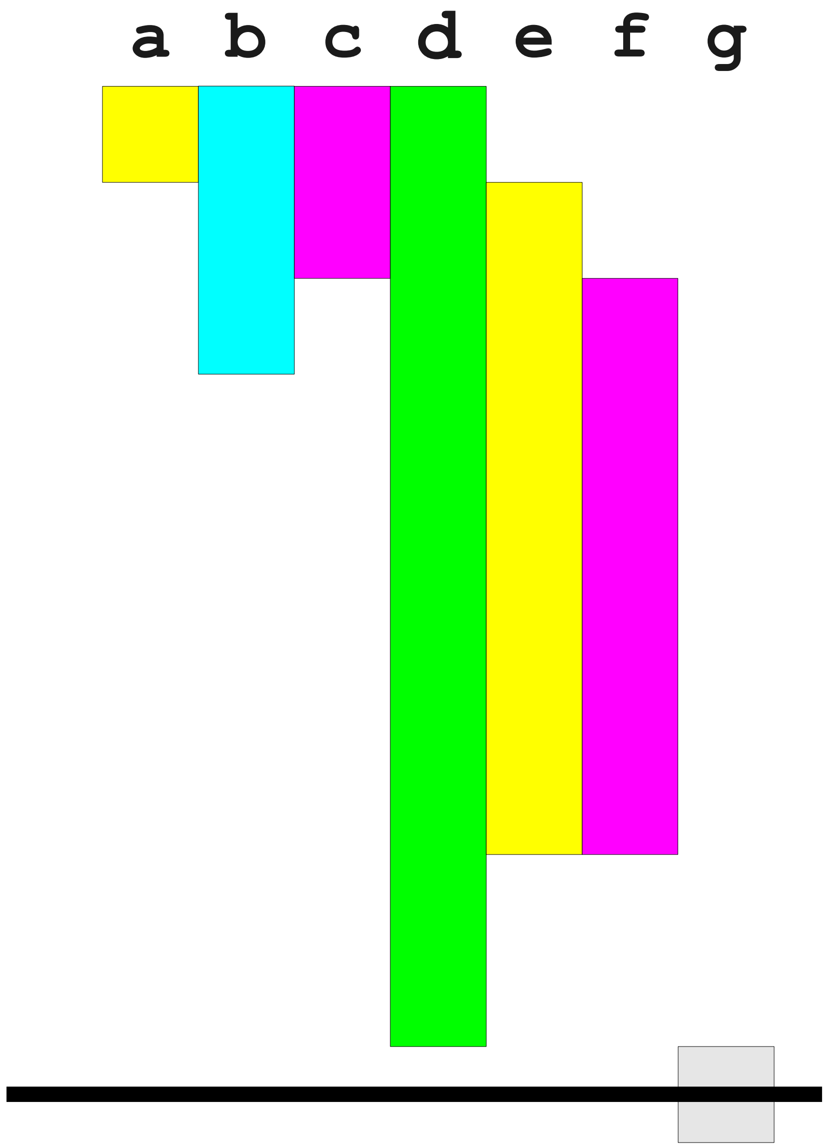
Register Allocation with Live Intervals



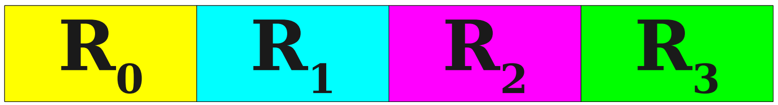
Free Registers

R_0	R_1	R_2	R_2

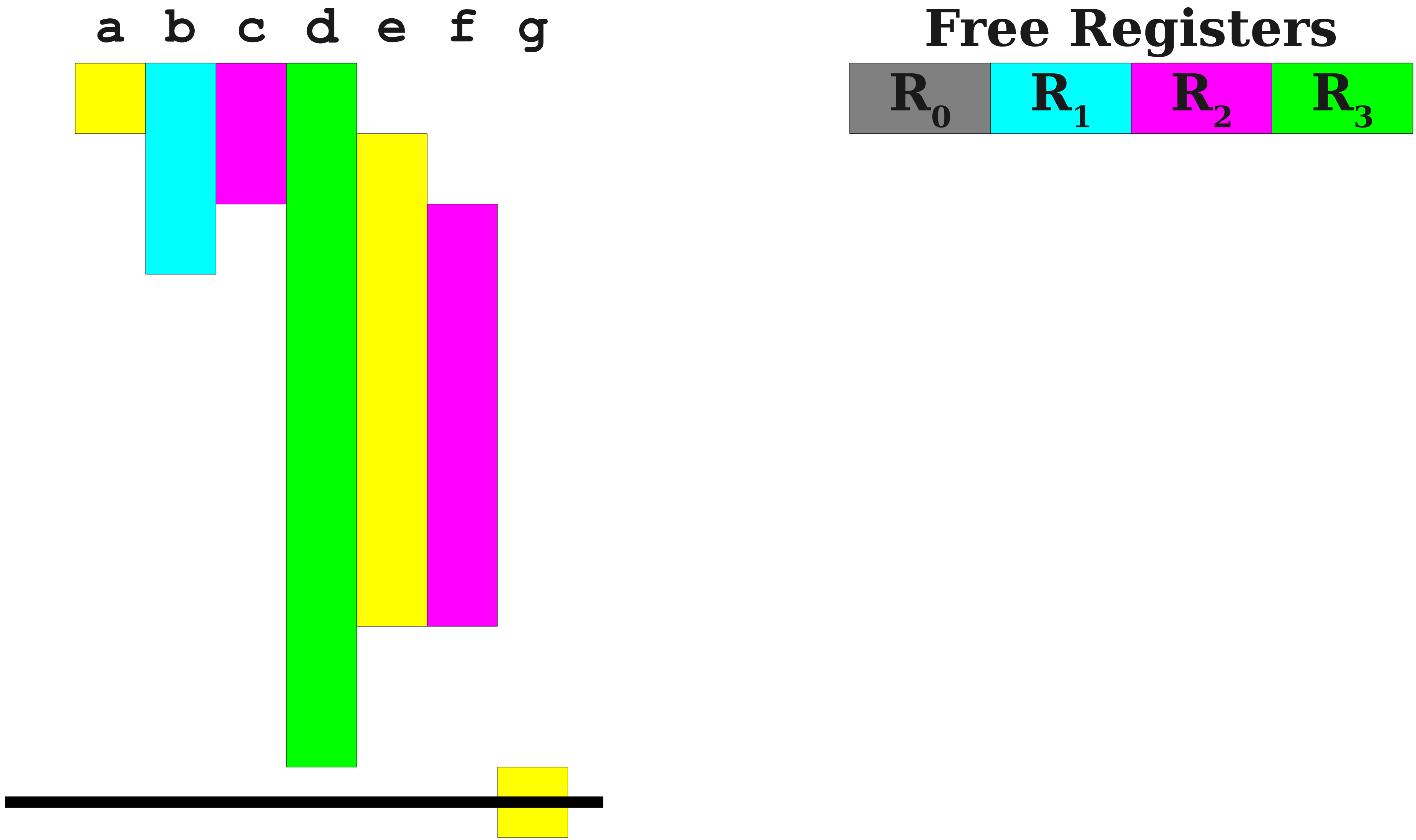
Register Allocation with Live Intervals



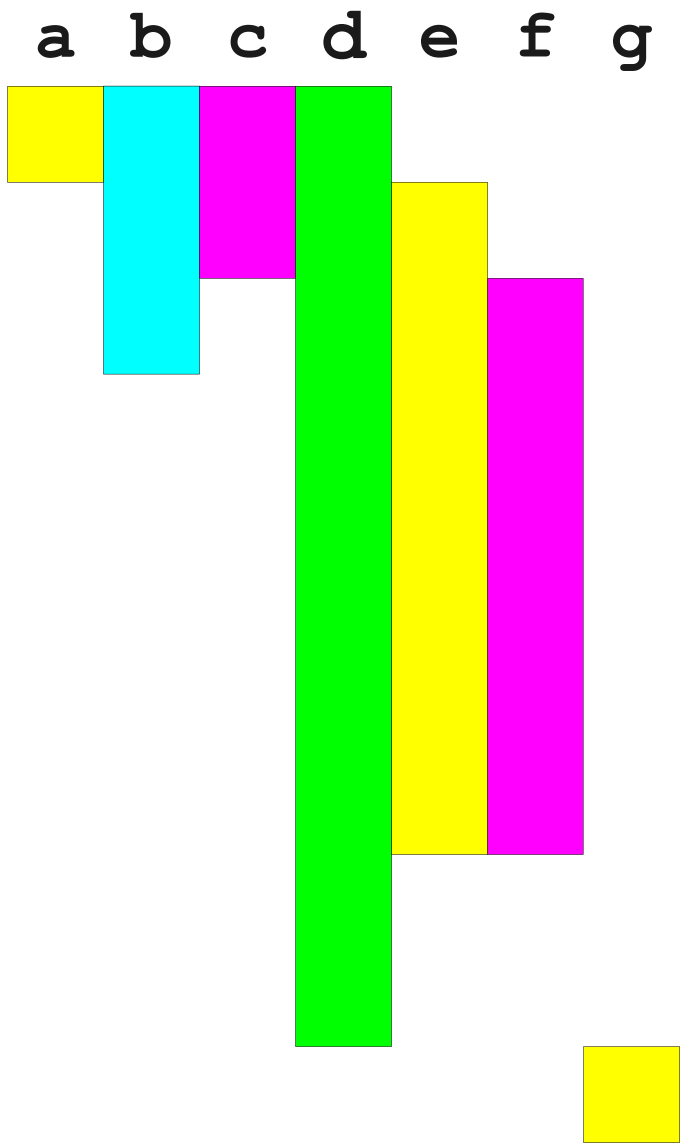
Free Registers



Register Allocation with Live Intervals



Register Allocation with Live Intervals

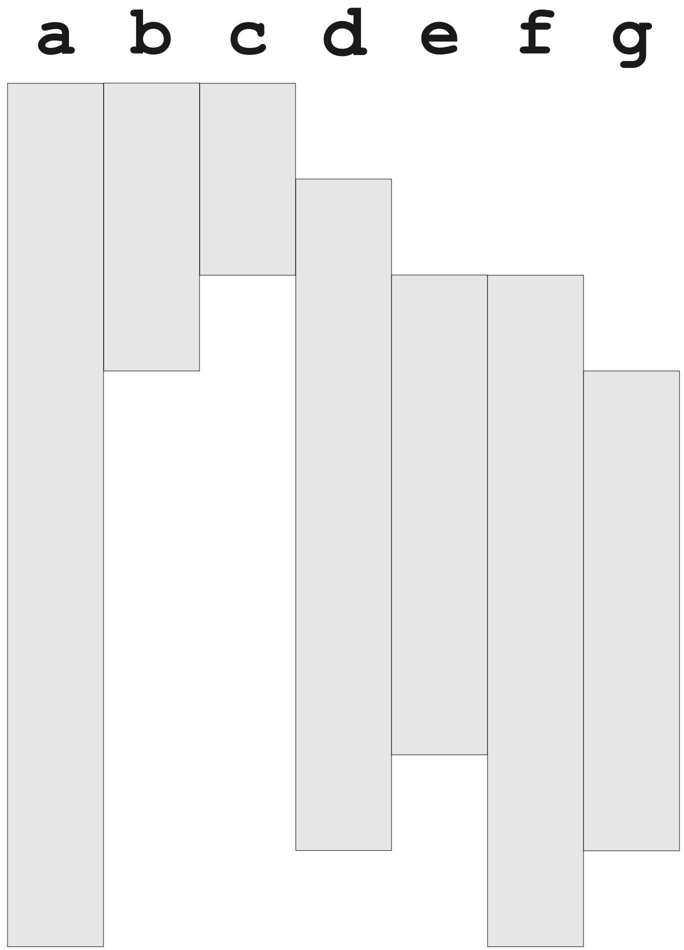


Free Registers

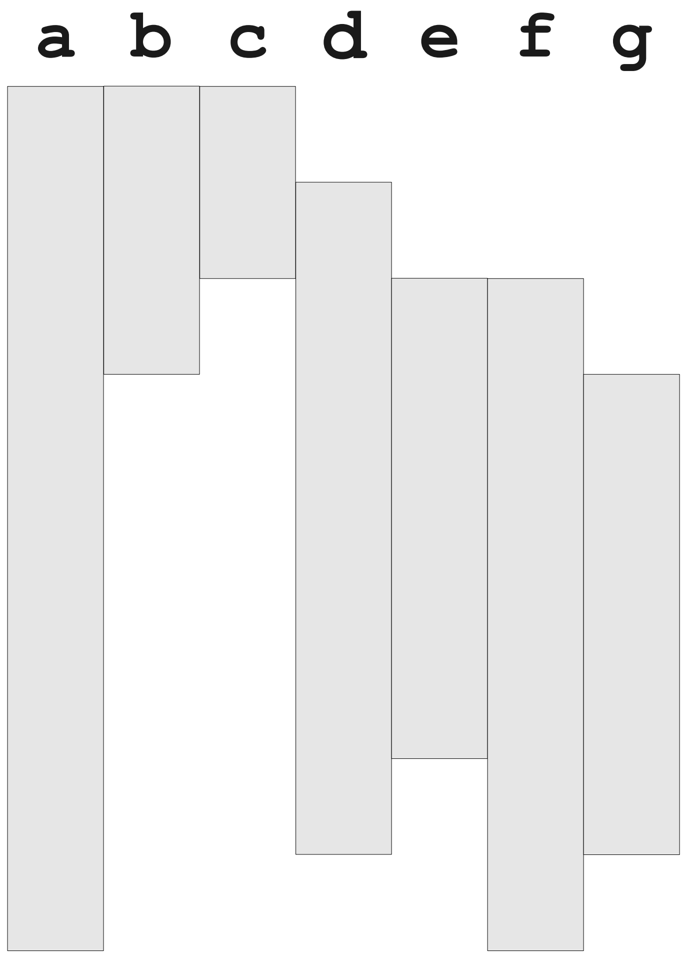
R_0	R_1	R_2	R_3

Another Example

Another Example



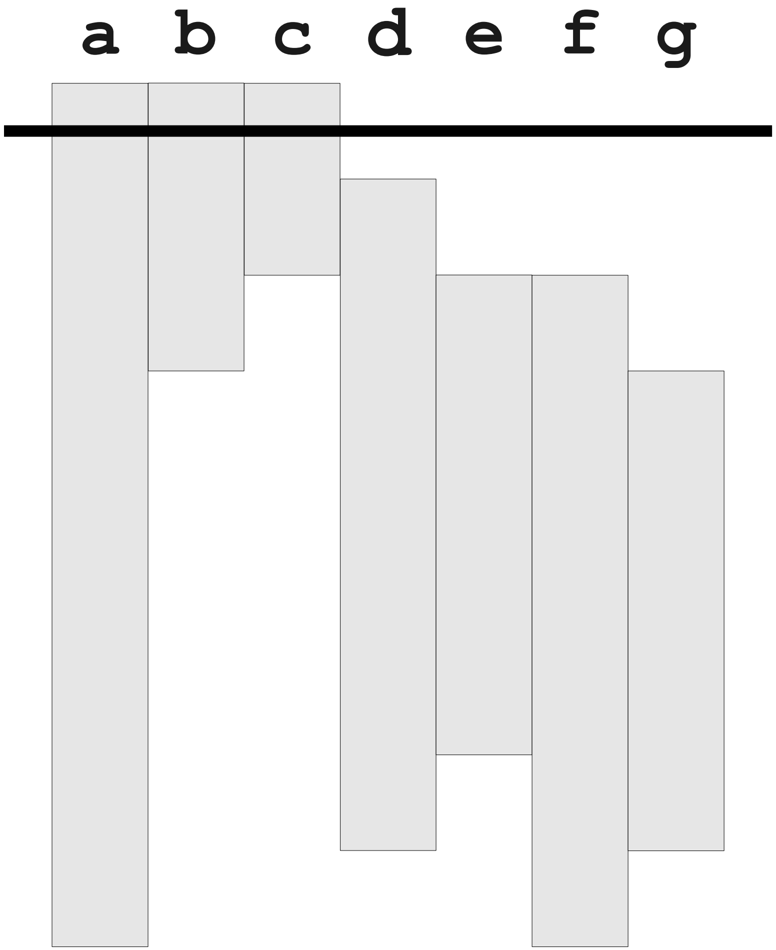
Another Example



Free Registers

R_0	R_1	R_2

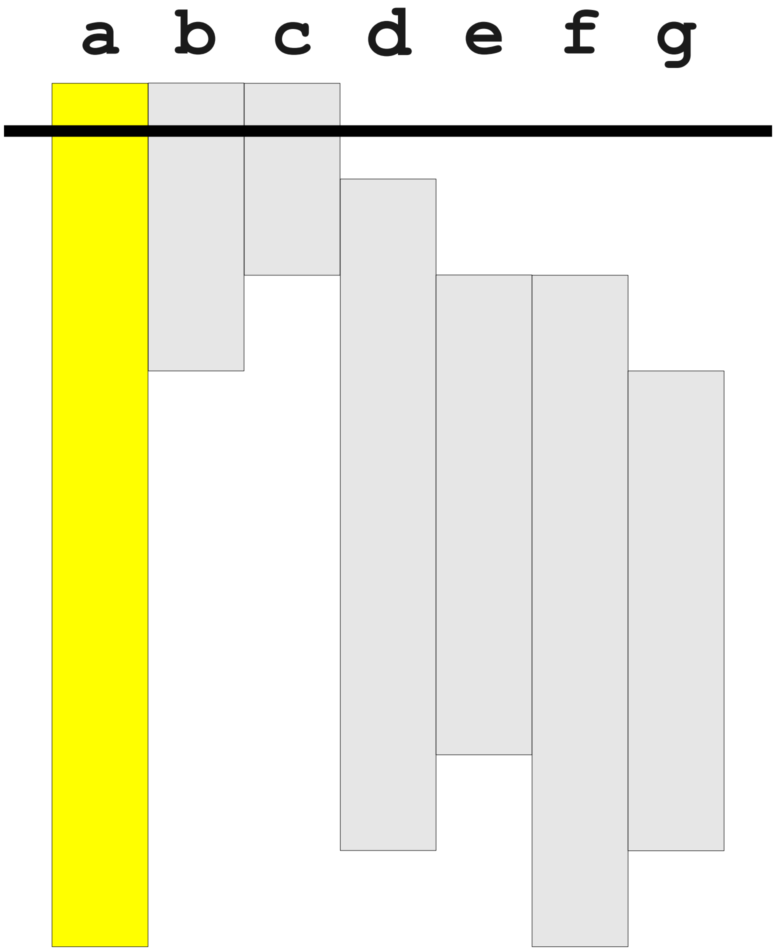
Another Example



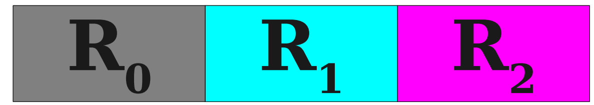
Free Registers

R_0	R_1	R_2

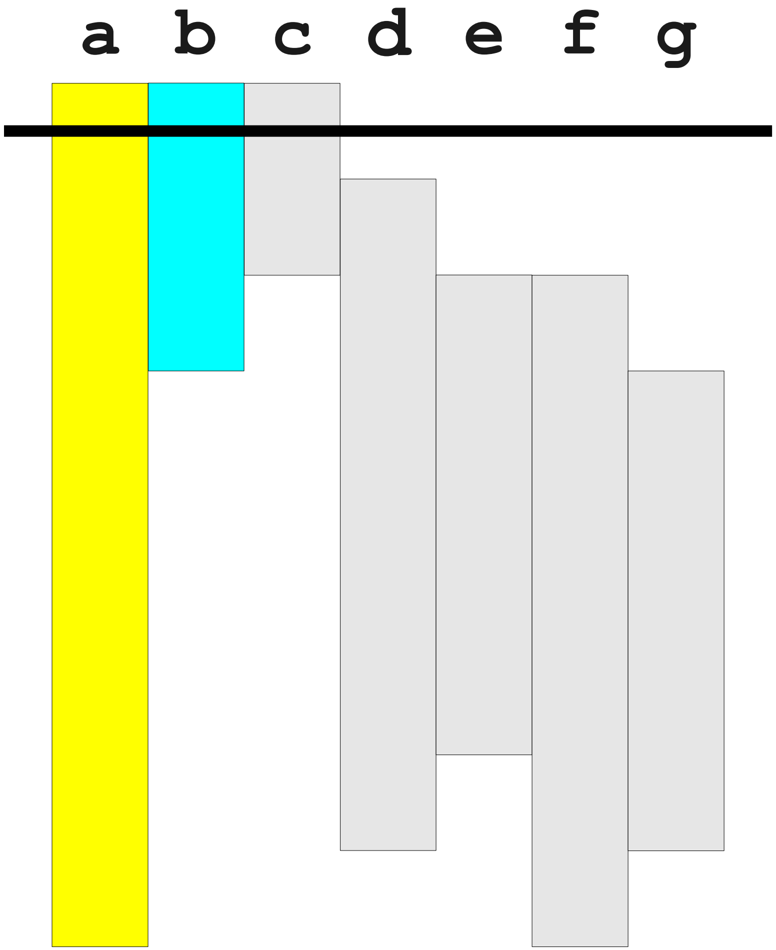
Another Example



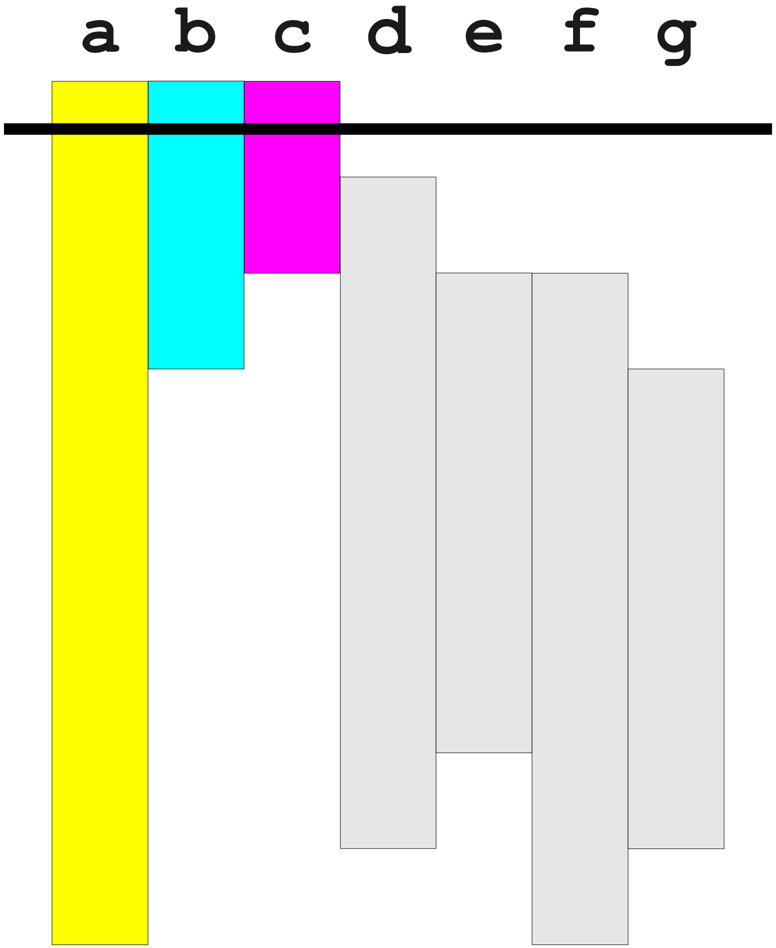
Free Registers



Another Example



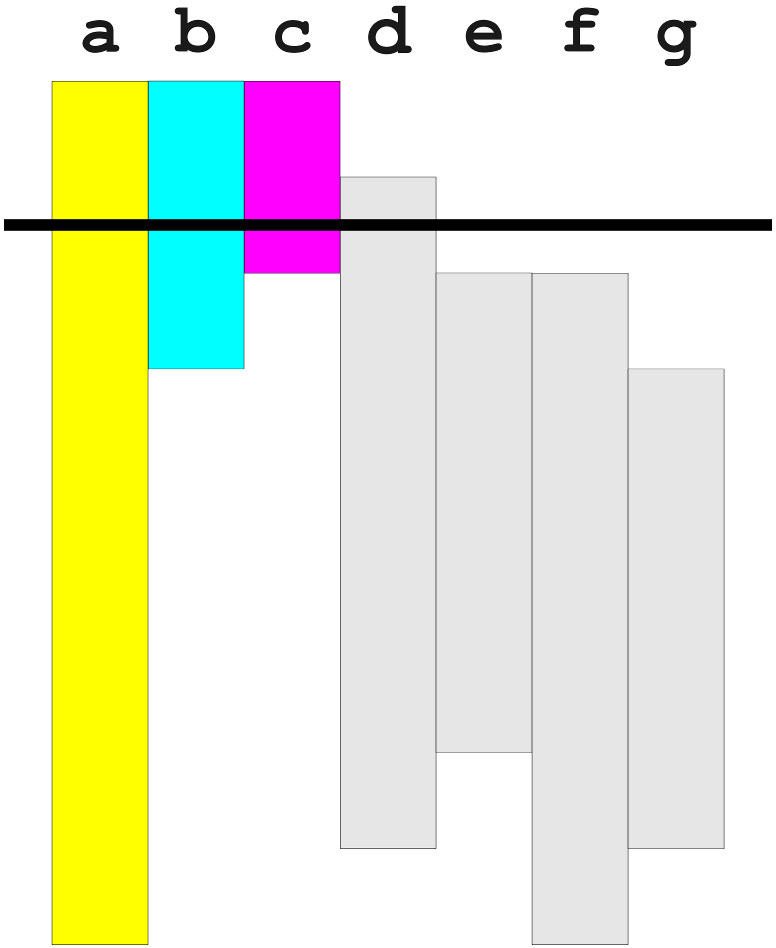
Another Example



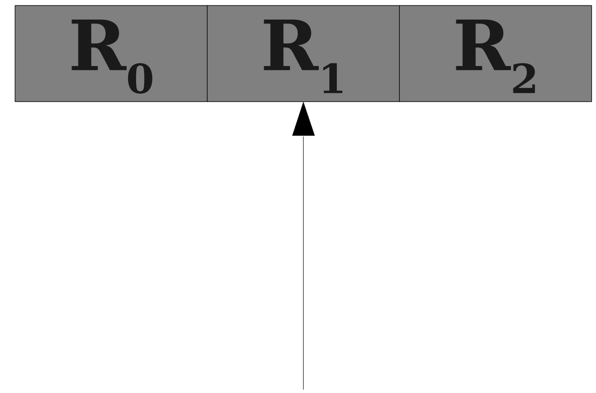
Free Registers

R_0	R_1	R_2

Another Example



Free Registers

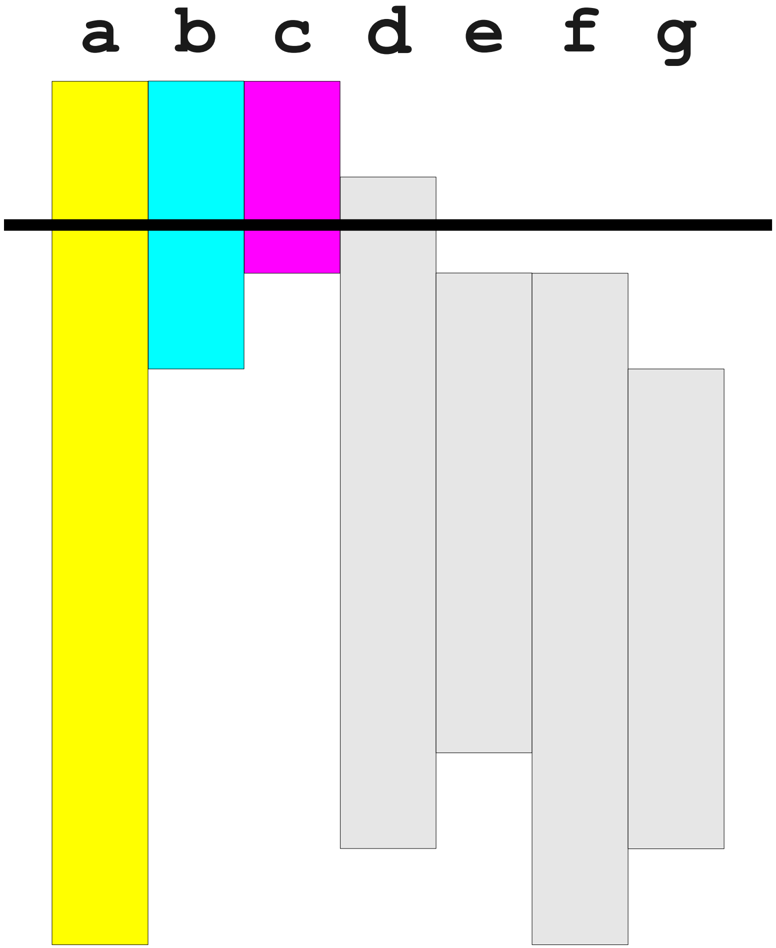


What do we do now?

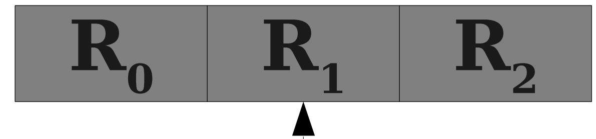
Register Spilling

- If a register cannot be found for a variable *v*, we may need to **spill** a variable.
- When a variable is spilled, it is stored in memory rather than a register.
- When we need a register for the spilled variable:
  - Evict some existing register to memory.
  - Load the variable into the register.
  - When done, write the register back to memory and reload the register with its original value.
- Spilling is slow, but sometimes necessary.

Another Example

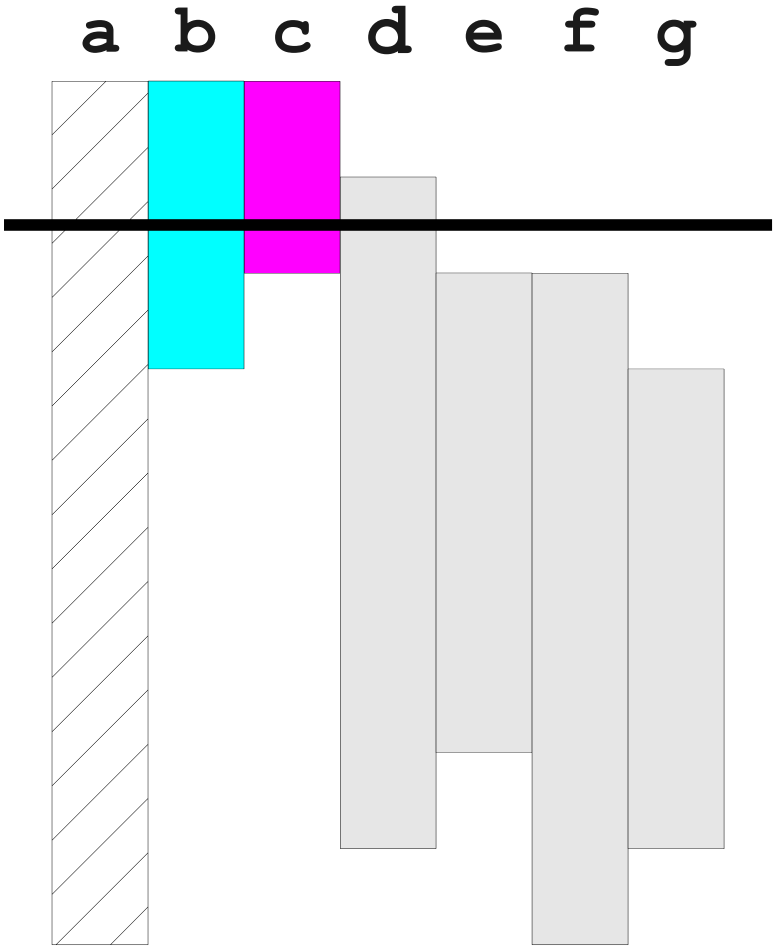


Free Registers



What do we do now?

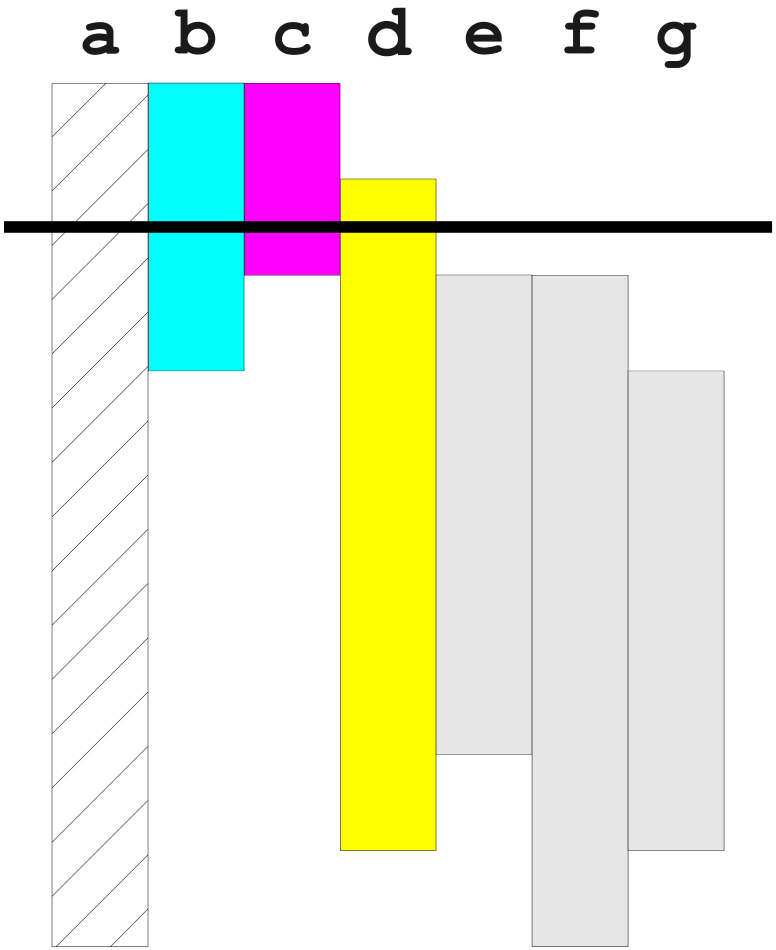
Another Example



Free Registers

R_0	R_1	R_2

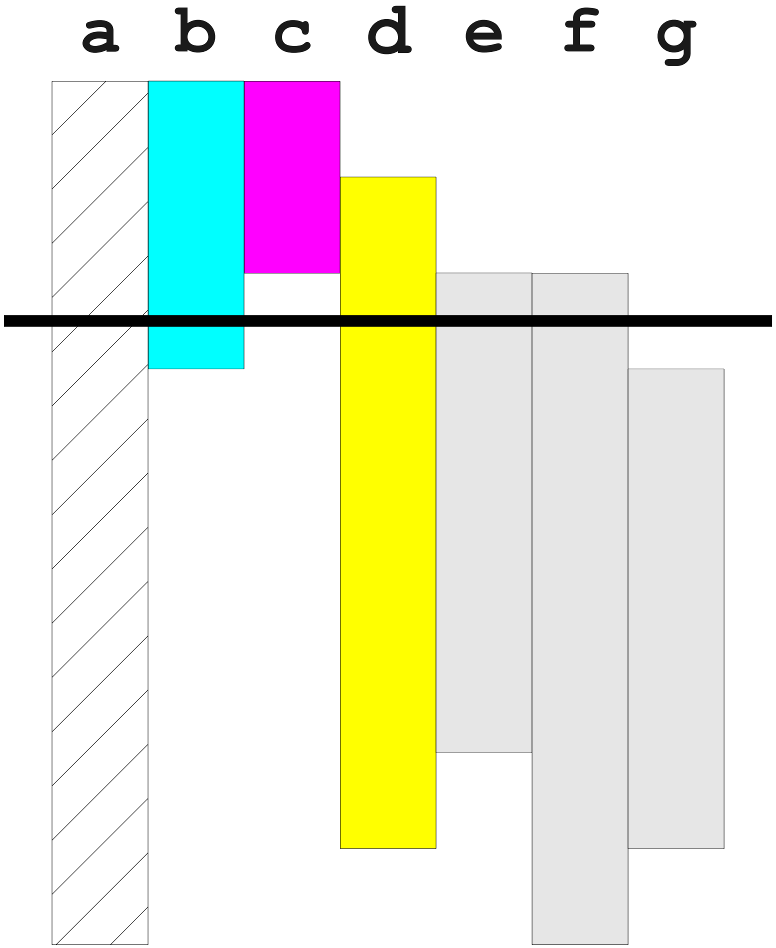
Another Example



Free Registers

R_0	R_1	R_2

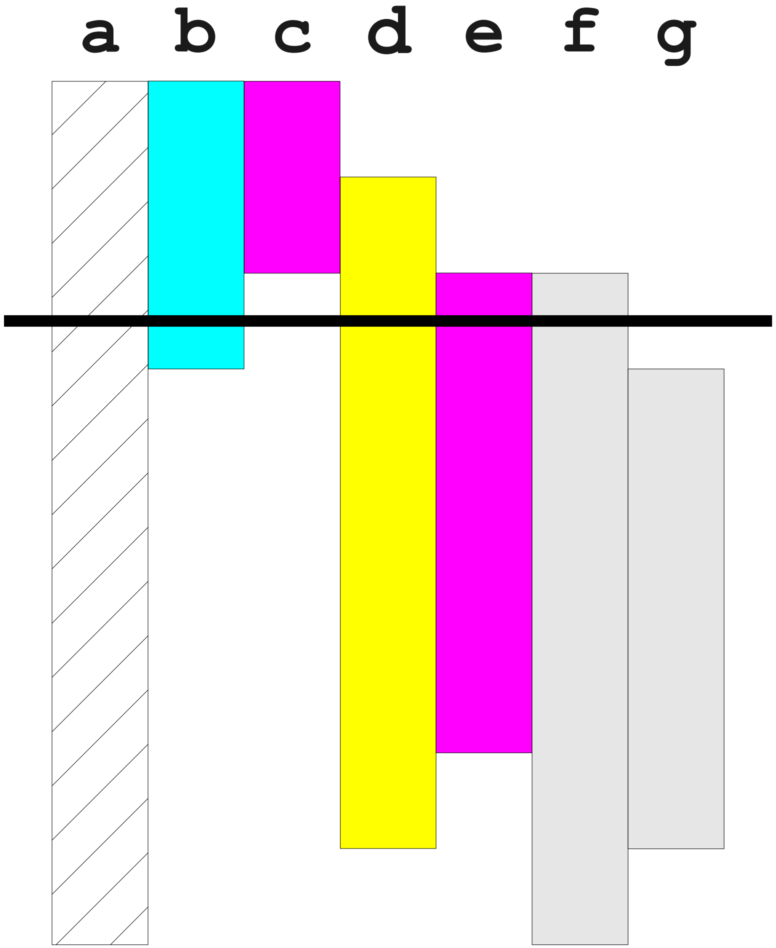
Another Example



Free Registers

R_0	R_1	R_2

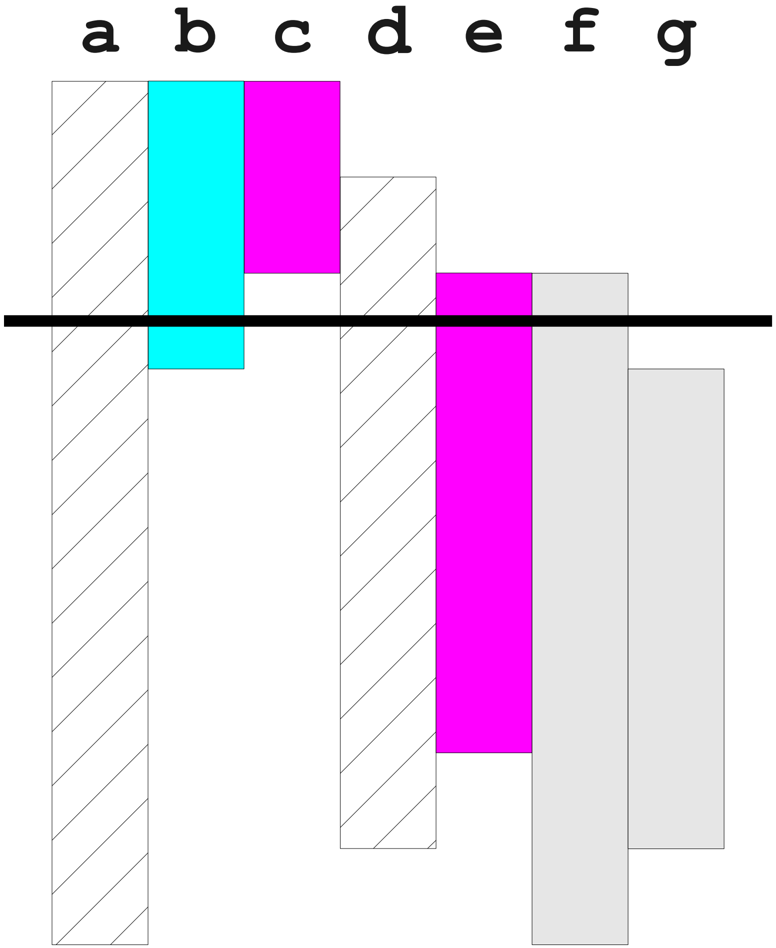
Another Example



Free Registers

R_0	R_1	R_2

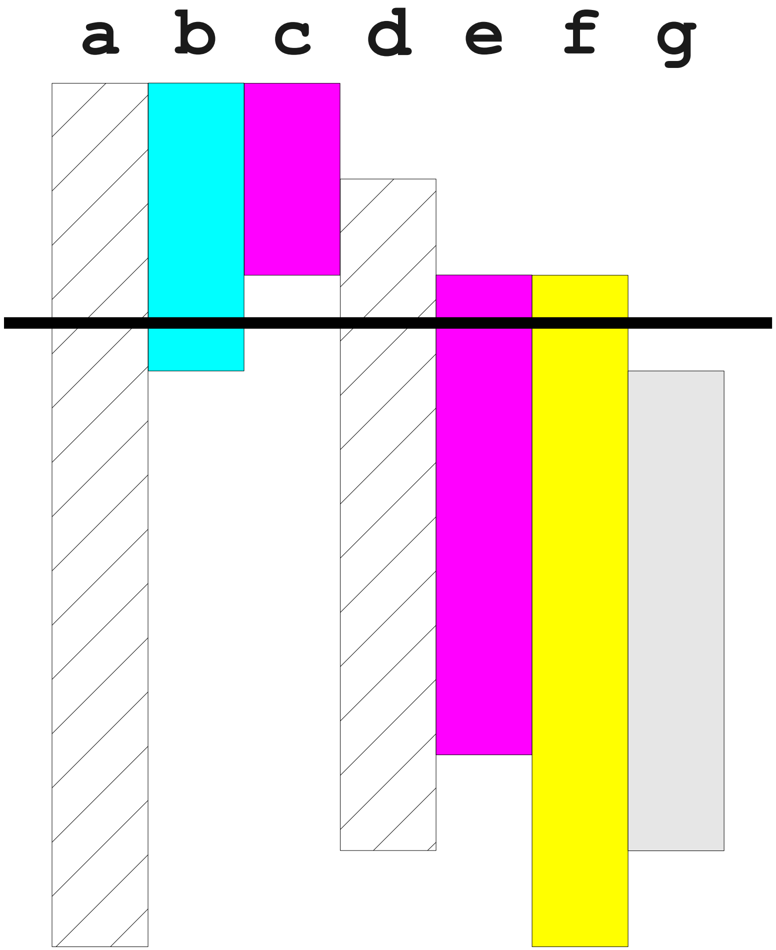
Another Example



Free Registers

R_0	R_1	R_2

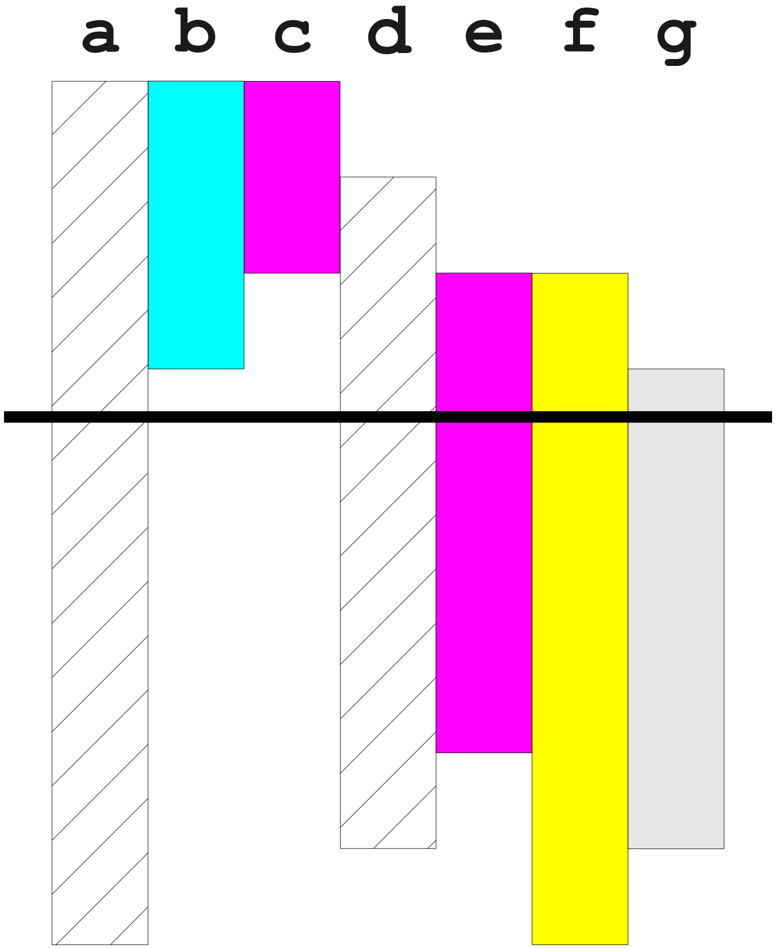
Another Example



Free Registers

R_0	R_1	R_2

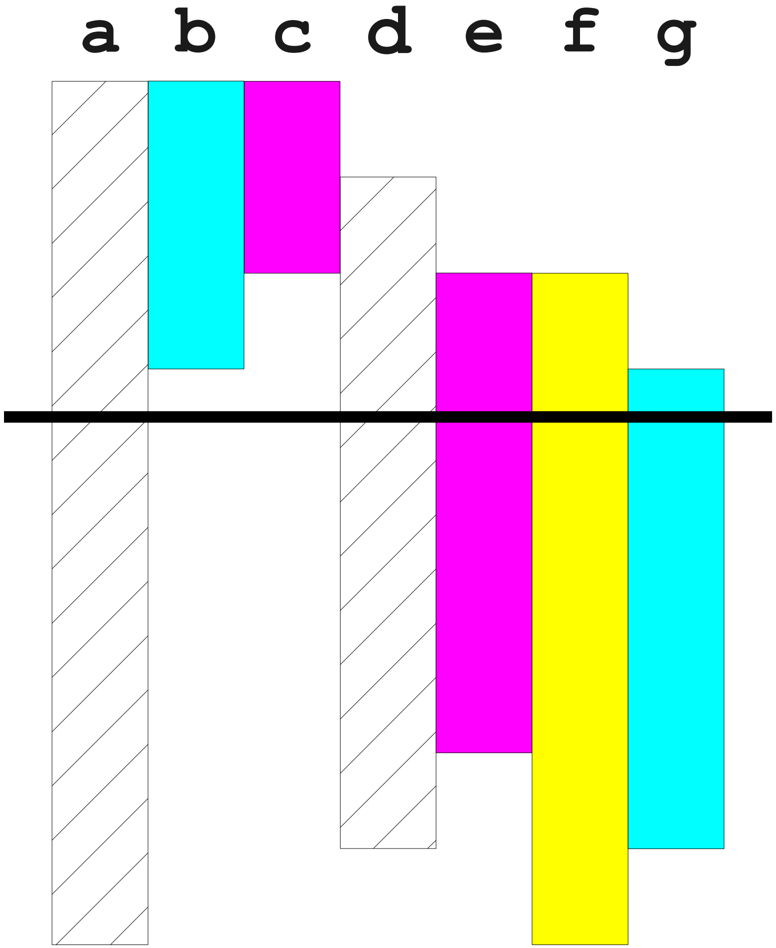
Another Example



Free Registers

R_0	R_1	R_2

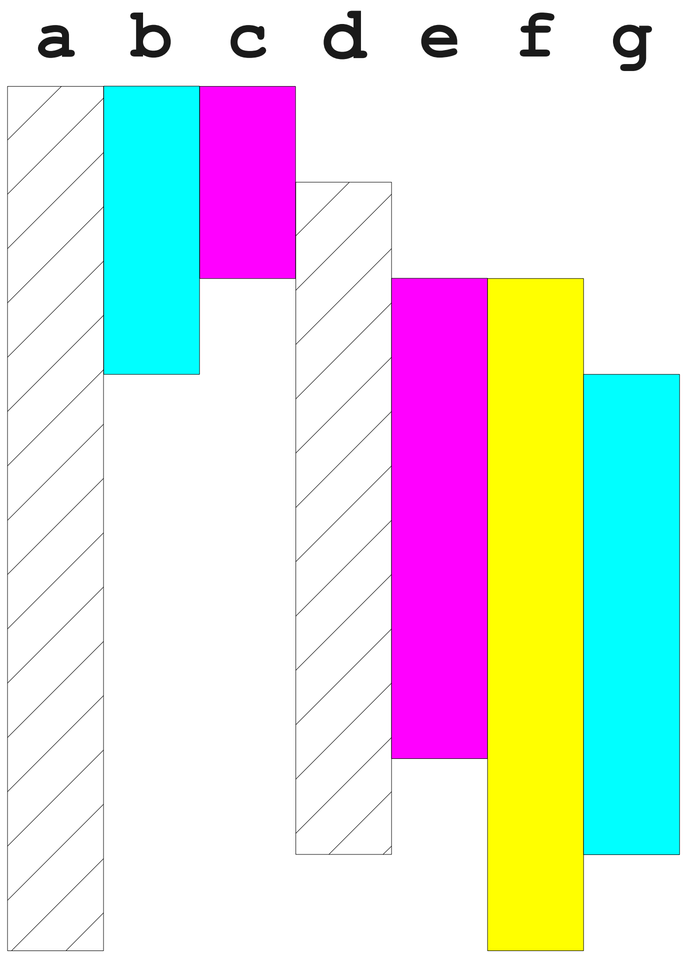
Another Example



Free Registers

R_0	R_1	R_2

Another Example



Free Registers

R_0	R_1	R_2

Linear Scan Register Allocation

- This algorithm is called **linear scan register allocation** and is a comparatively new algorithm.
- Advantages:
  - Very efficient (after computing live intervals, runs in linear time)
  - Produces good code in many instances.
  - Allocation step works in one pass; can generate code during iteration.
  - Often used in JIT compilers like Java HotSpot.
- Disadvantages:
  - Imprecise due to use of live **intervals** rather than live **ranges**.
  - Other techniques known to be superior in many cases.

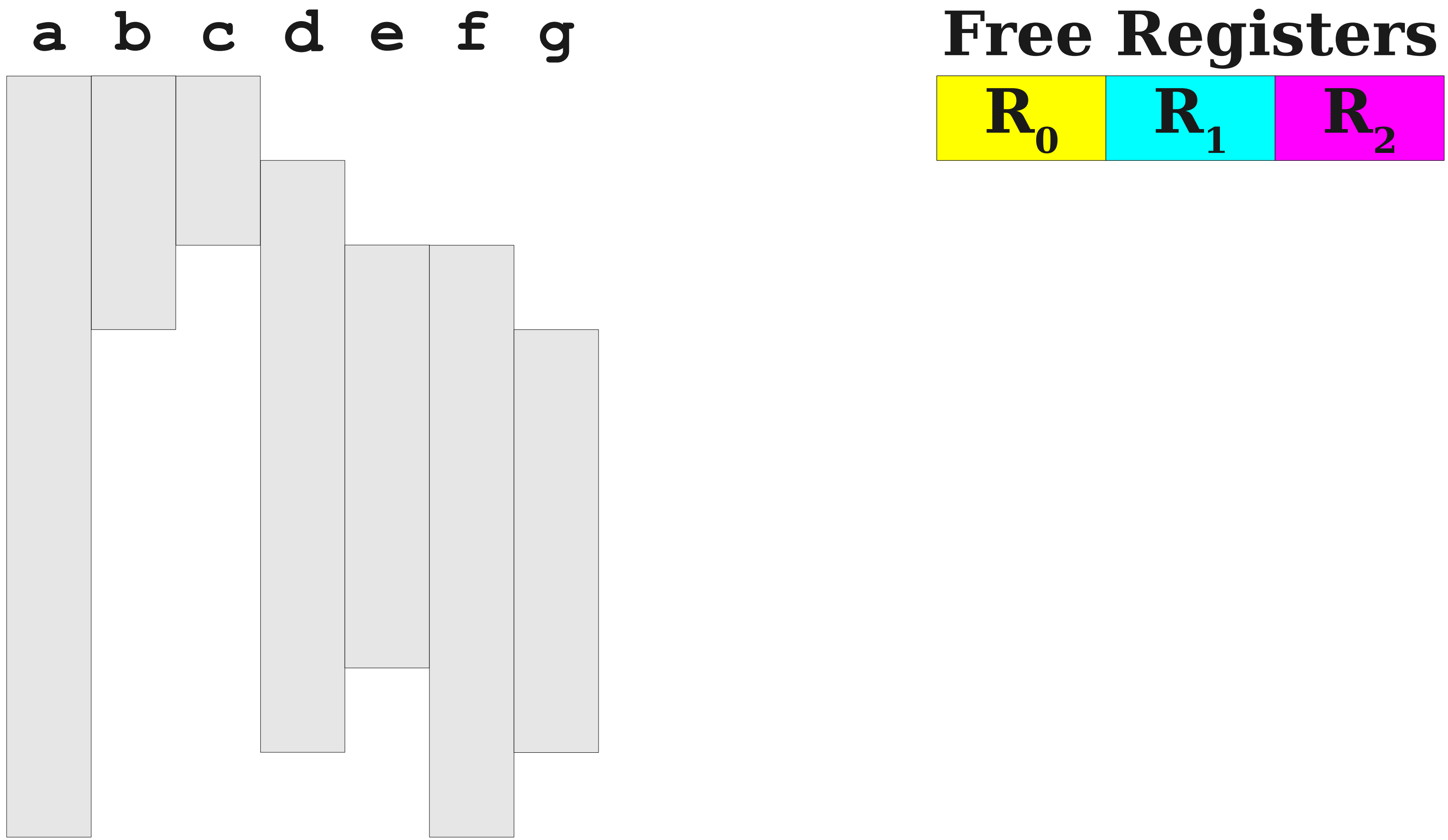
Correctness Proof Sketch

- No register holds two live variables at once:
  - Live intervals are conservative approximations of live ranges.
  - No two variables with overlapping live ranges placed in the same register.
- At each program point, every variable is in the same location:
  - All variables assigned a unique location.

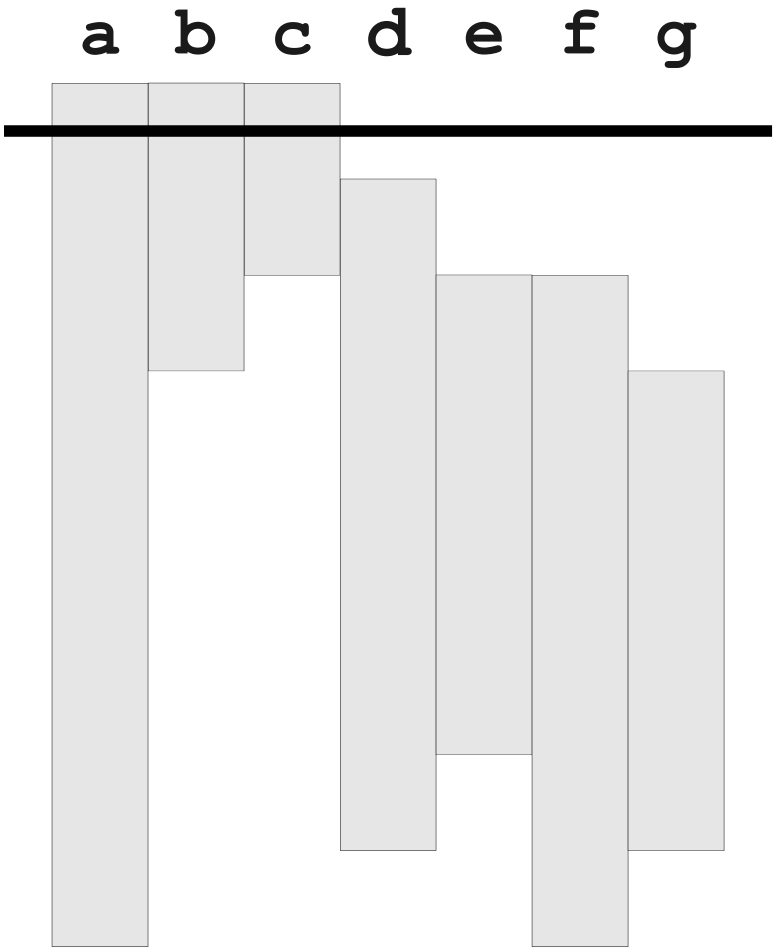
Second-Chance Bin Packing

- A more aggressive version of linear-scan.
- Uses live **ranges** instead of live **intervals**.
- If a variable must be spilled, don't spill all uses of it.
  - A later live range might still fit into a register.
- Requires a final data-flow analysis to confirm variables are assigned consistent locations.
- See “Quality and Speed in Linear-scan Register Allocation” by Traub, Holloway, and Smith.

Second-Chance Bin Packing



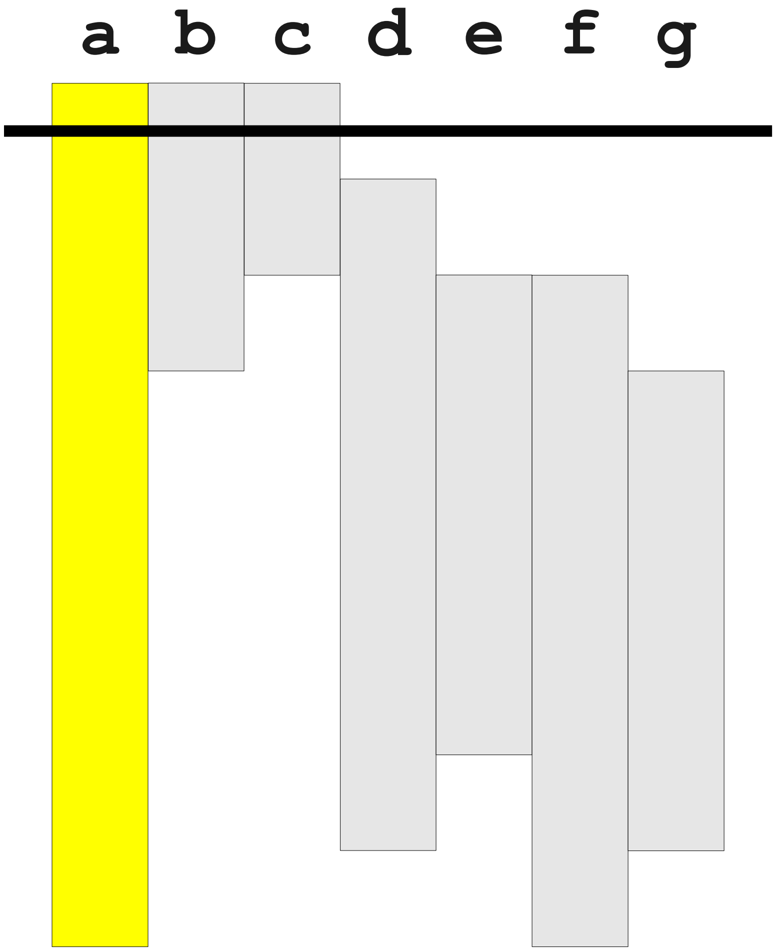
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

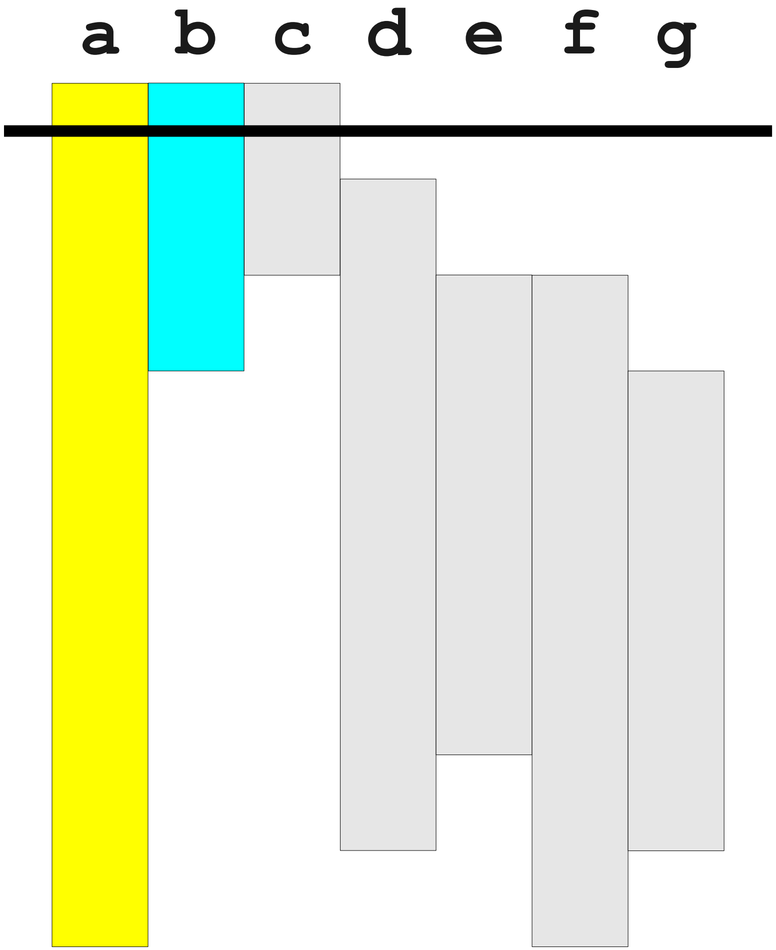
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

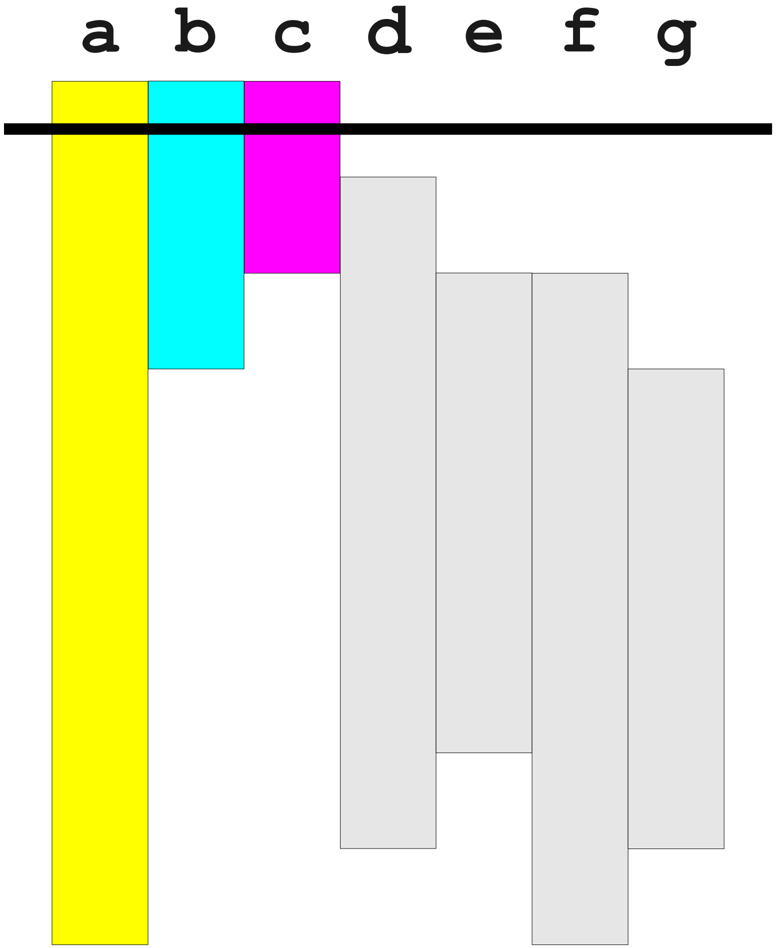
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

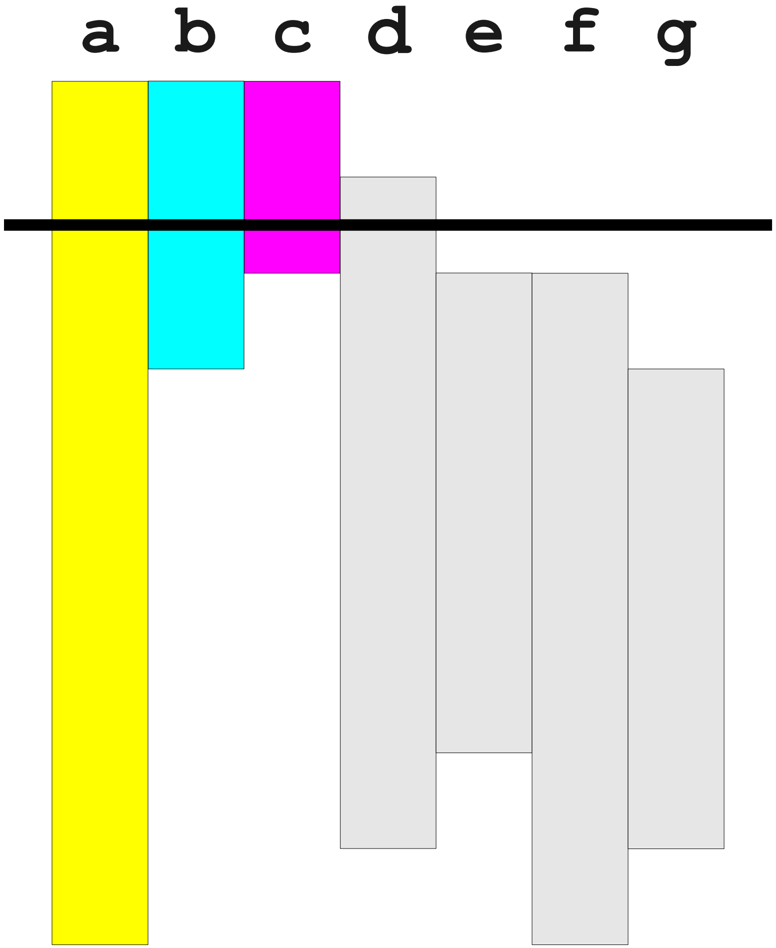
Second-Chance Bin Packing



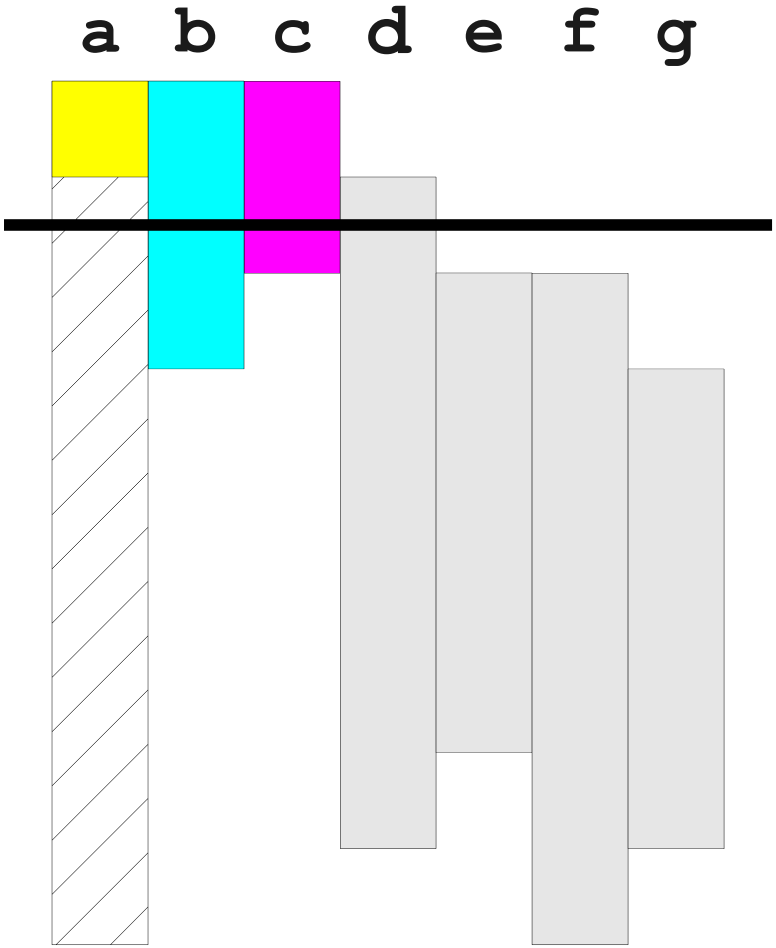
Free Registers

R_0	R_1	R_2

Second-Chance Bin Packing



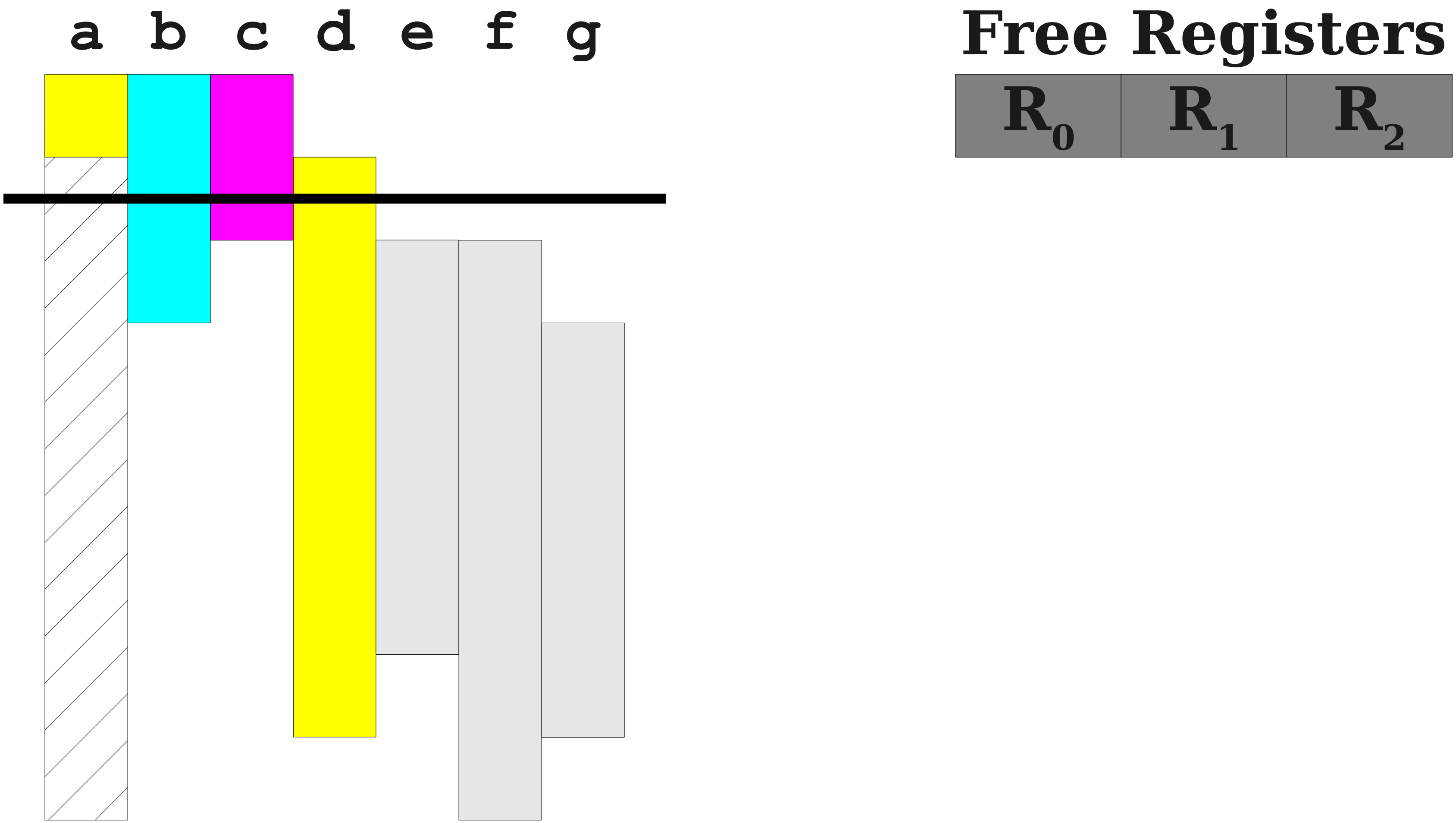
Second-Chance Bin Packing



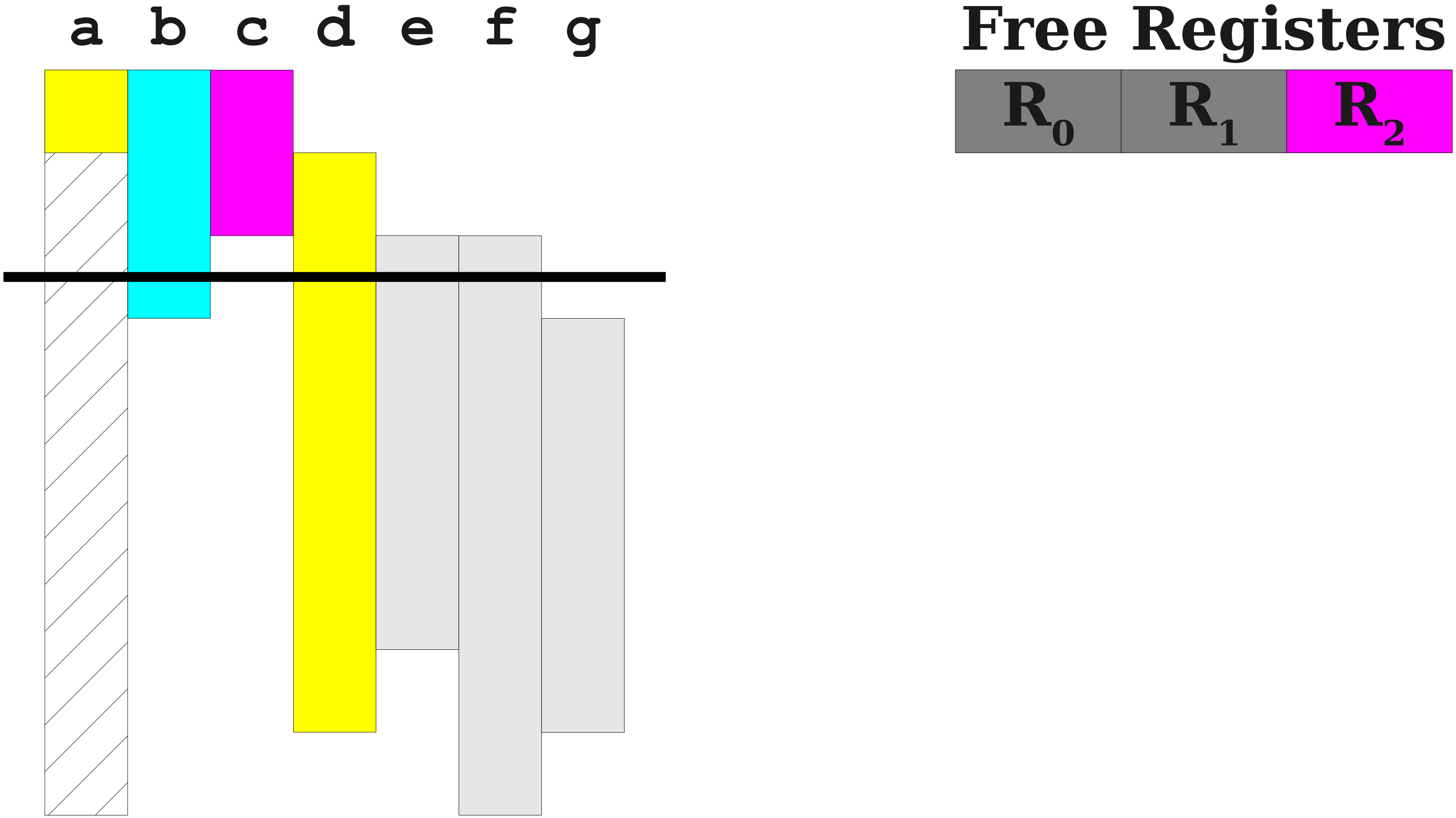
Free Registers

R_0	R_1	R_2

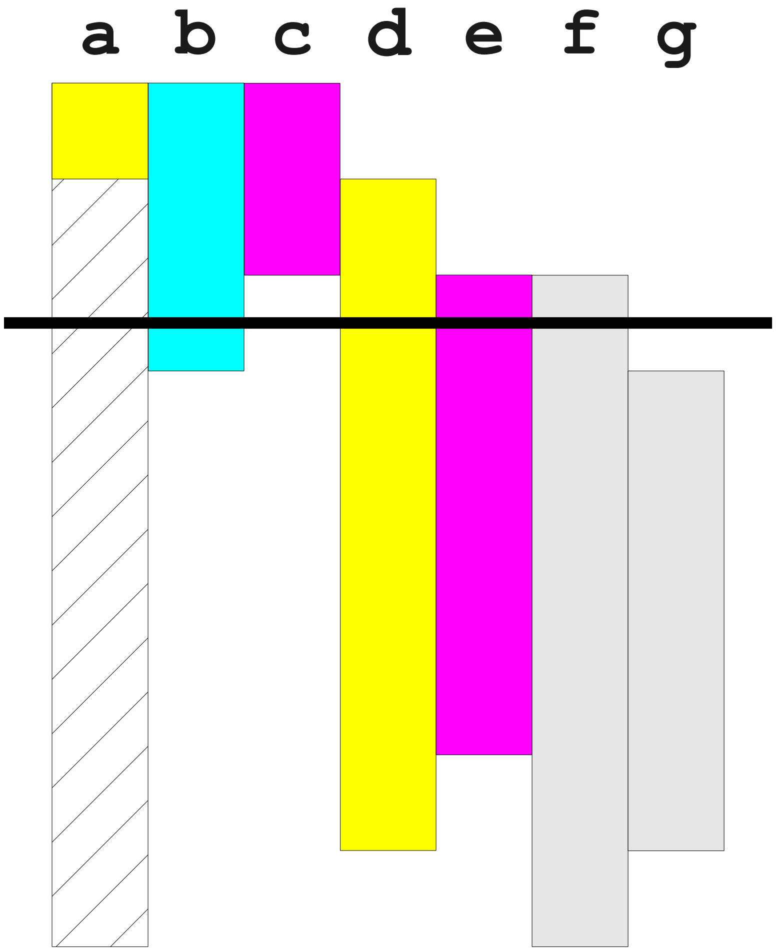
Second-Chance Bin Packing



Second-Chance Bin Packing



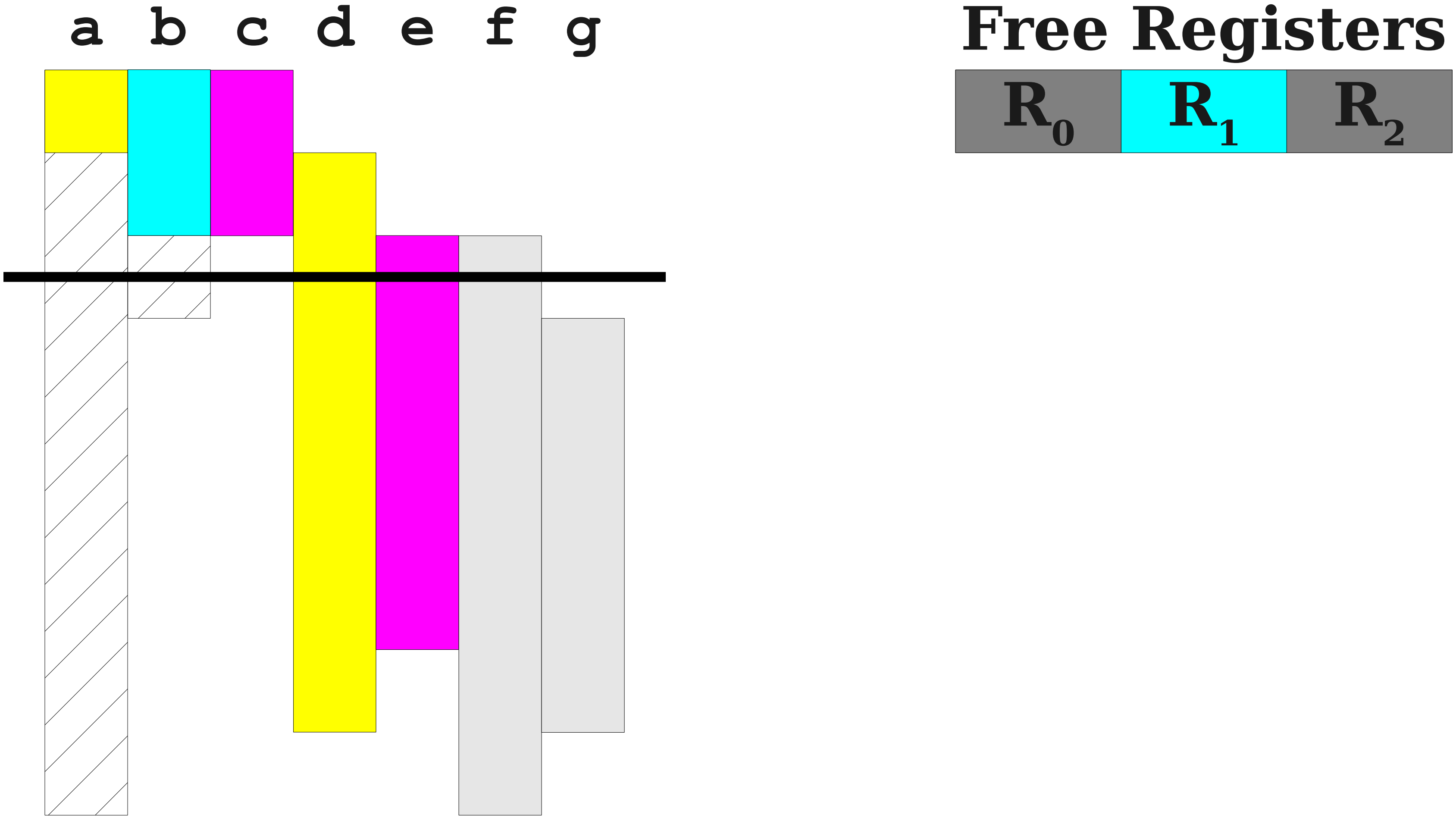
Second-Chance Bin Packing



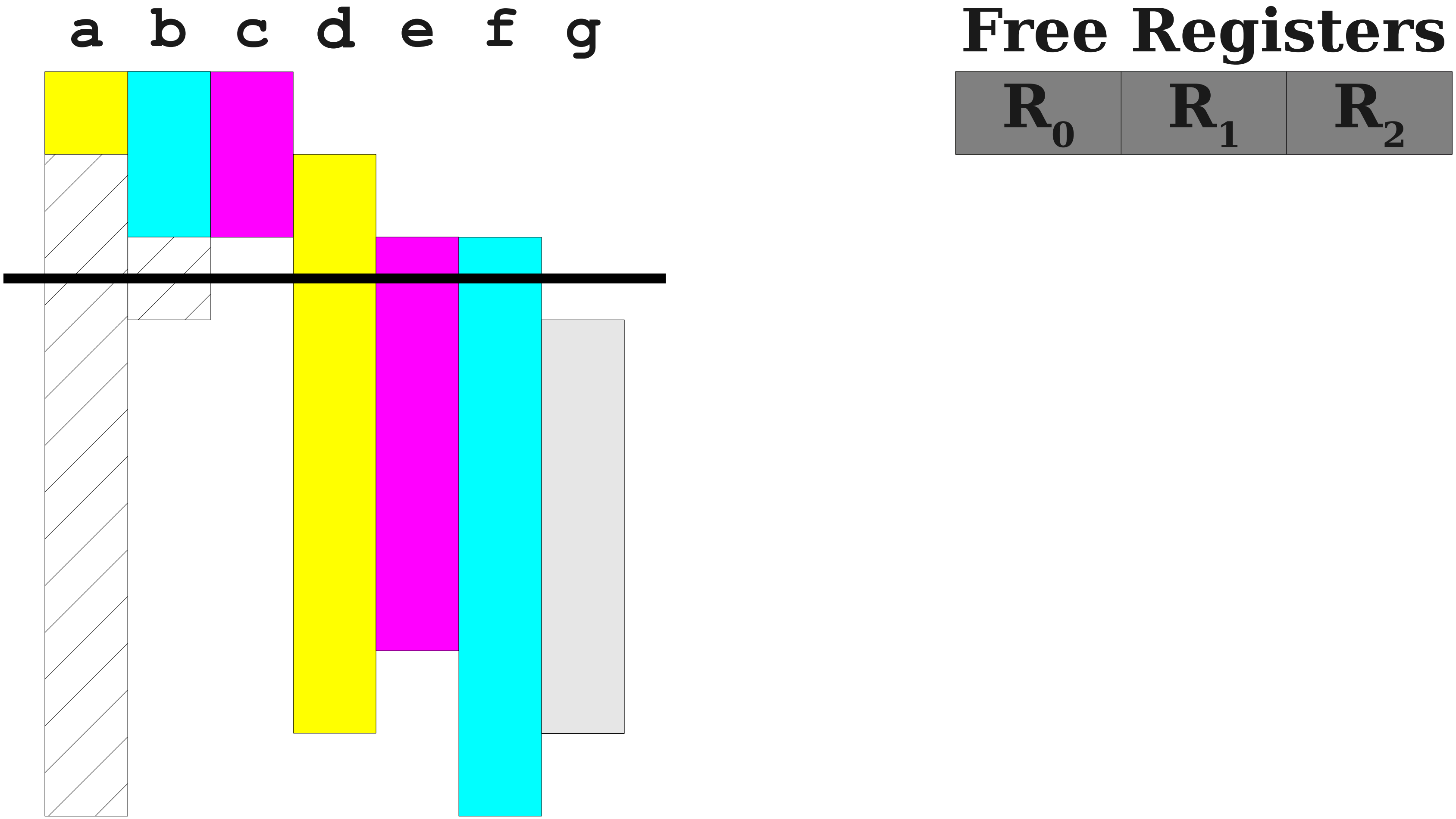
Free Registers

R_0	R_1	R_2

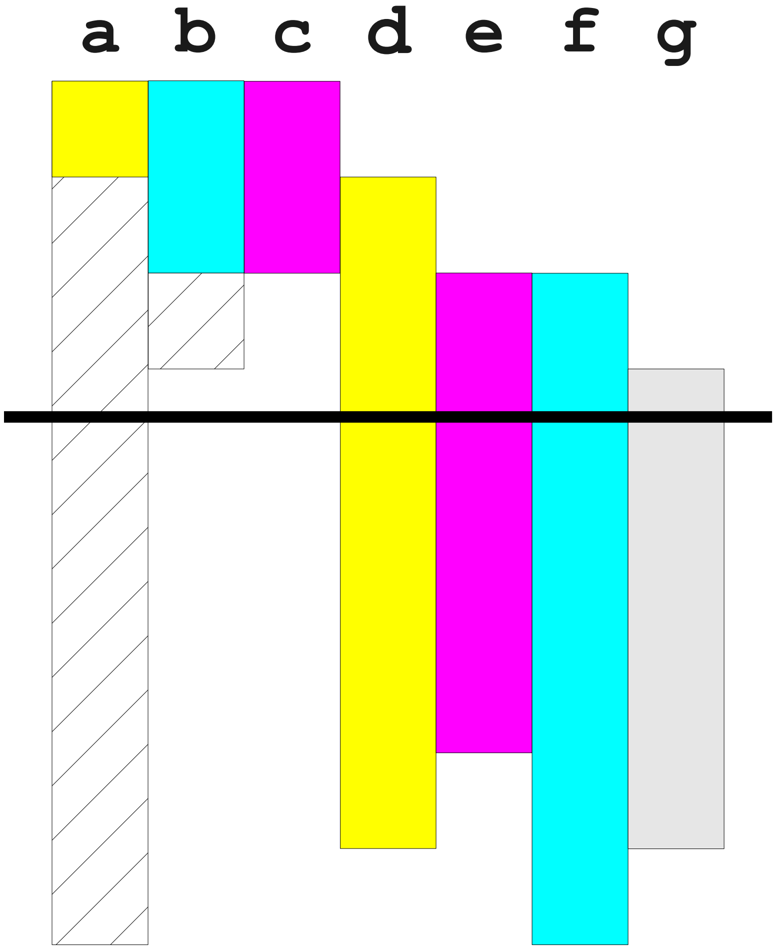
Second-Chance Bin Packing



Second-Chance Bin Packing



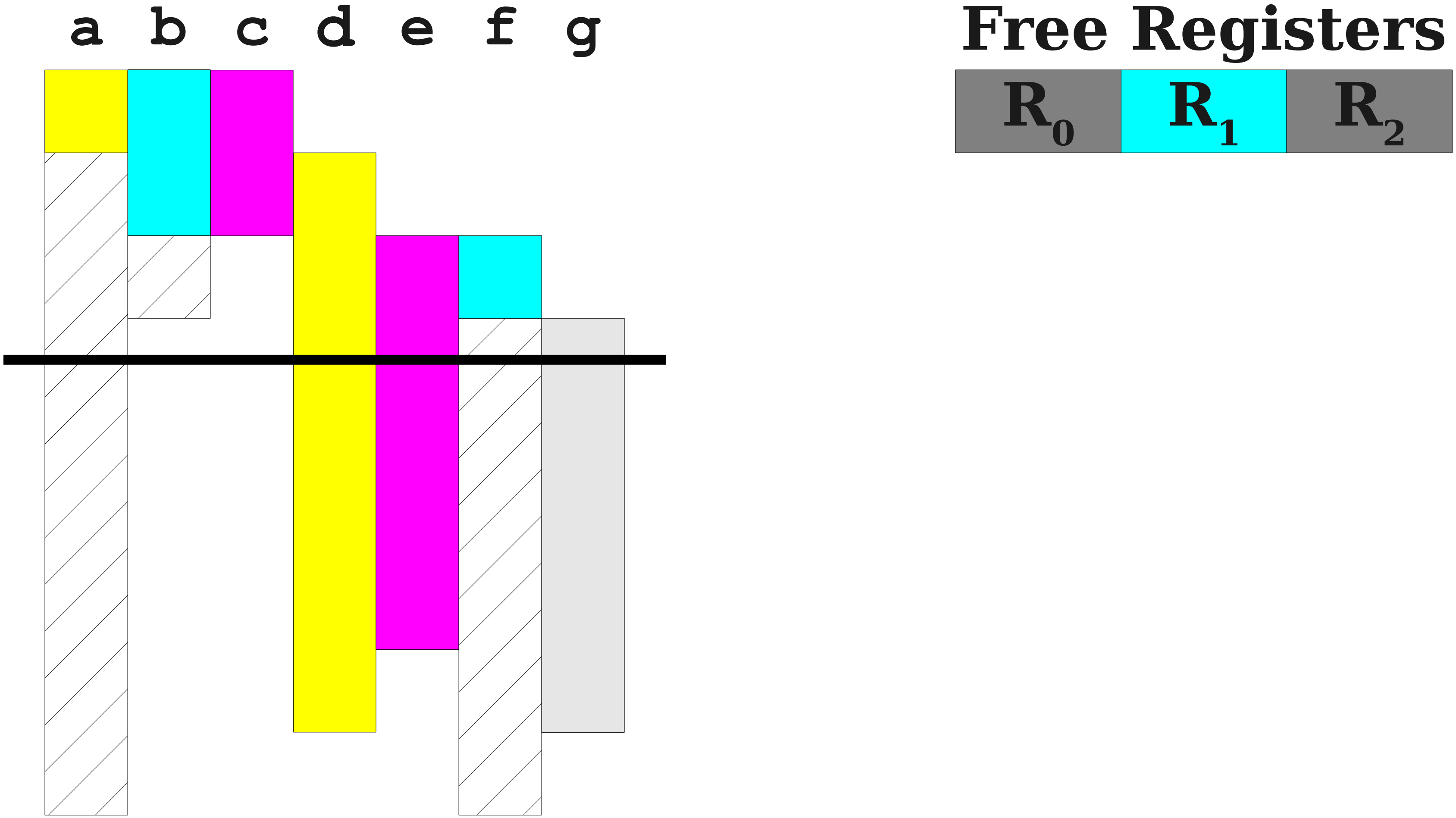
Second-Chance Bin Packing



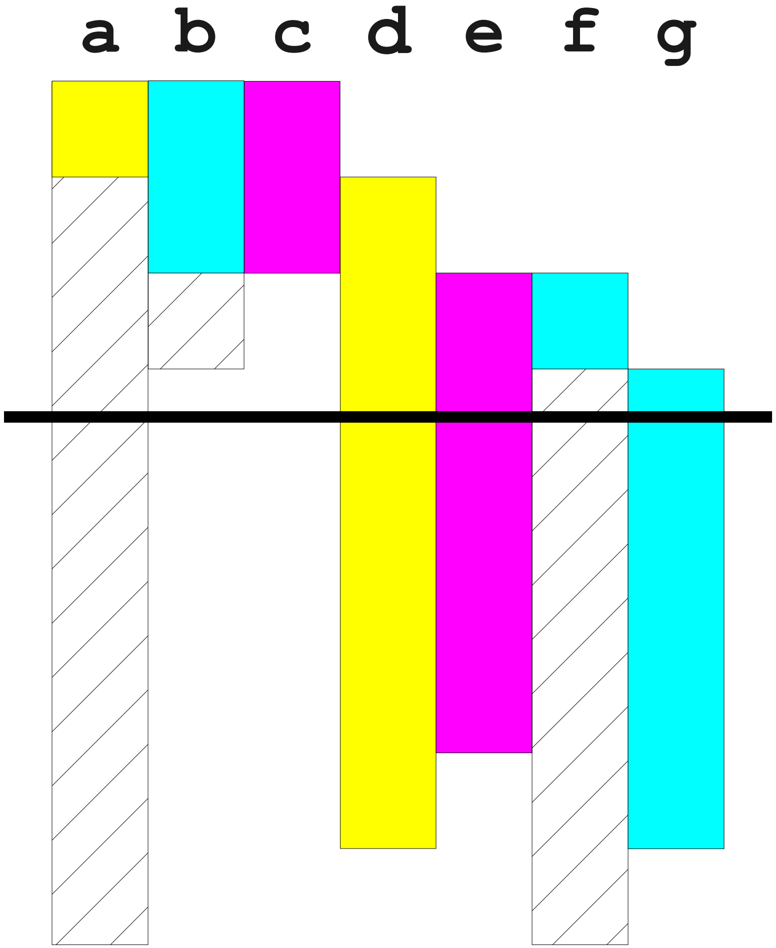
Free Registers

R_0	R_1	R_2

Second-Chance Bin Packing



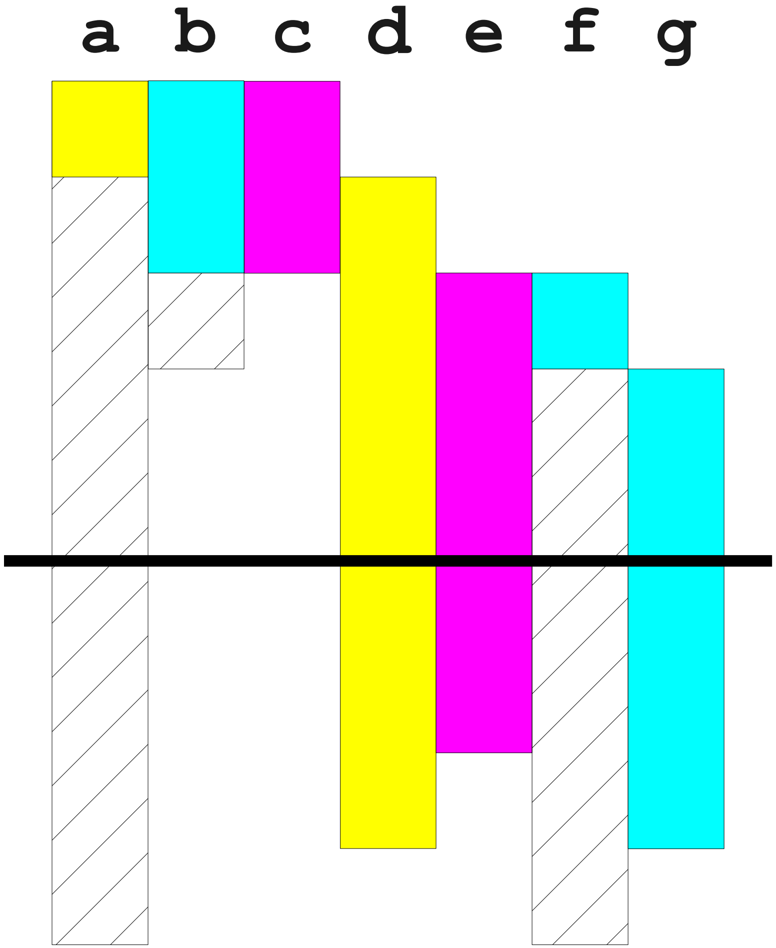
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

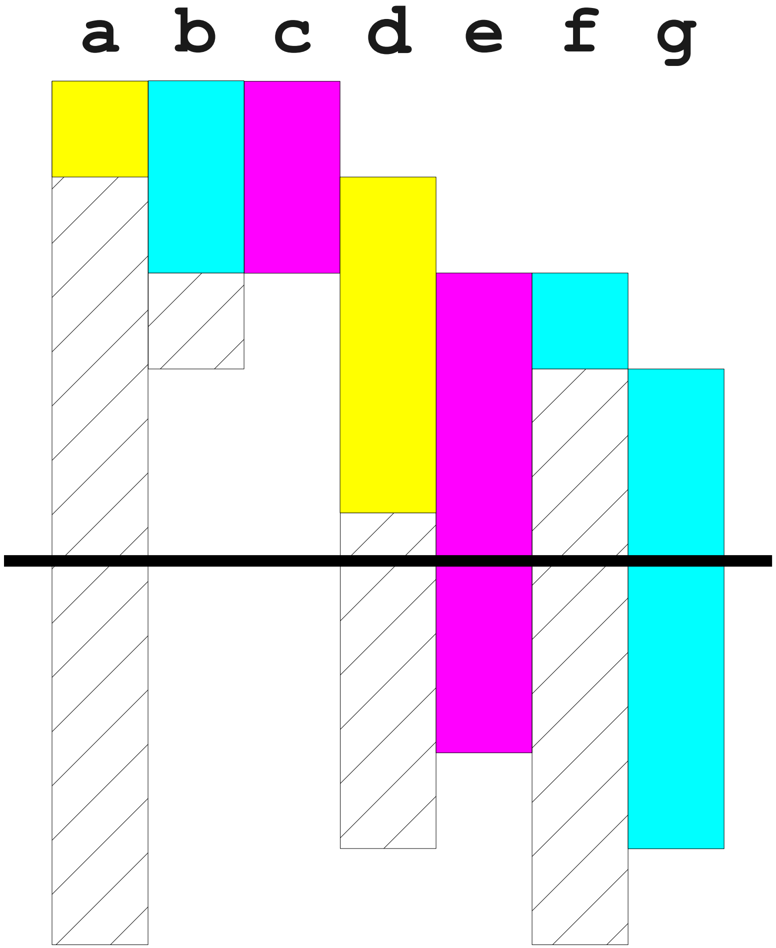
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

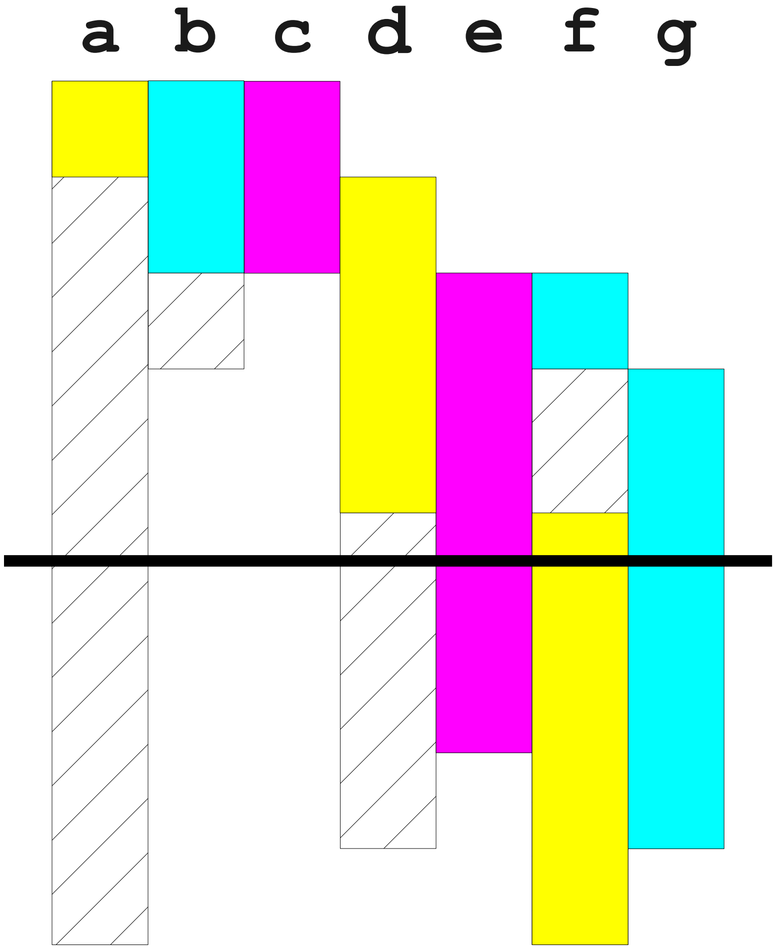
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

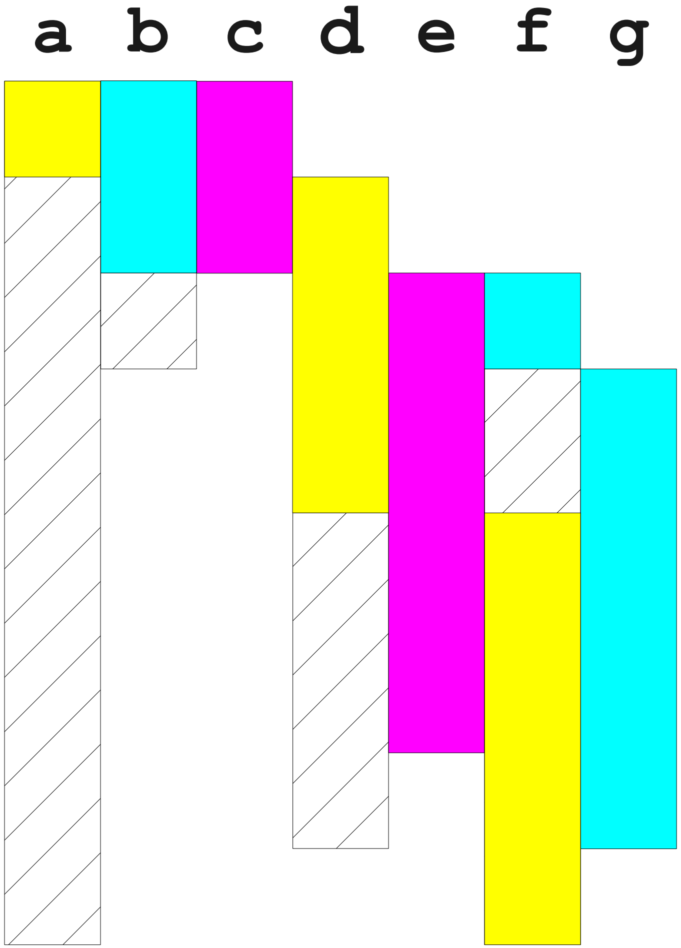
Second-Chance Bin Packing



Free Registers

R_0	R_1	R_2

Second-Chance Bin Packing

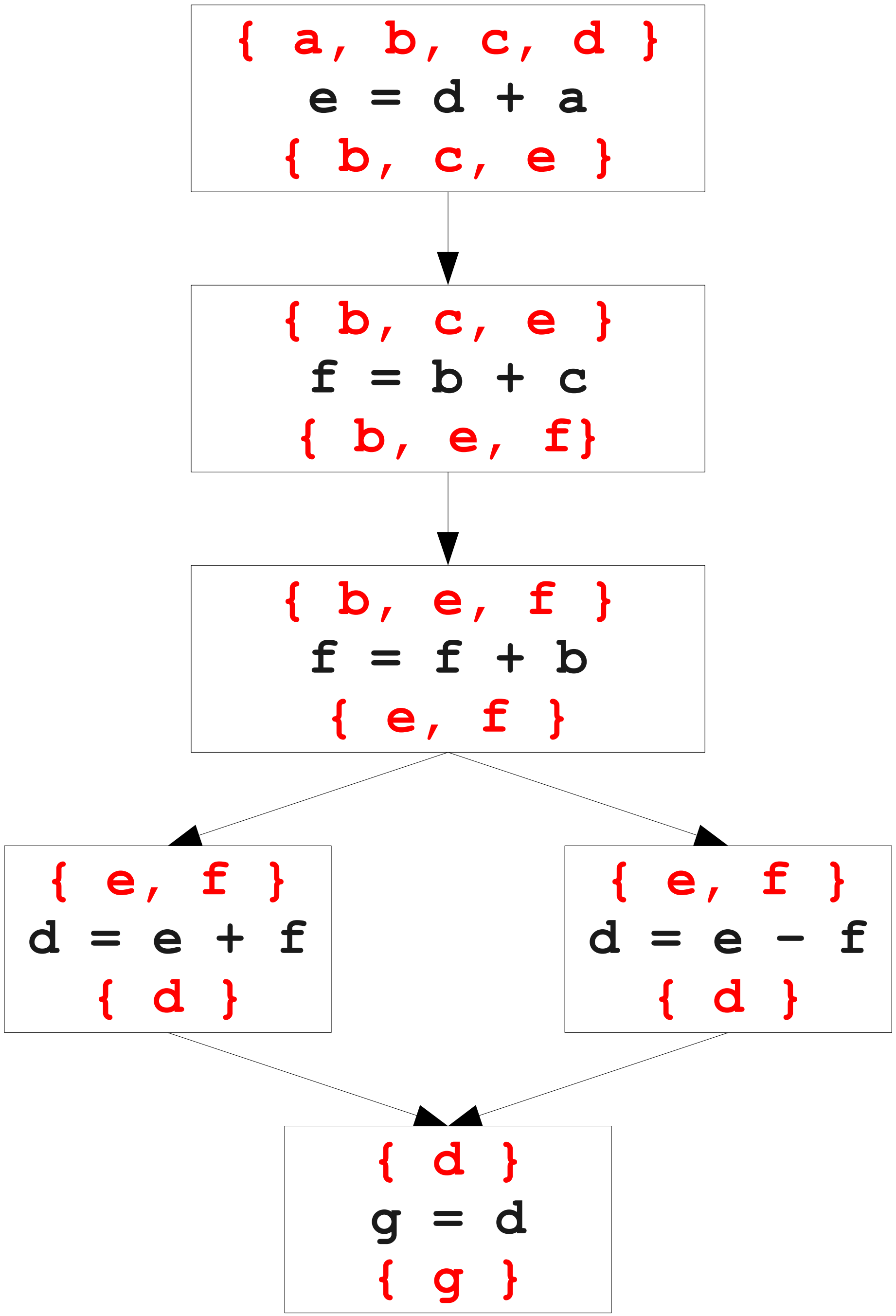


Free Registers

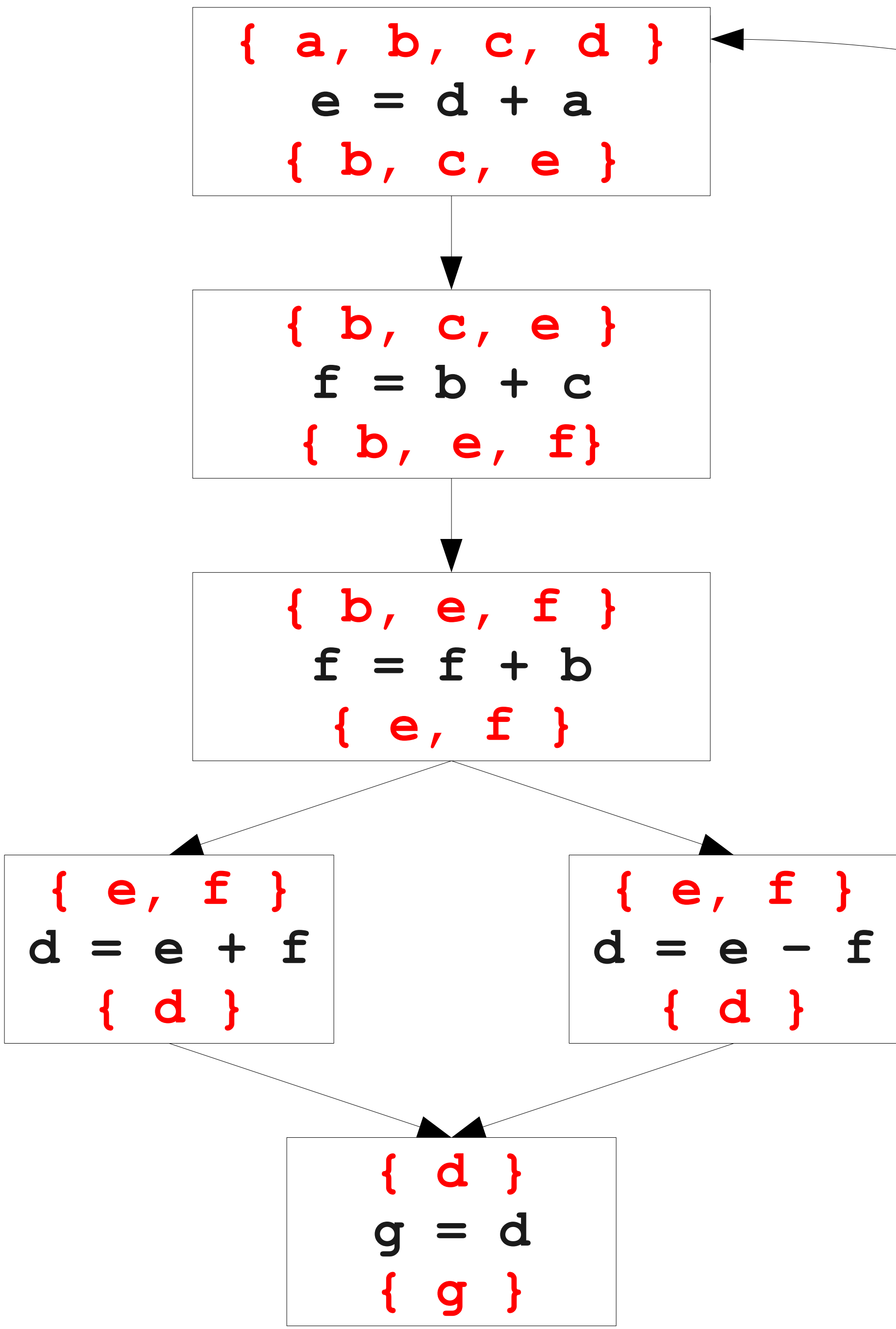
R_0	R_1	R_2

An Entirely Different Approach

An Entirely Different Approach

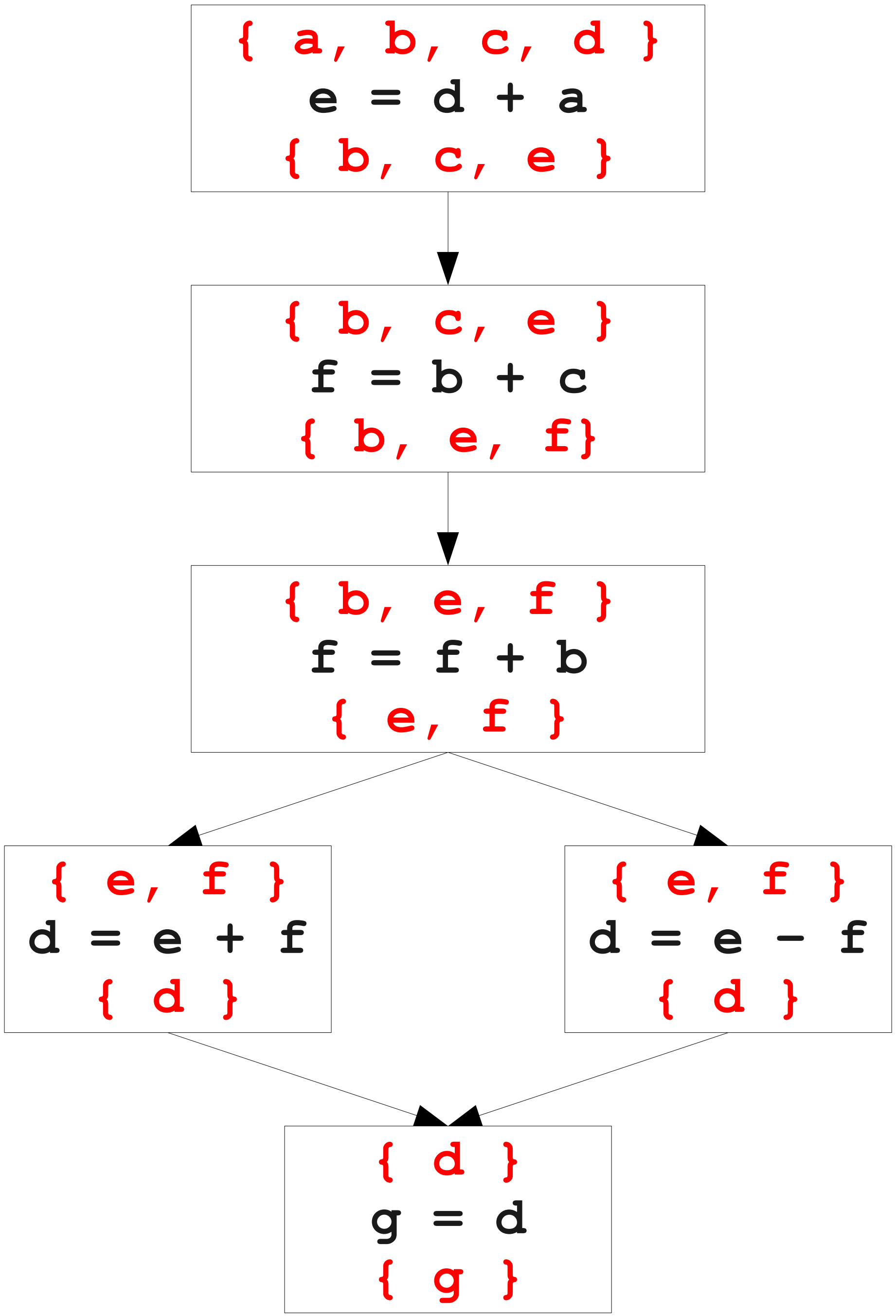


An Entirely Different Approach

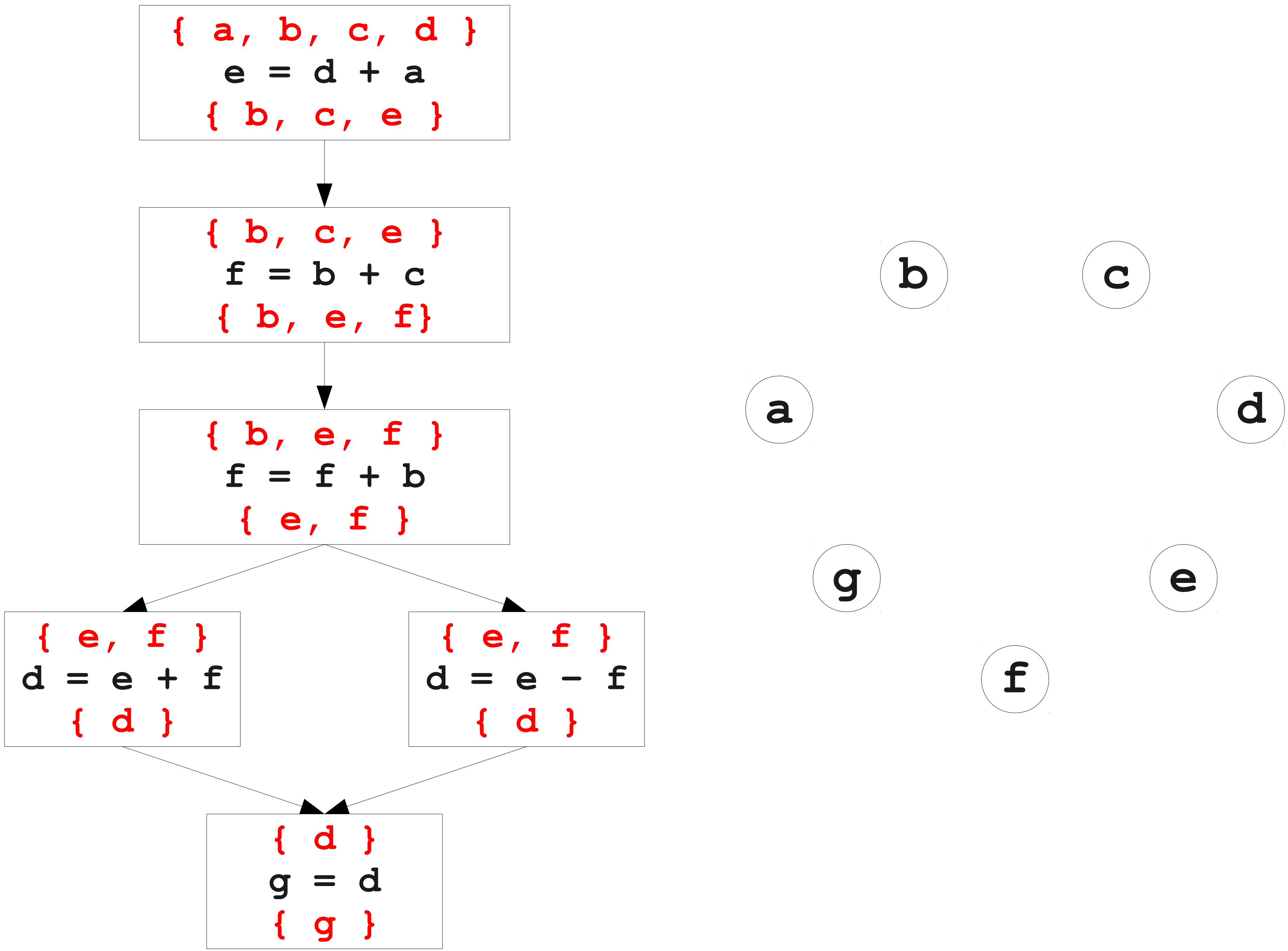


What can we infer from all these variables being live at this point?

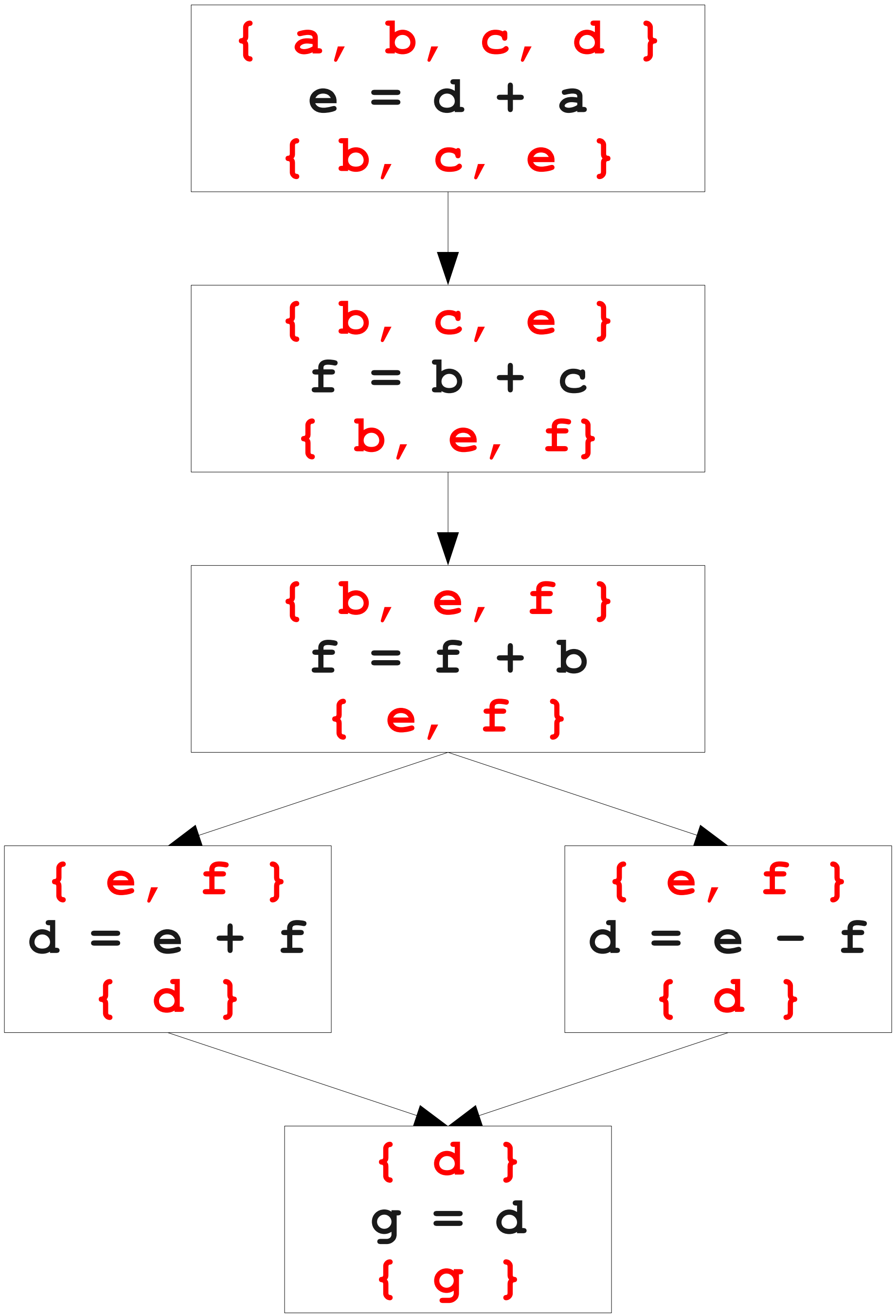
An Entirely Different Approach

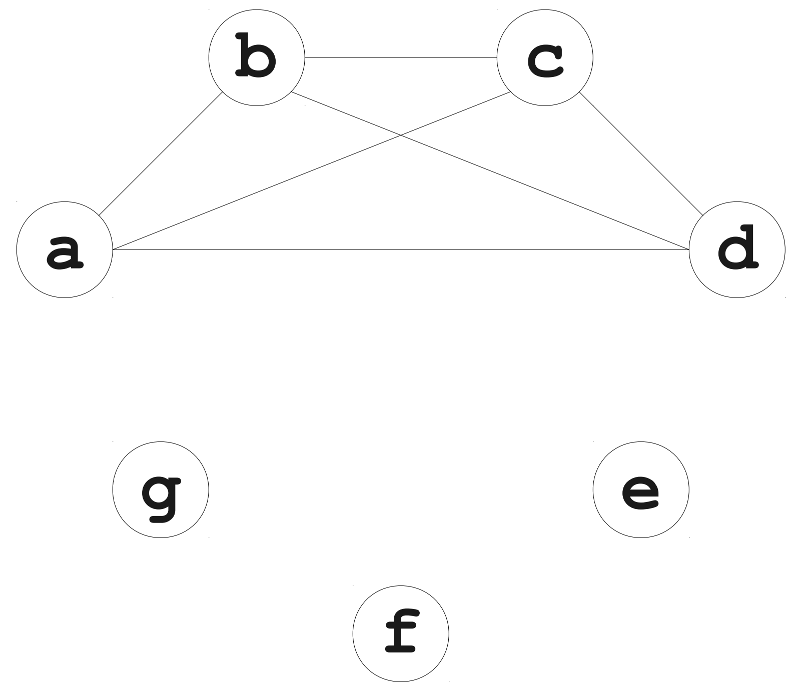


An Entirely Different Approach

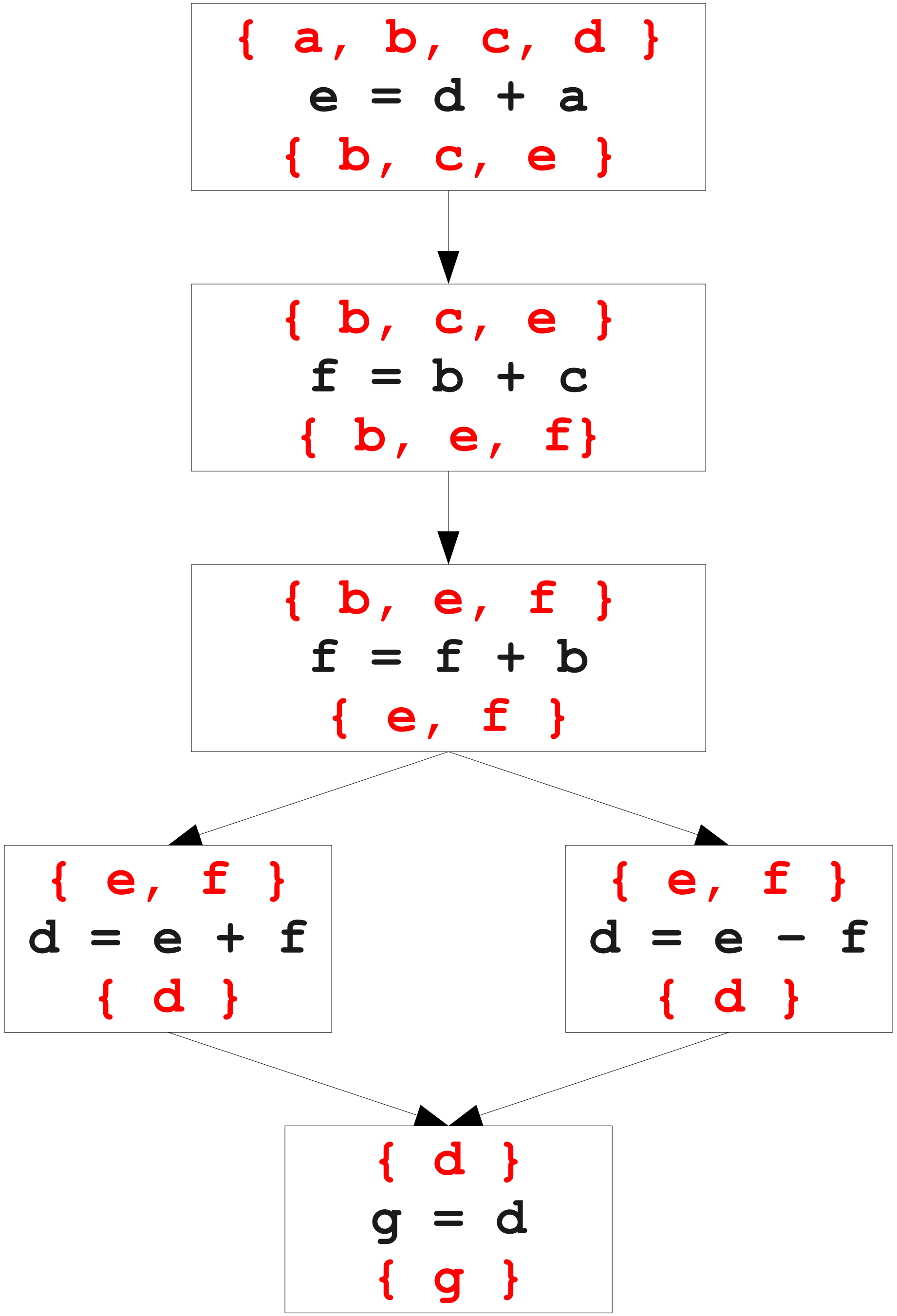


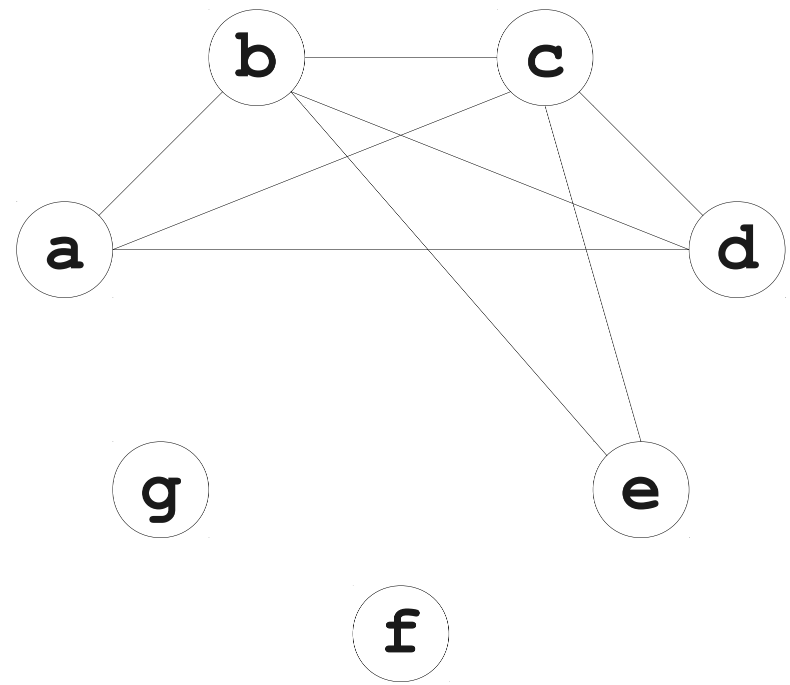
An Entirely Different Approach



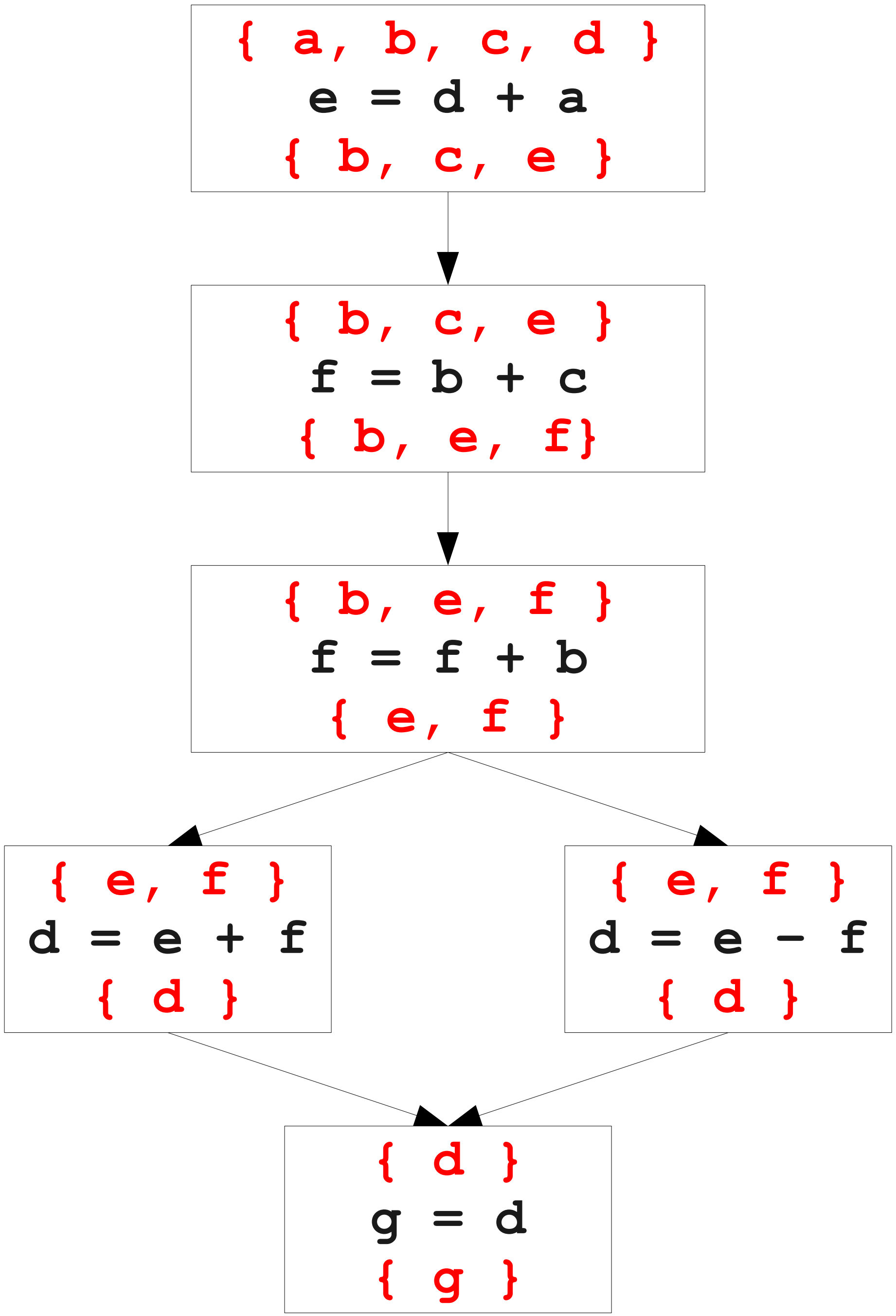


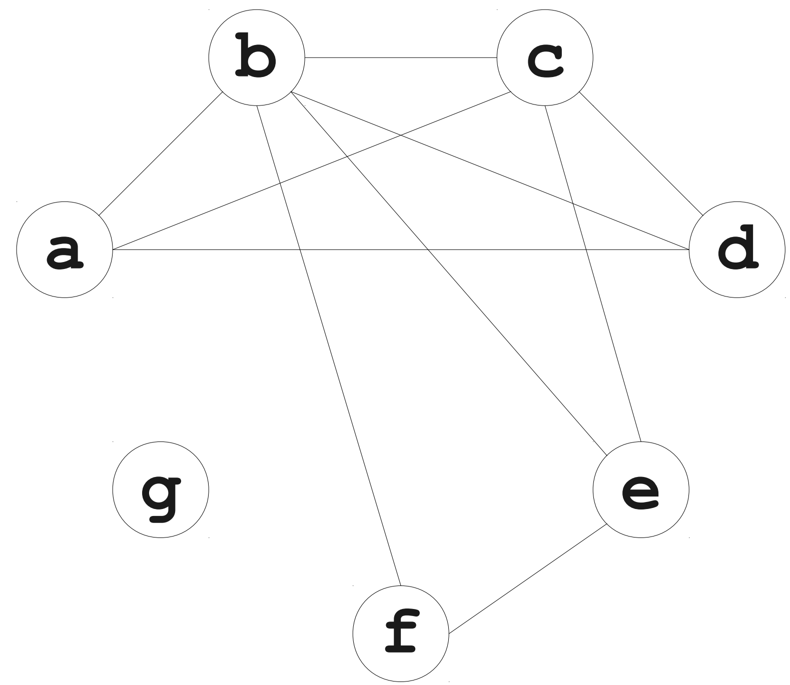
An Entirely Different Approach



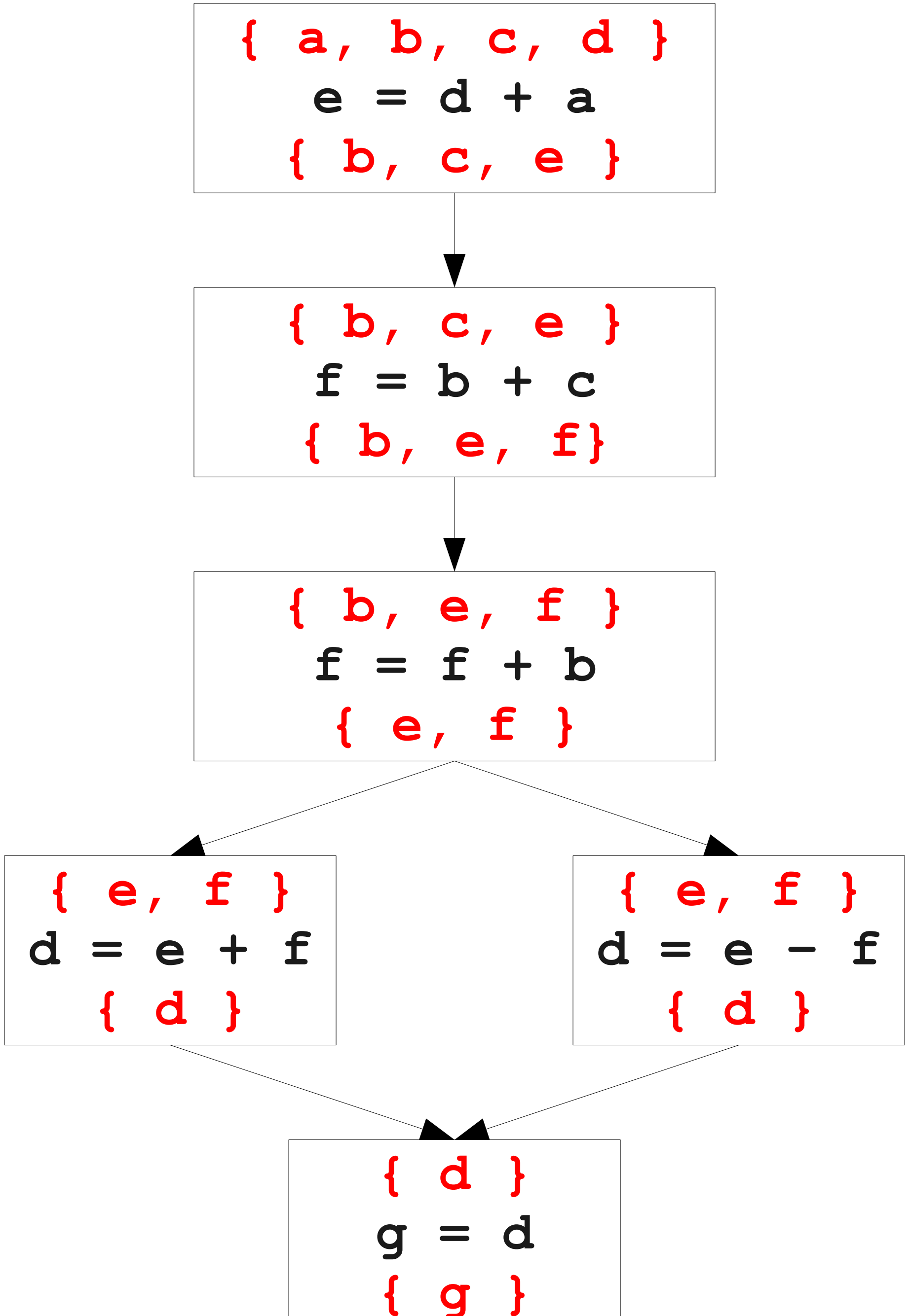


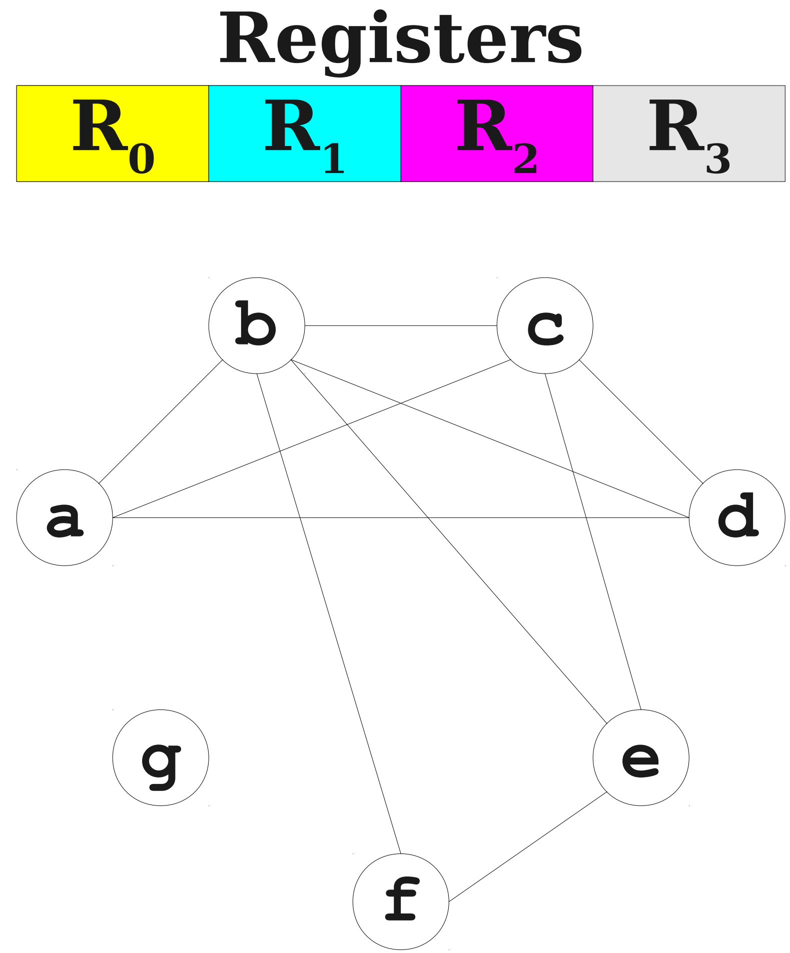
An Entirely Different Approach



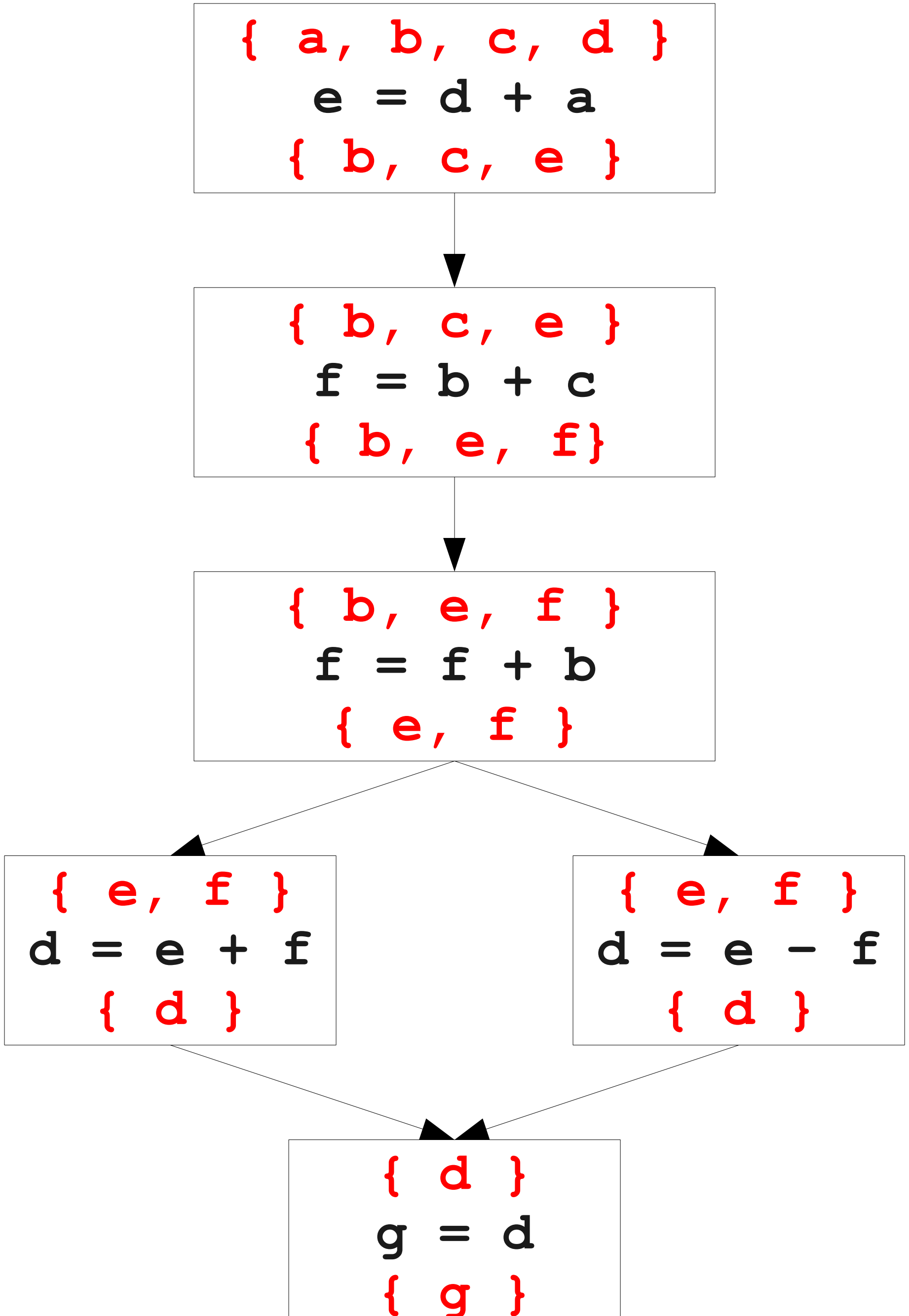


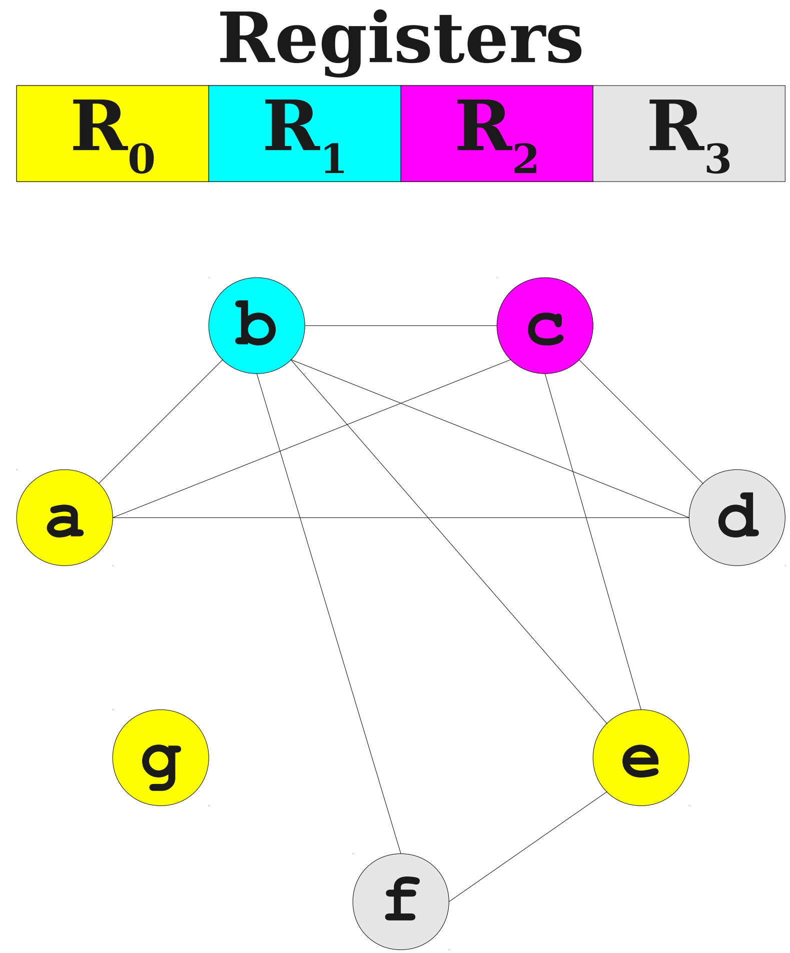
An Entirely Different Approach





An Entirely Different Approach





The Register Interference Graph

- The **register interference graph** (RIG) of a control-flow graph is an undirected graph where
  - Each node is a variable.
  - There is an edge between two variables that are live at the same program point.
- Perform register allocation by assigning each variable a different register from all of its neighbors.
- There's just one catch...

The One Catch

- This problem is equivalent to **graph-coloring**, which is **NP-hard** if there are at least three registers.
- No good polynomial-time algorithms (or even good approximations!) are known for this problem.
- We have to be content with a heuristic that is good enough for RIGs that arise in practice.

The One Catch to The One Catch

The One Catch to The One Catch

If you can figure out a way to assign registers to arbitrary RIGs, you've just proven **P = NP** and will get a **$1,000,000 check** from the Clay Mathematics Institute.

The One Catch to The One Catch

If you can figure out a way to assign registers to arbitrary RIGs, you've just proven **P = NP** and will get a **$1,000,000 check** from the Clay Mathematics Institute.

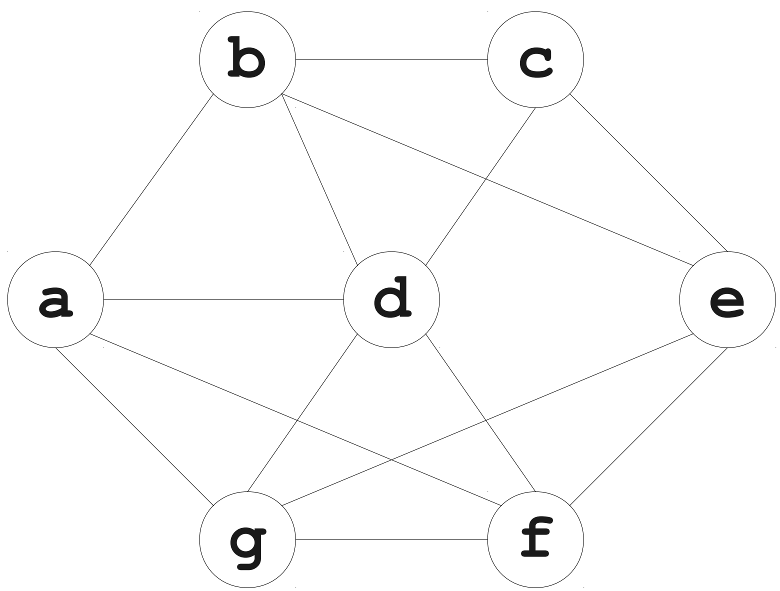
Battling **NP**-Hardness

Chaitin's Algorithm

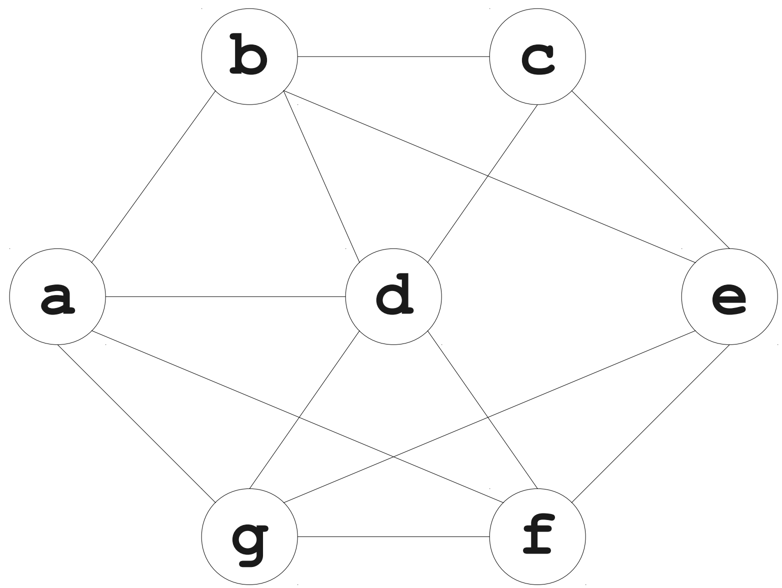
- Intuition:
  - Suppose we are trying to k-color a graph and find a node with fewer than k edges.
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in.
  - Reason: With fewer than k neighbors, some color must be left over.
- Algorithm:
  - Find a node with fewer than k outgoing edges.
  - Remove it from the graph.
  - Recursively color the rest of the graph.
  - Add the node back in.
  - Assign it a valid color.

Chaitin's Algorithm

Chaitin's Algorithm



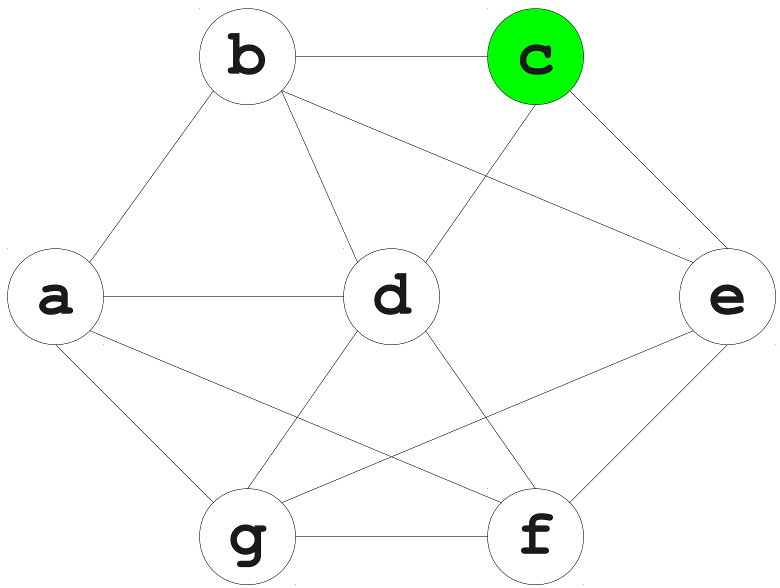
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

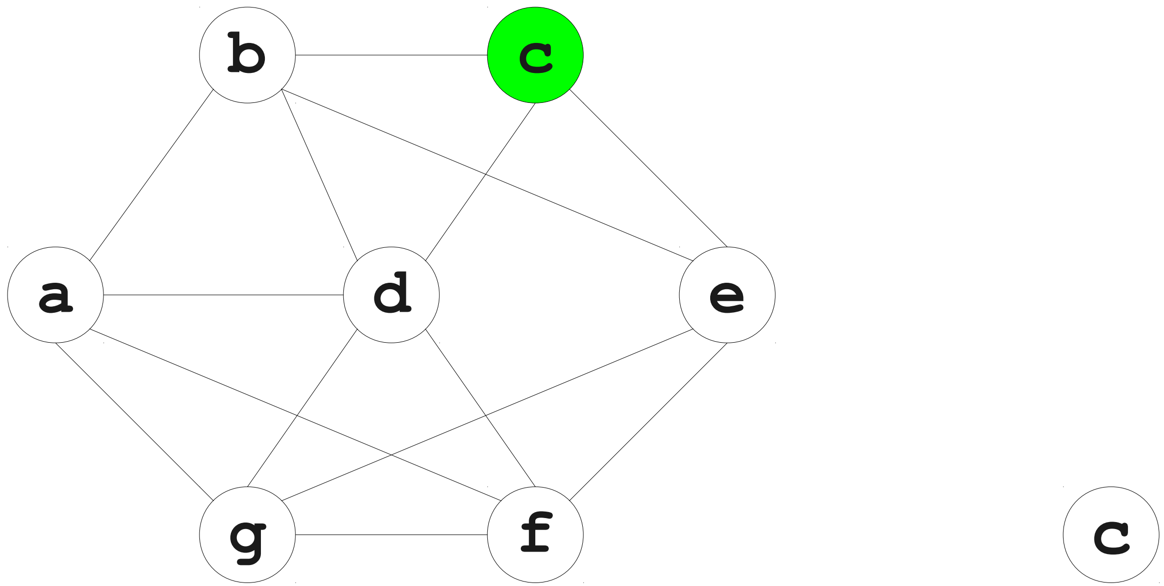
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

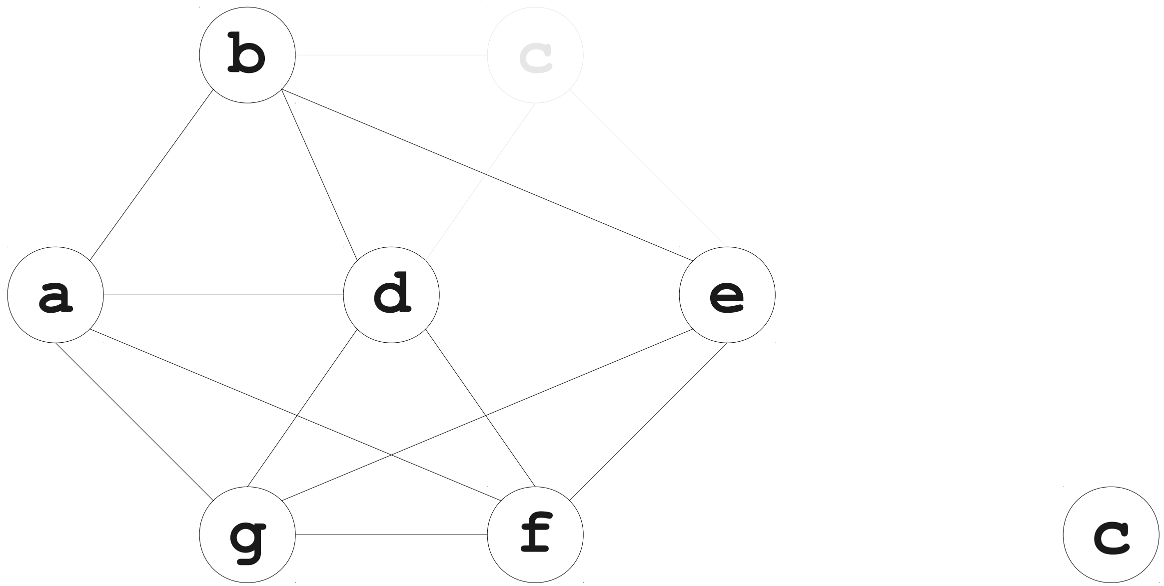
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

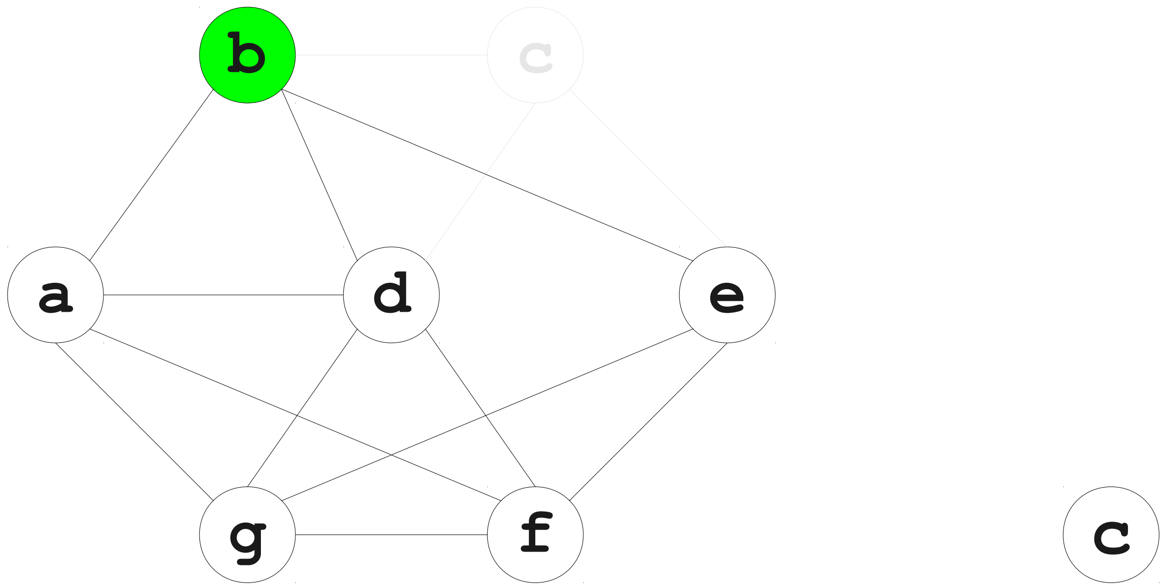
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

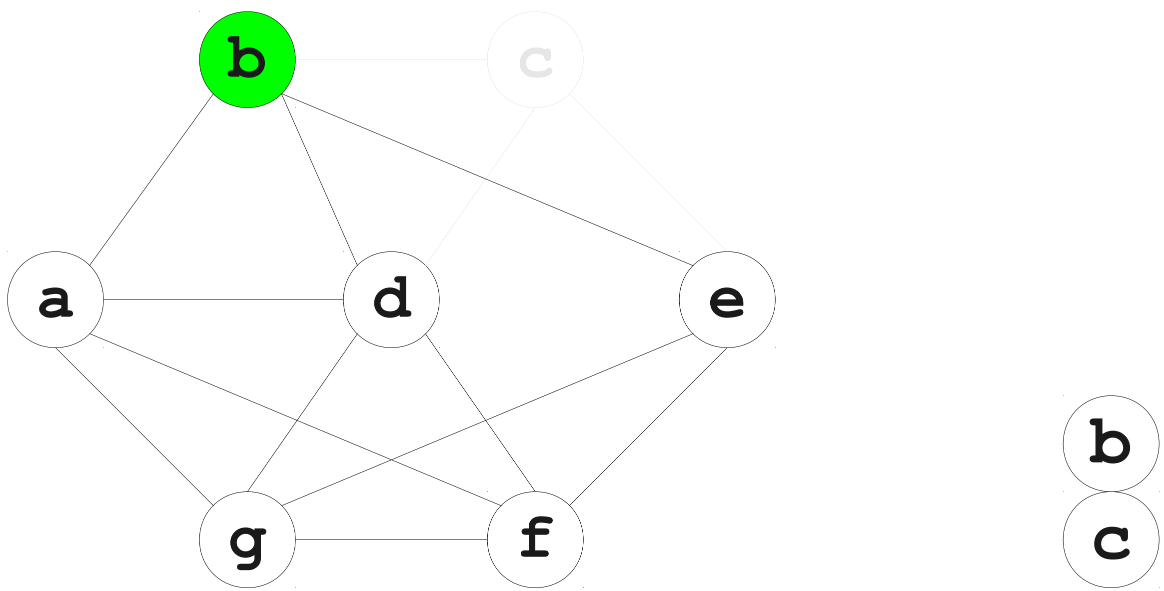
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

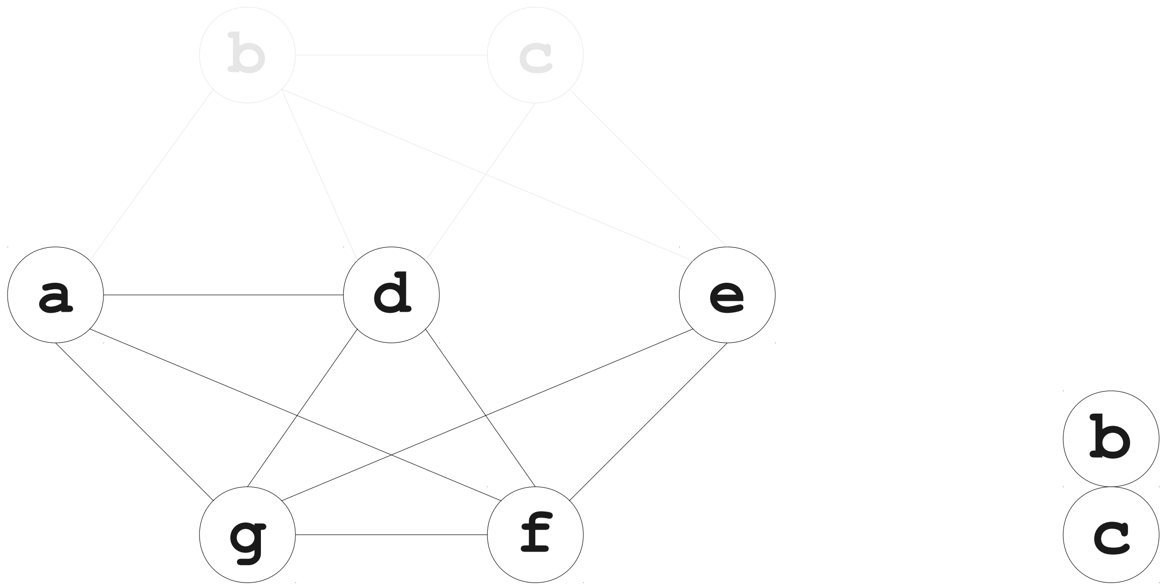
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

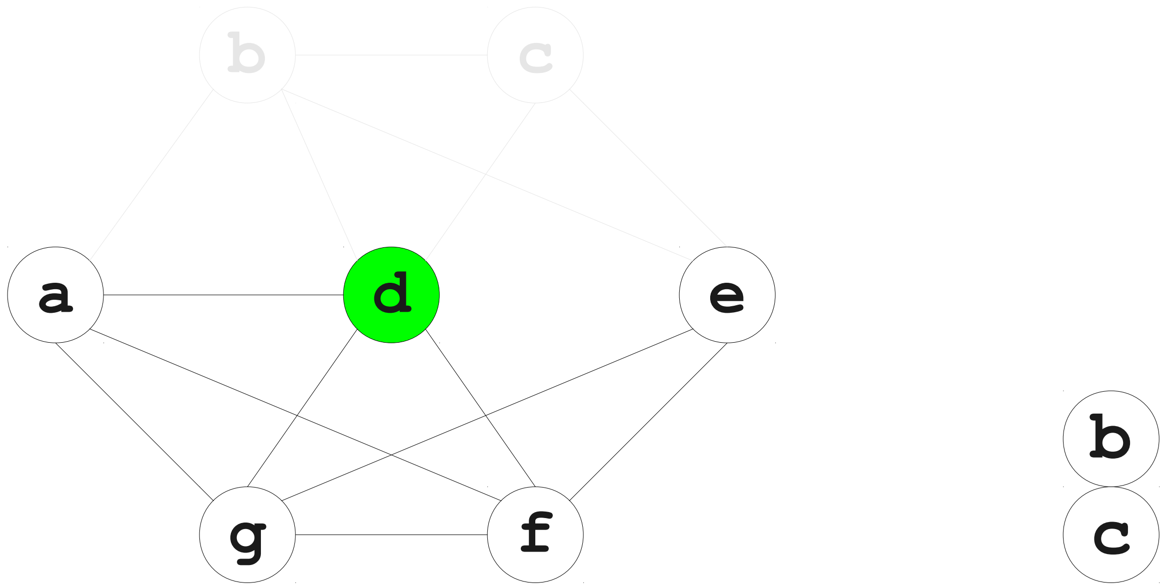
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

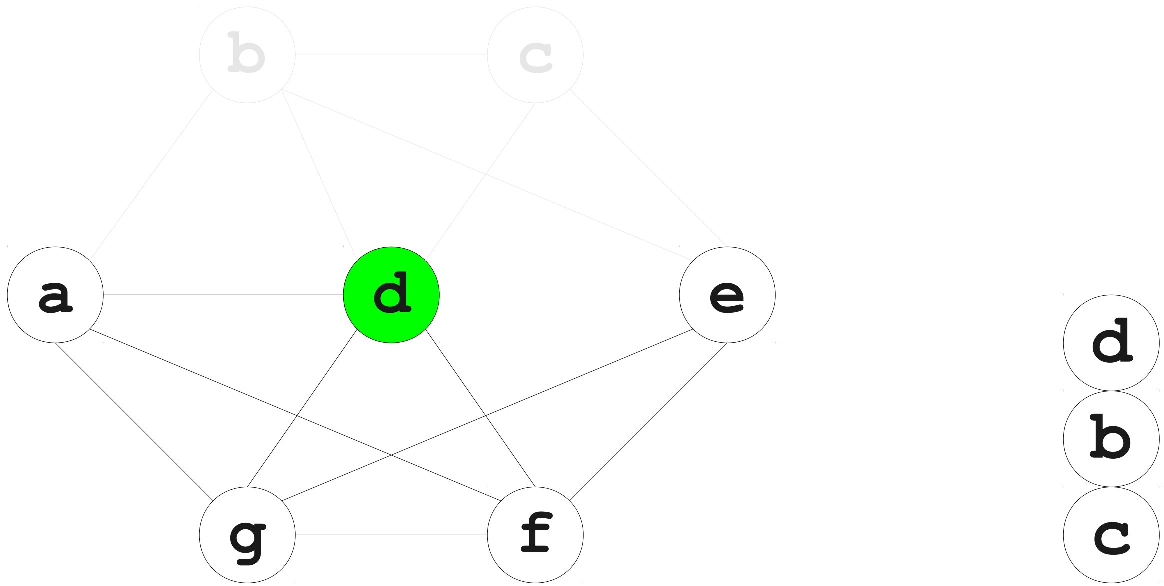
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

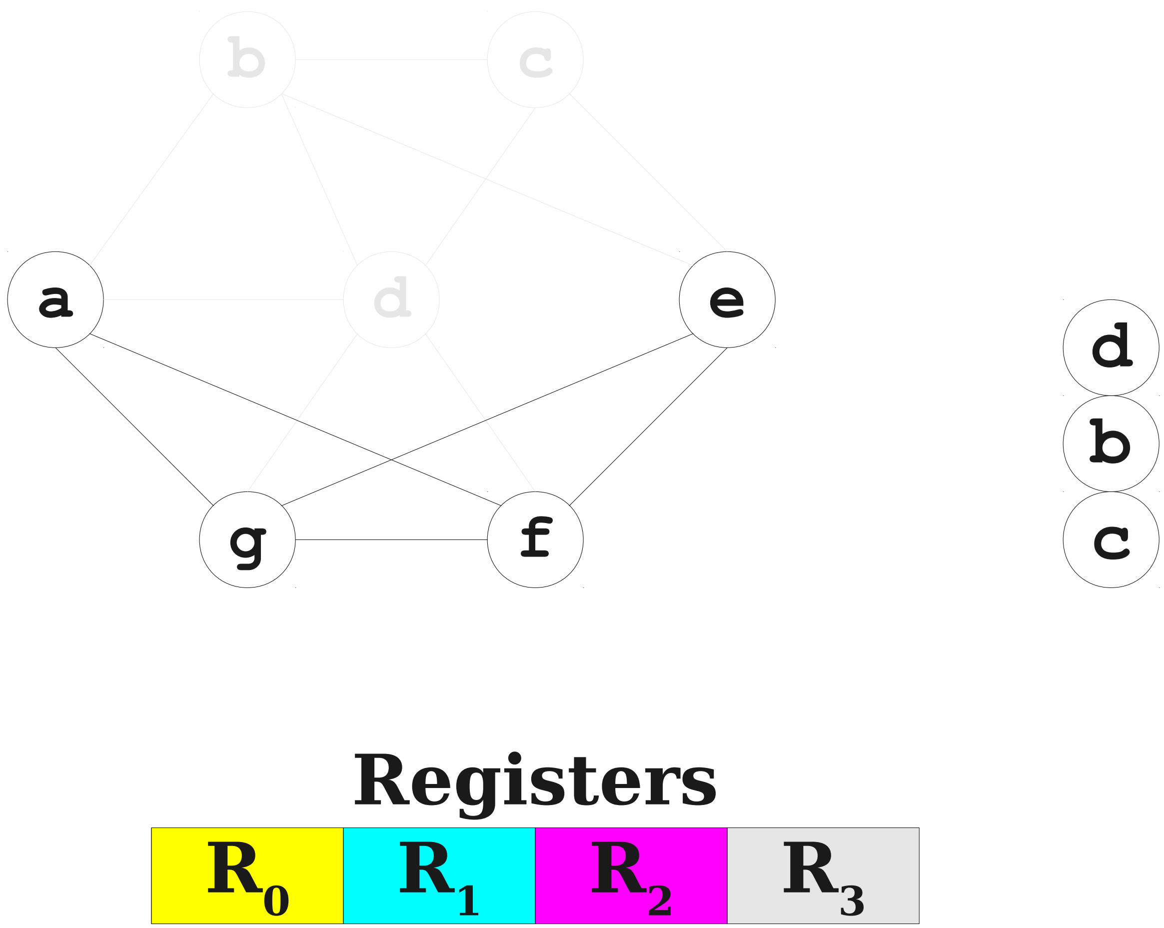
Chaitin's Algorithm



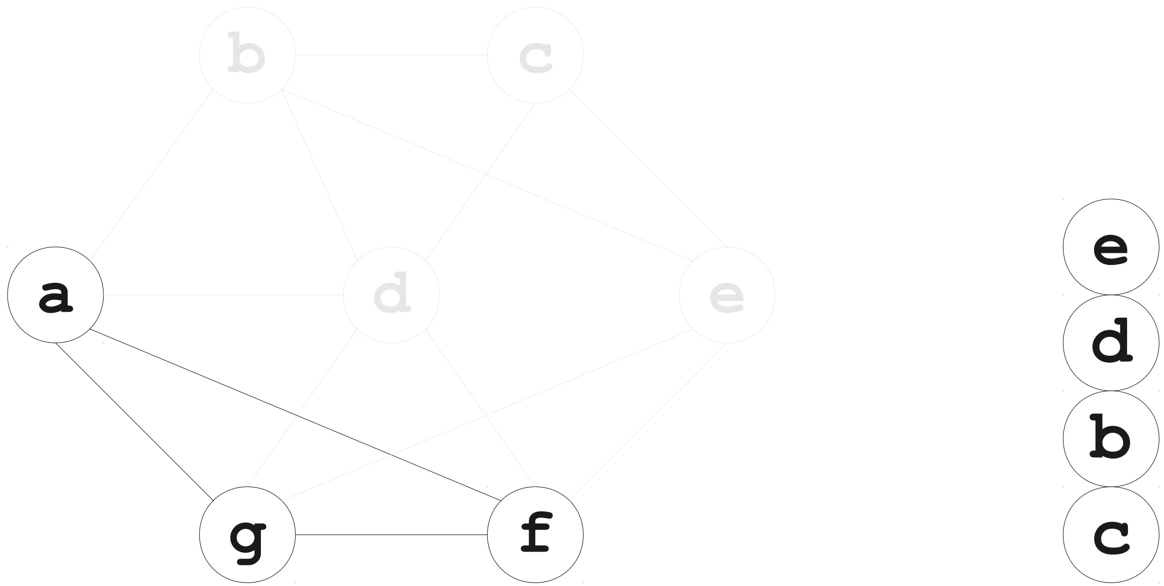
Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



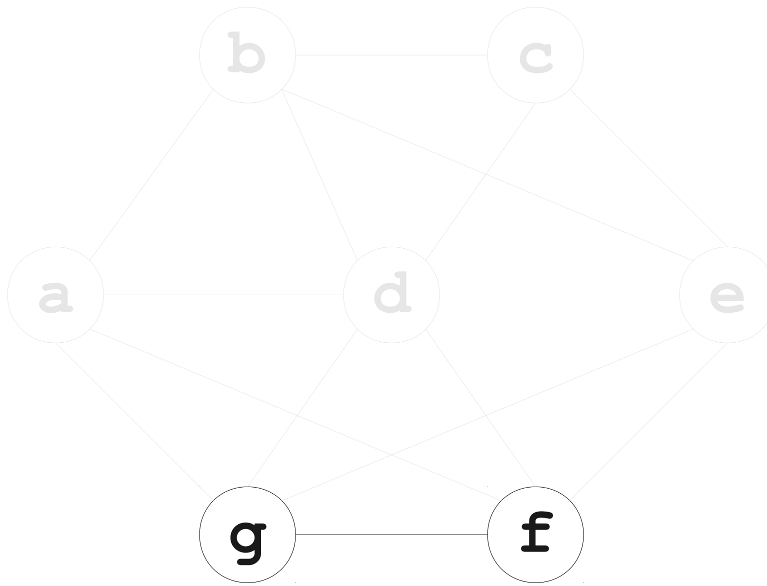
Chaitin's Algorithm

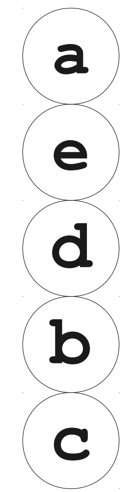


Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm

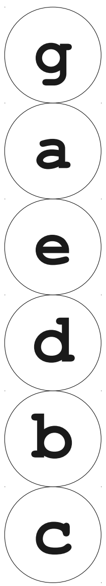




Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



Registers

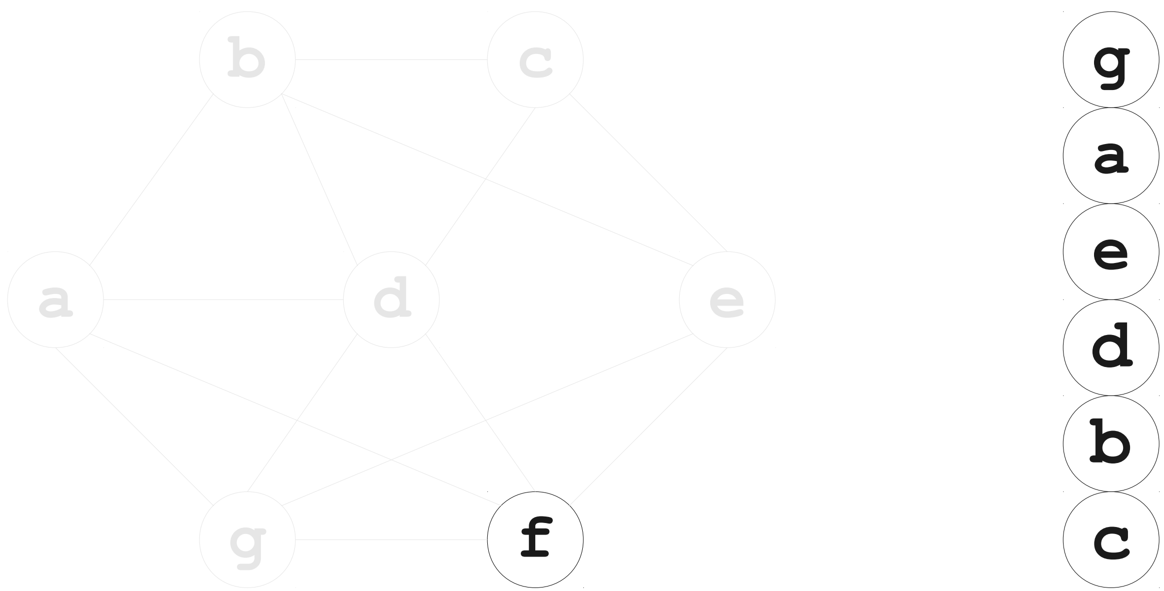
R_0	R_1	R_2	R_3

Chaitin's Algorithm

Registers

R_0	R_1	R_2	R_3

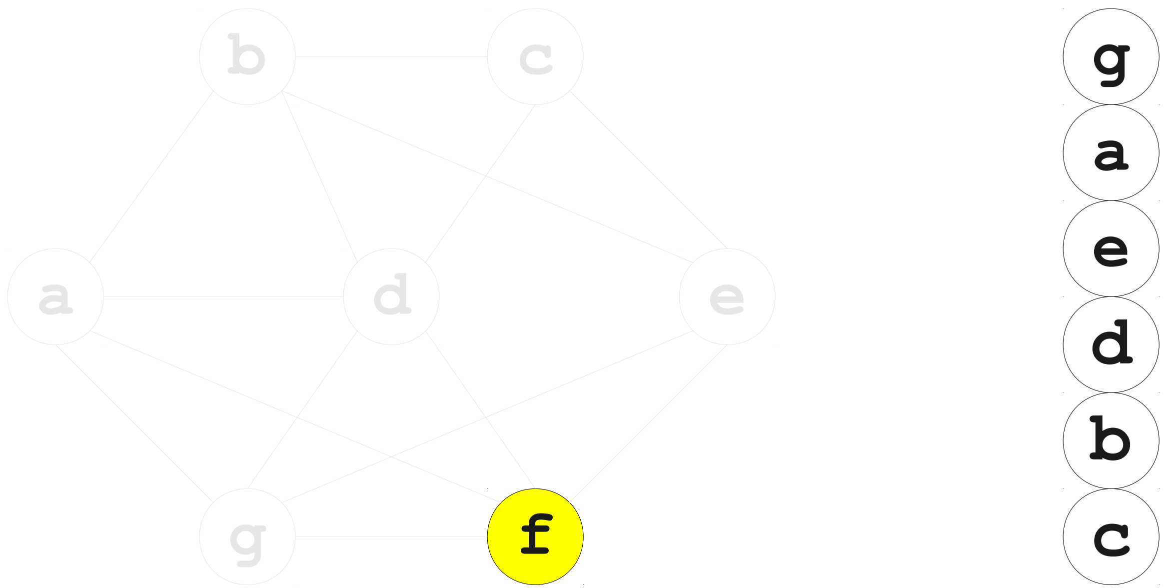
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

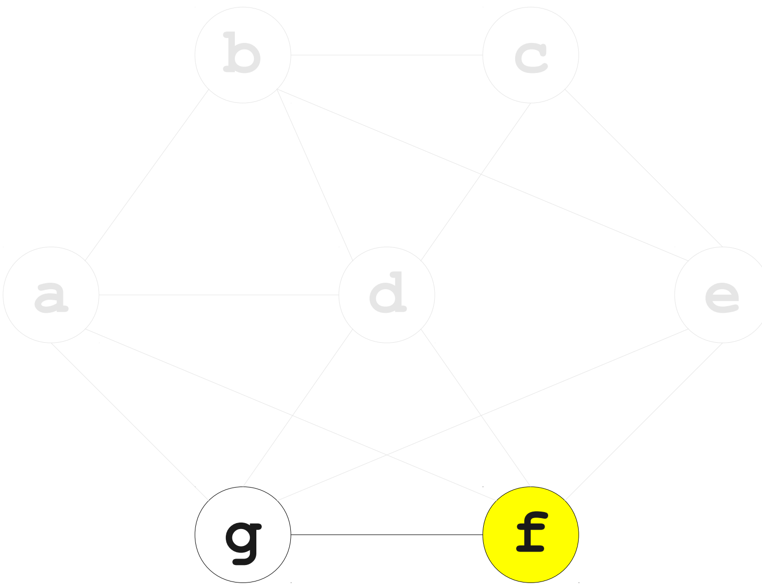
Chaitin's Algorithm

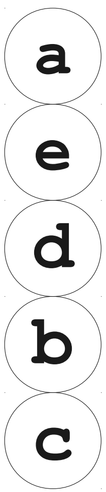


Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm

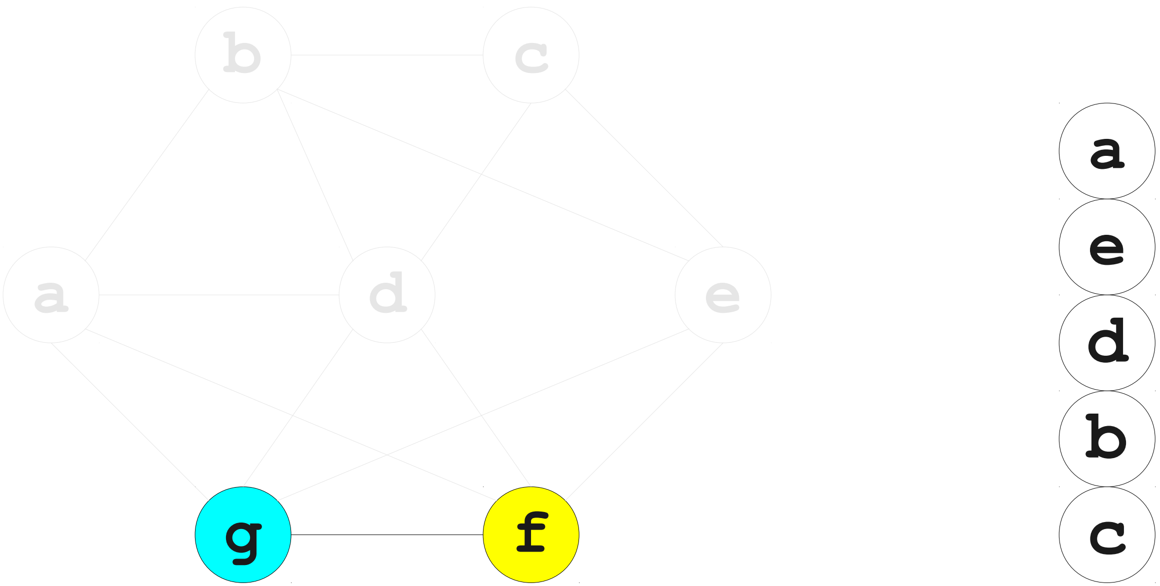




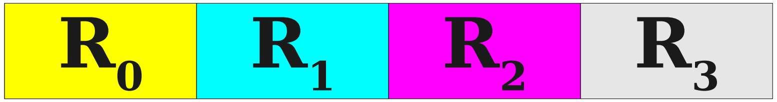
Registers

R_0	R_1	R_2	R_3

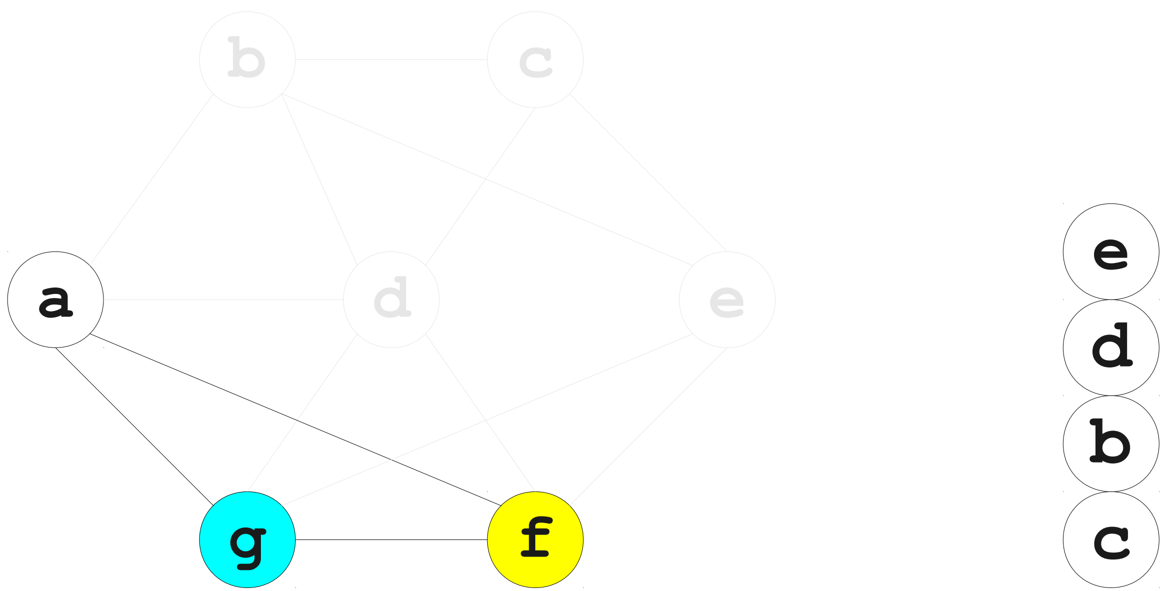
Chaitin's Algorithm



Registers



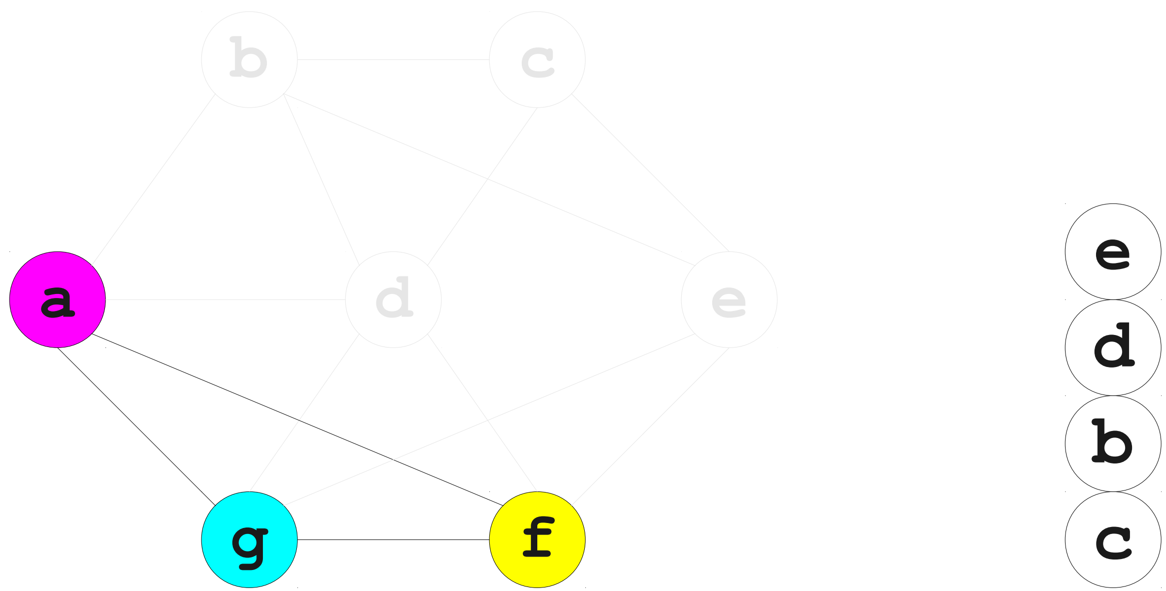
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

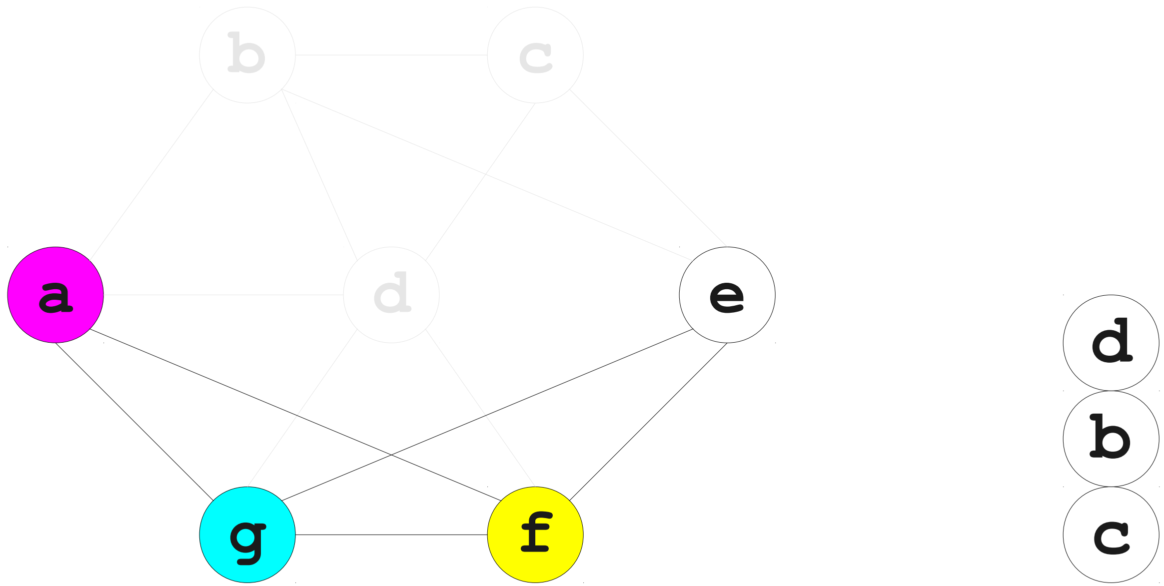
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

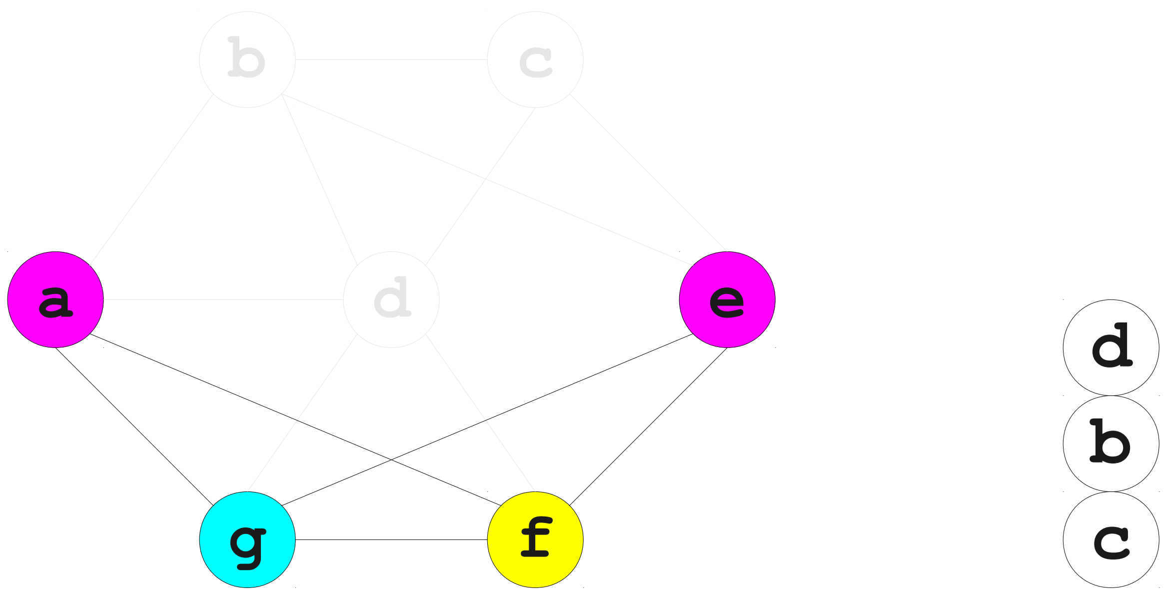
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

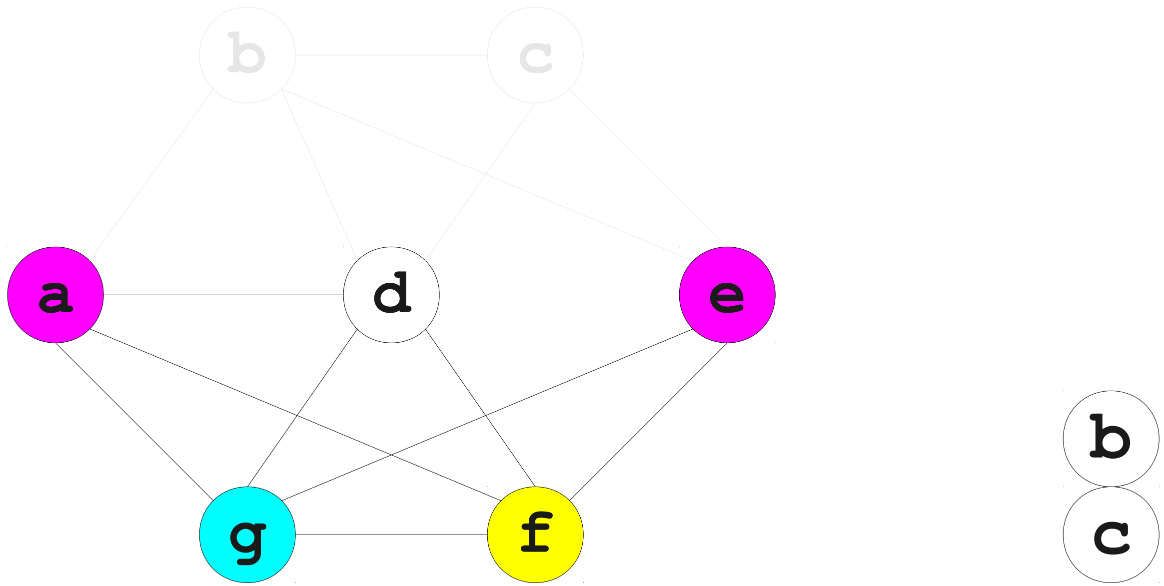
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

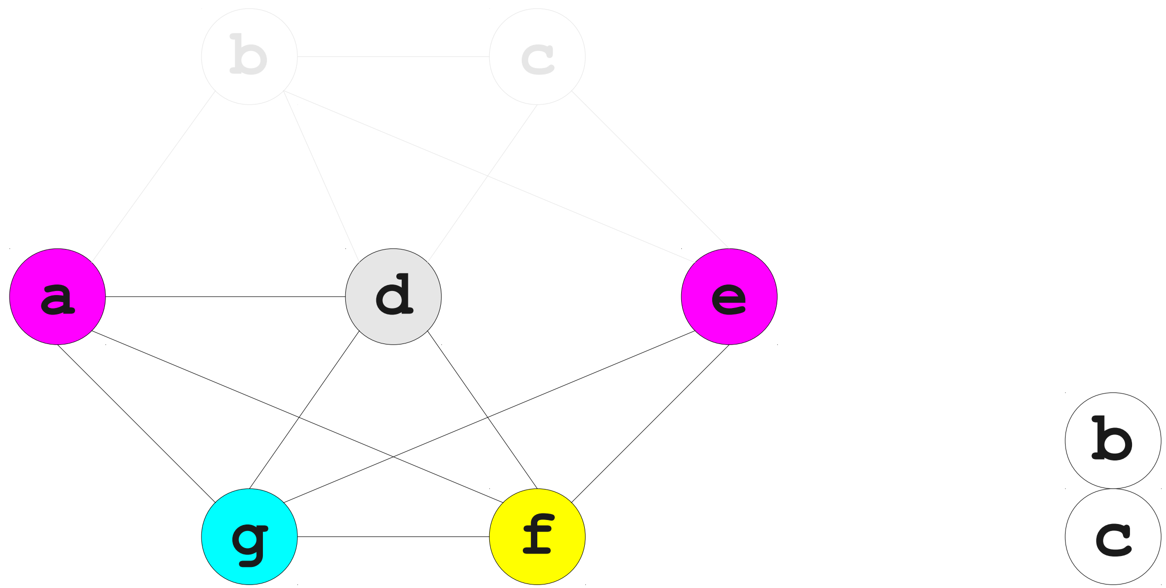
Chaitin's Algorithm



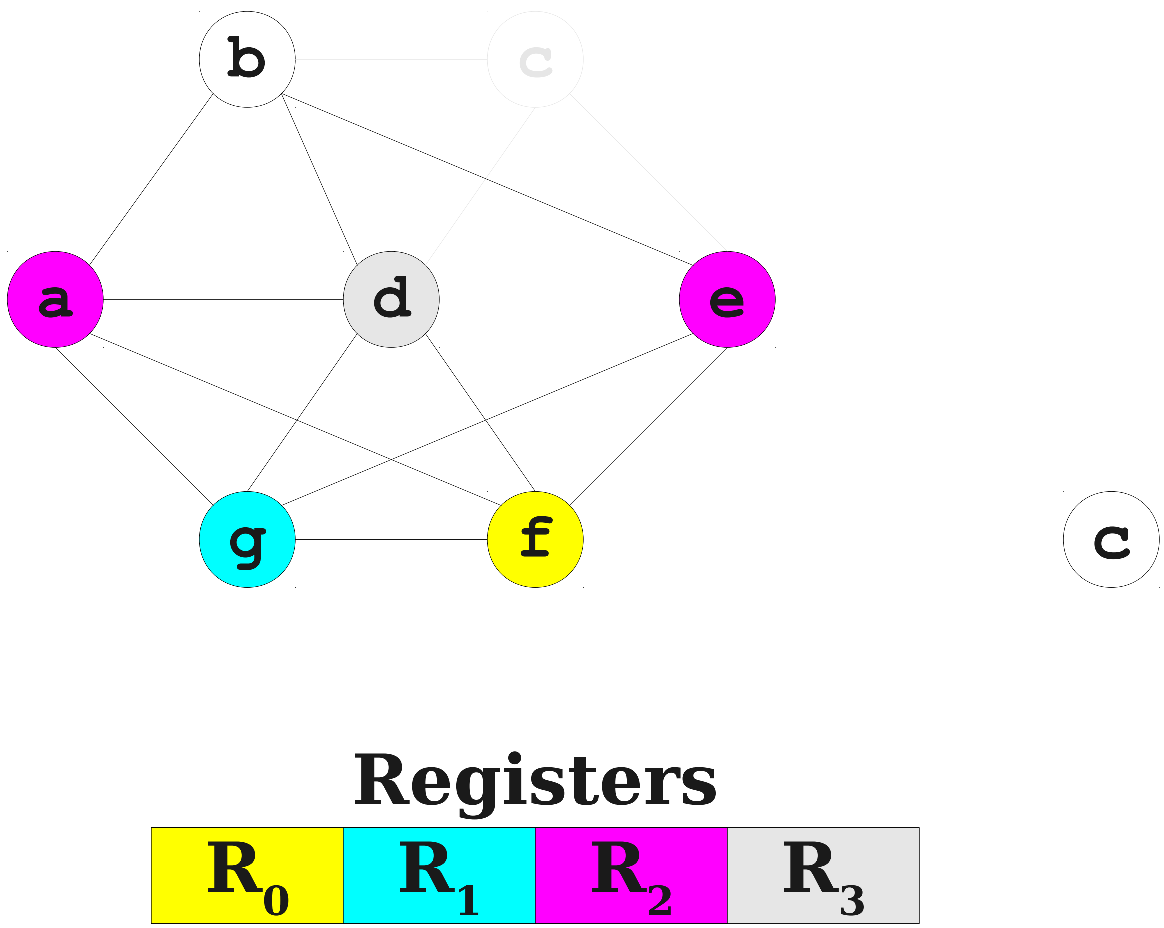
Registers

R_0	R_1	R_2	R_3

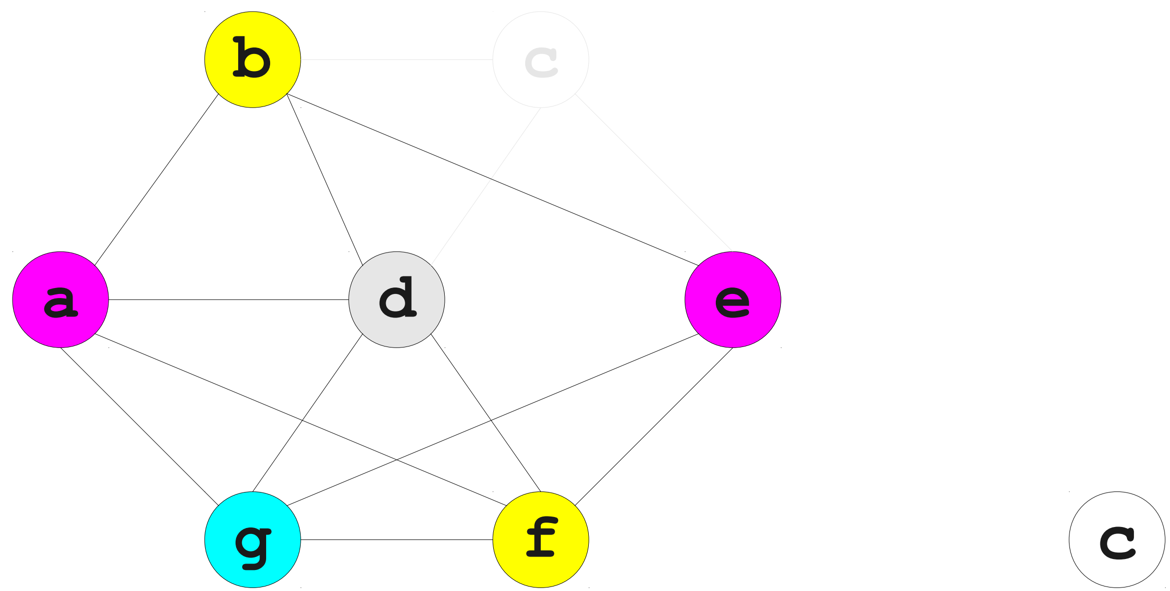
Chaitin's Algorithm



Chaitin's Algorithm



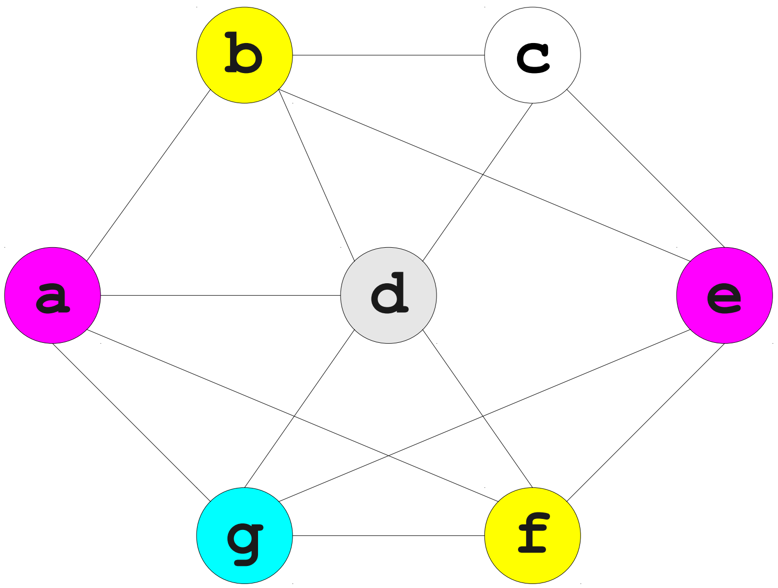
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

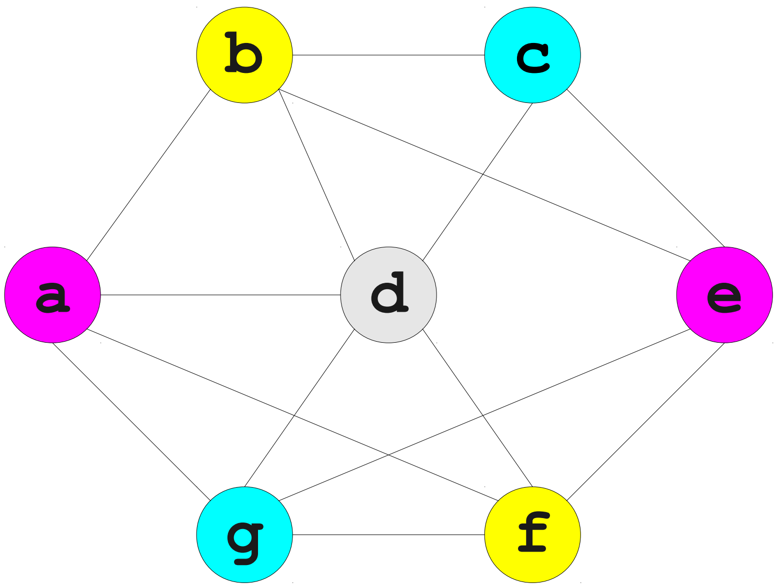
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



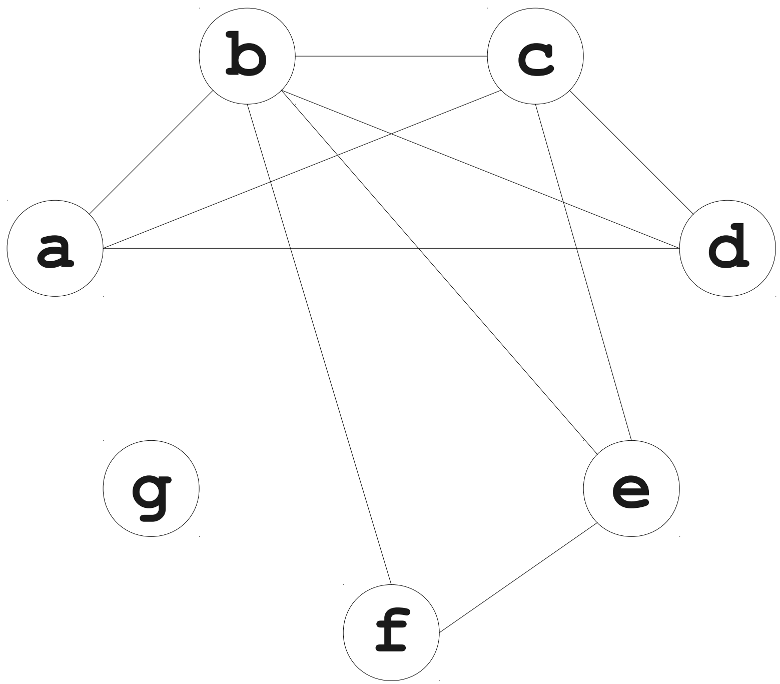
Registers

R_0	R_1	R_2	R_3

One Problem

- What if we can't find a node with fewer than *k* neighbors?
- Choose and remove an arbitrary node, marking it “troublesome.”
    - Use heuristics to choose which one.
- When adding node back in, it may be possible to find a valid color.
- Otherwise, we have to spill that node.

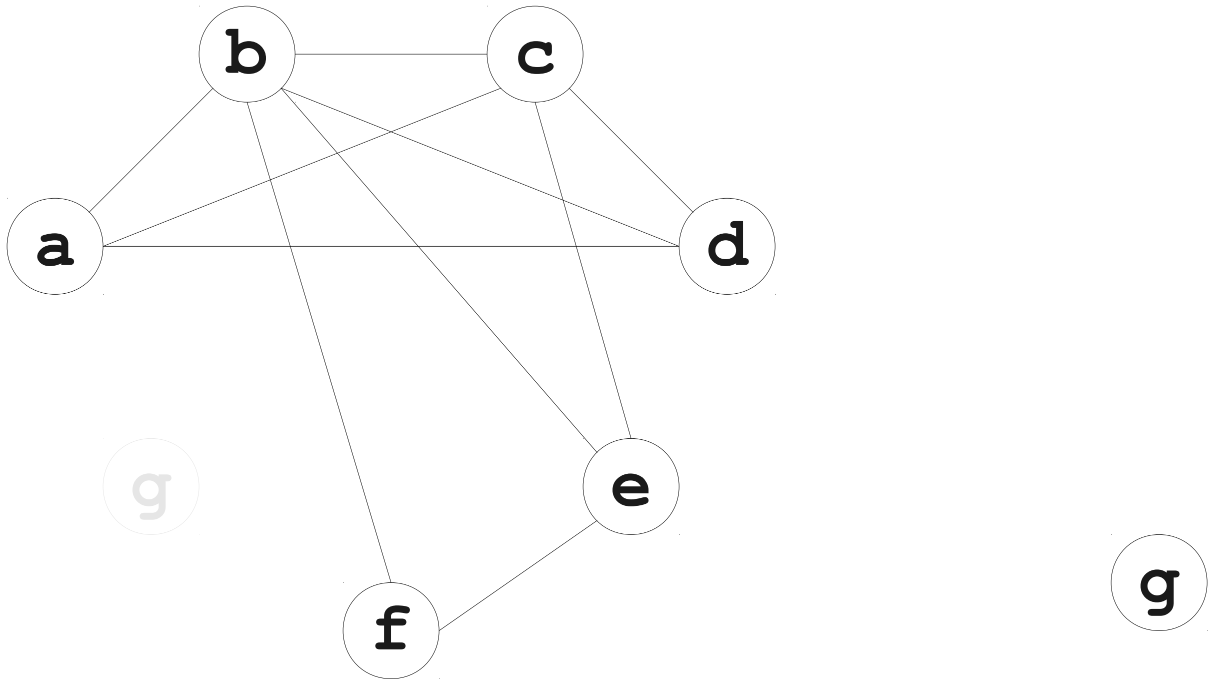
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

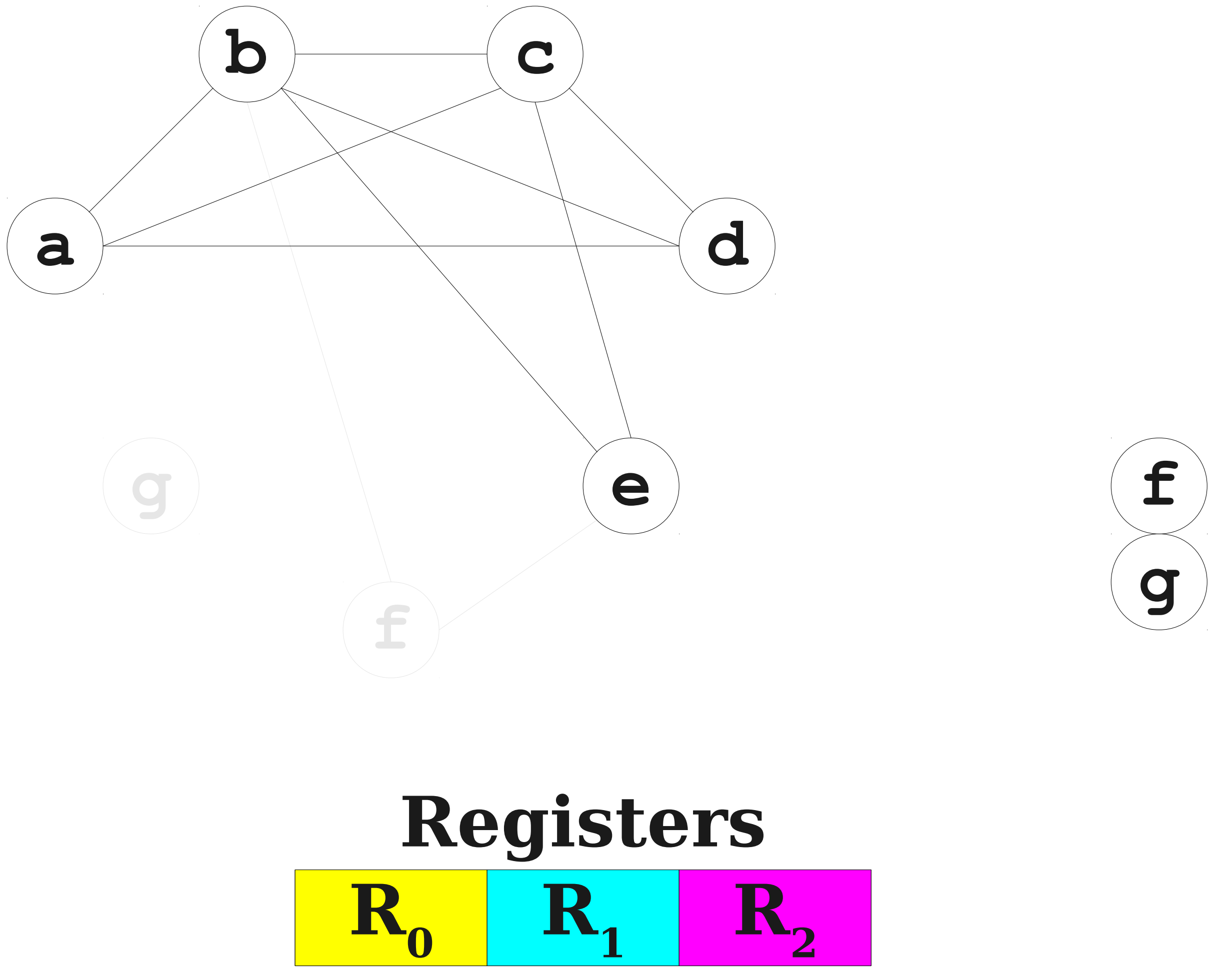
Chaitin's Algorithm Reloaded



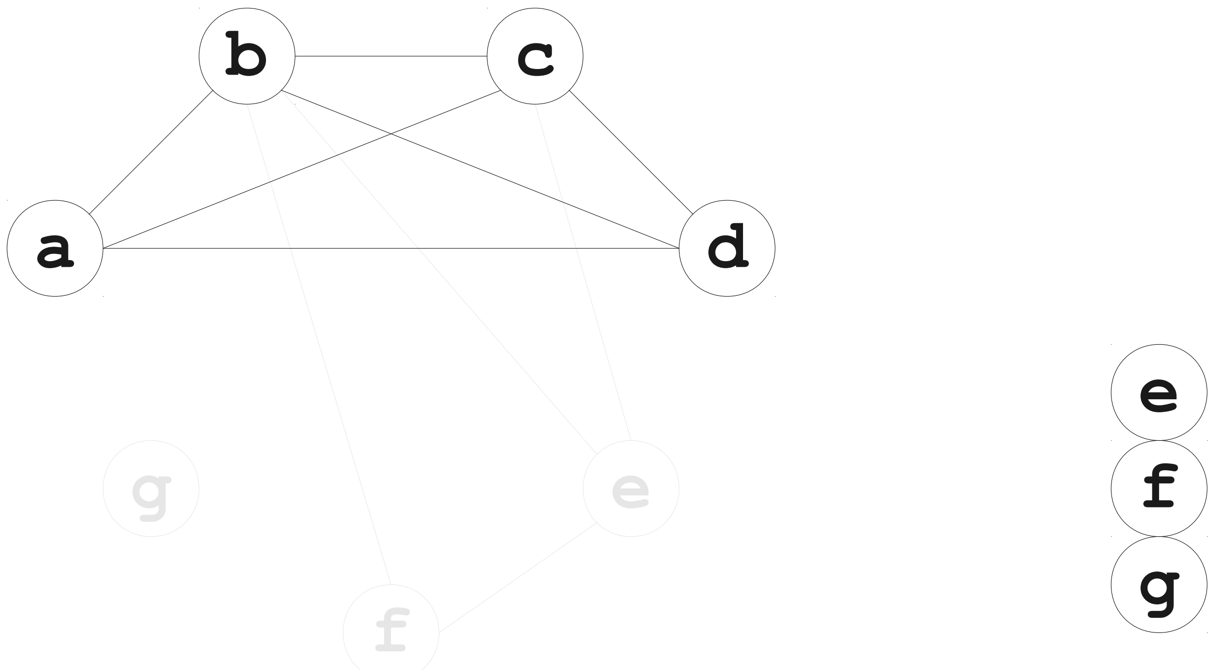
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



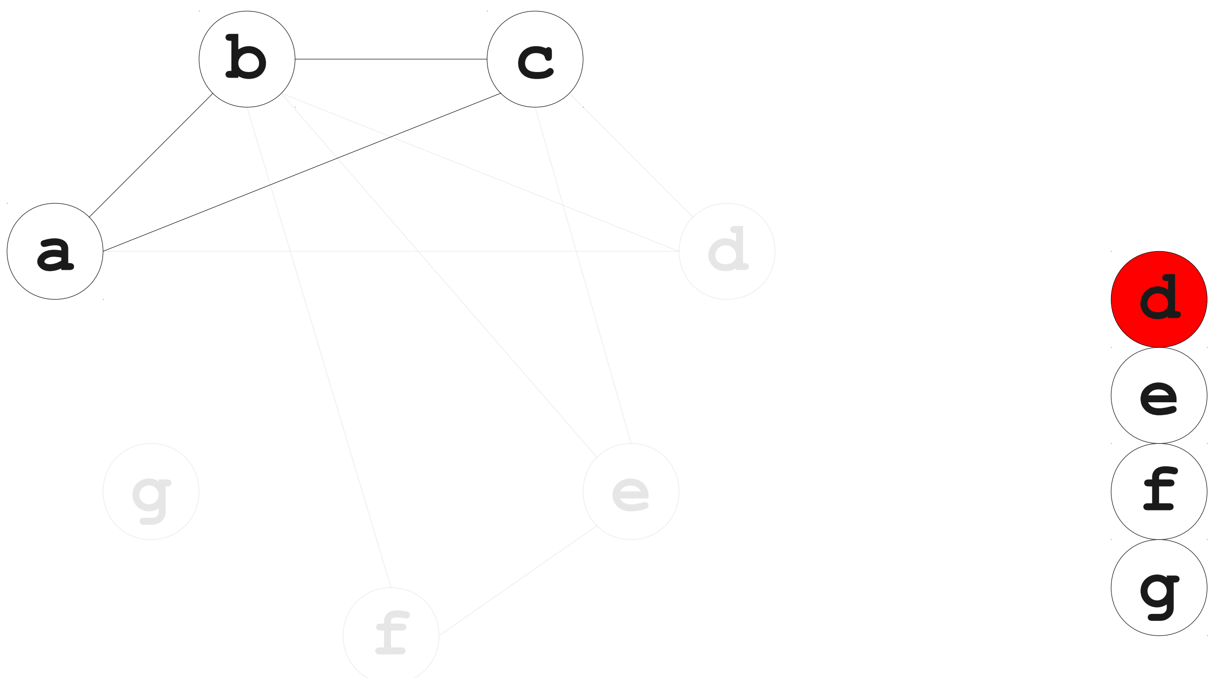
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



Registers

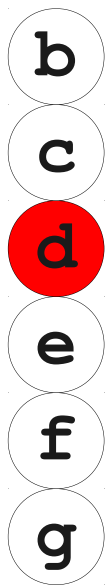
R_0	R_1	R_2

Chaitin's Algorithm Reloaded

Registers

R_0	R_1	R_2

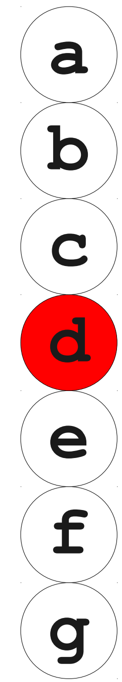
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

Registers

R_0	R_1	R_2

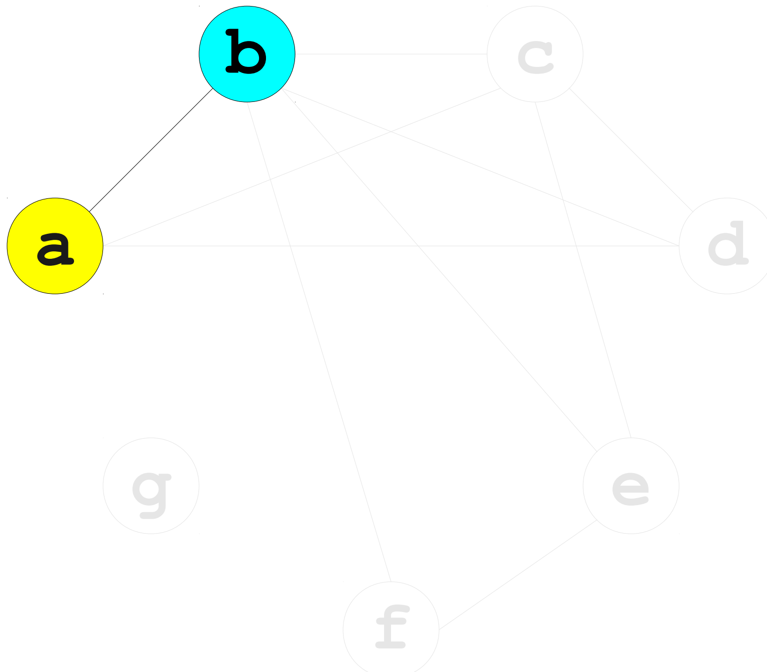
Chaitin's Algorithm Reloaded

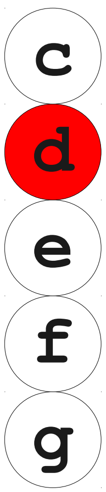
Chaitin's Algorithm Reloaded

Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

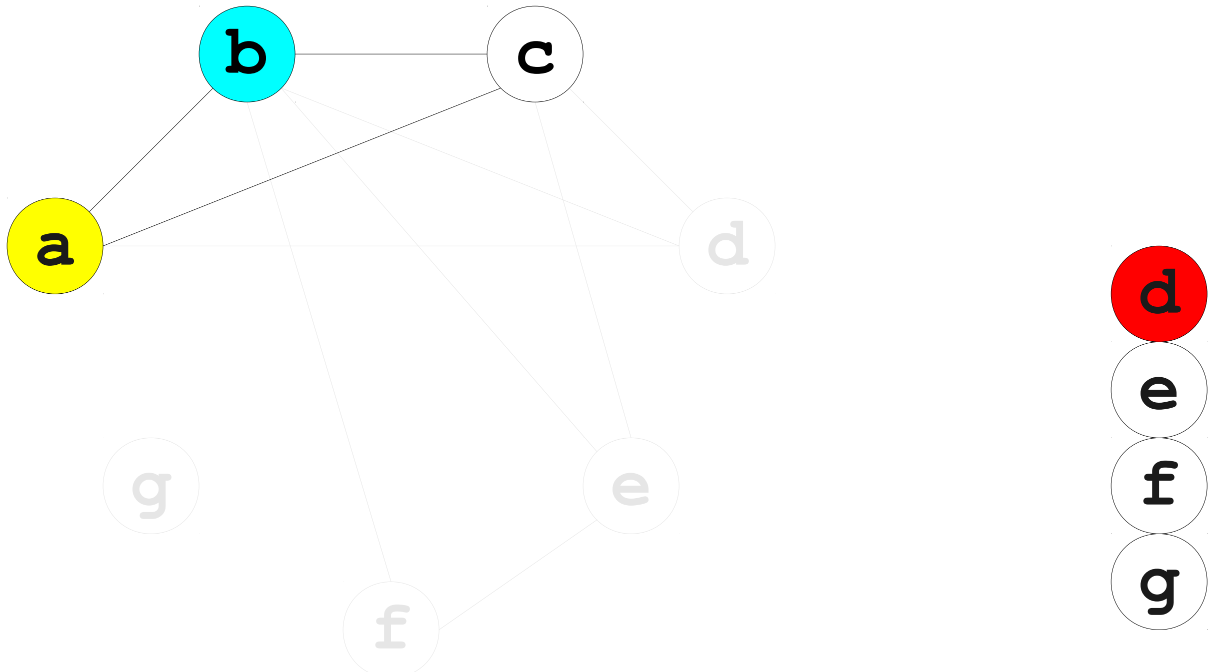




Registers

R_0	R_1	R_2

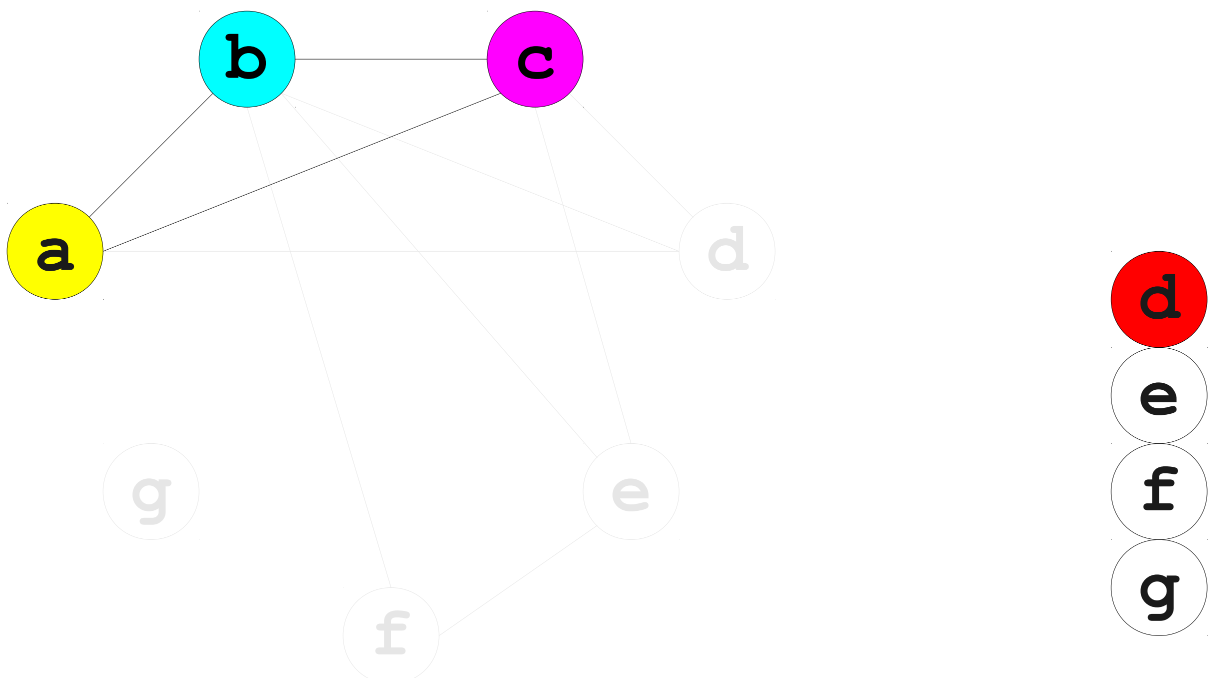
Chaitin's Algorithm Reloaded



Registers

R$_0$	R$_1$	R$_2$

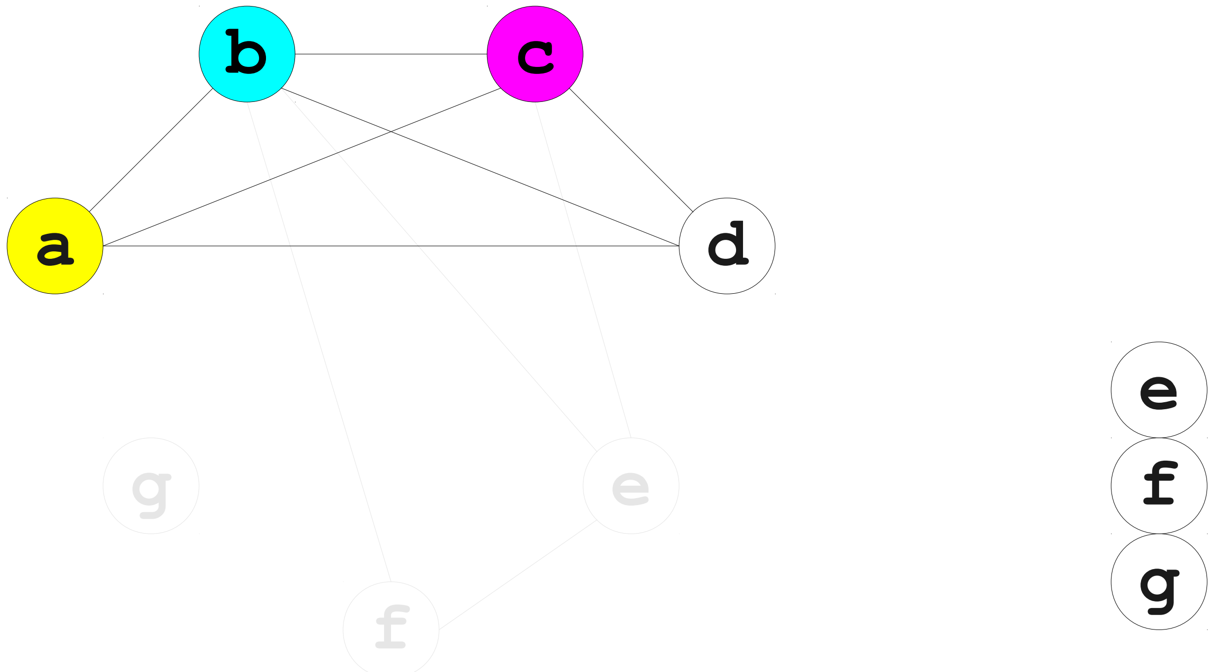
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

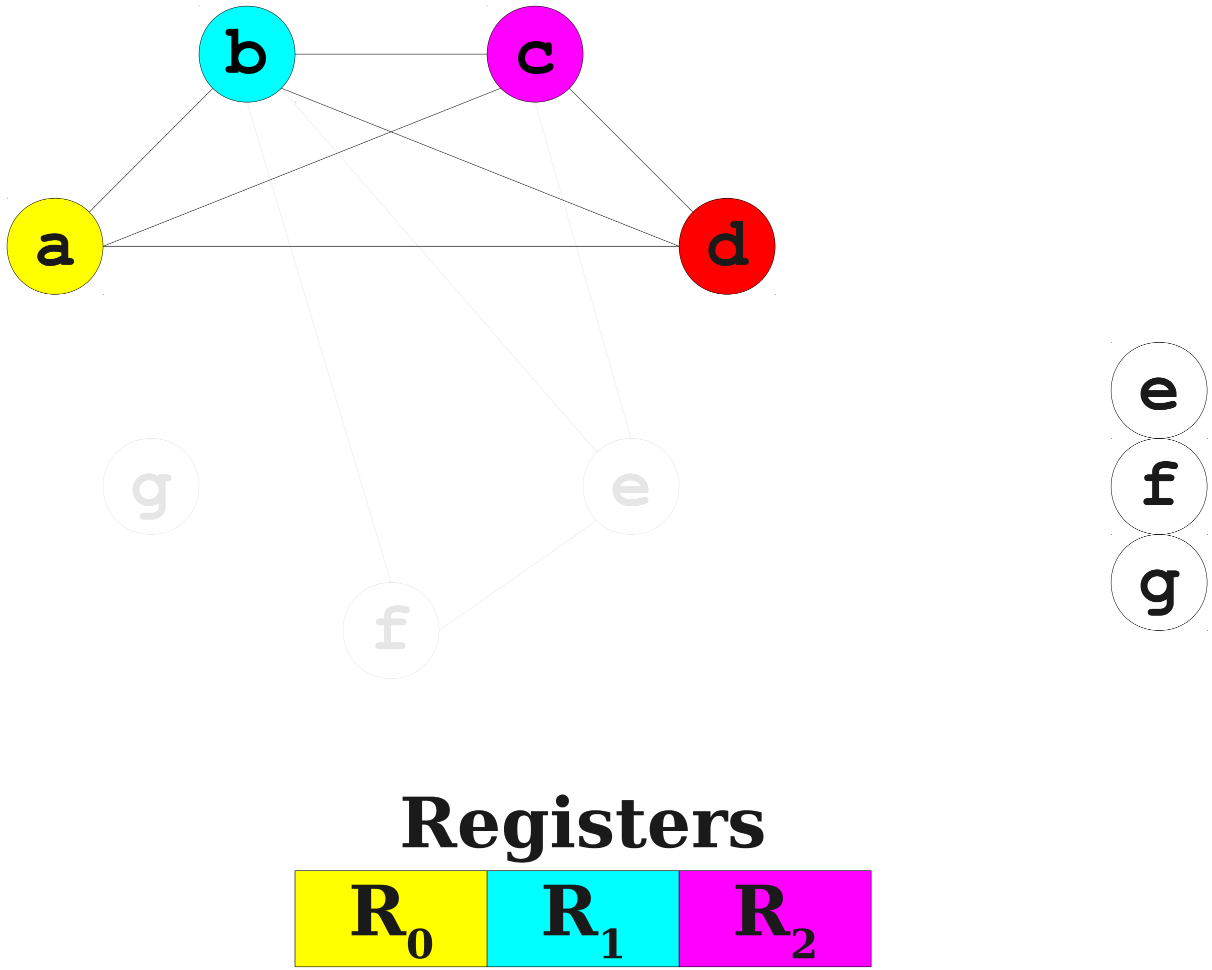
Chaitin's Algorithm Reloaded



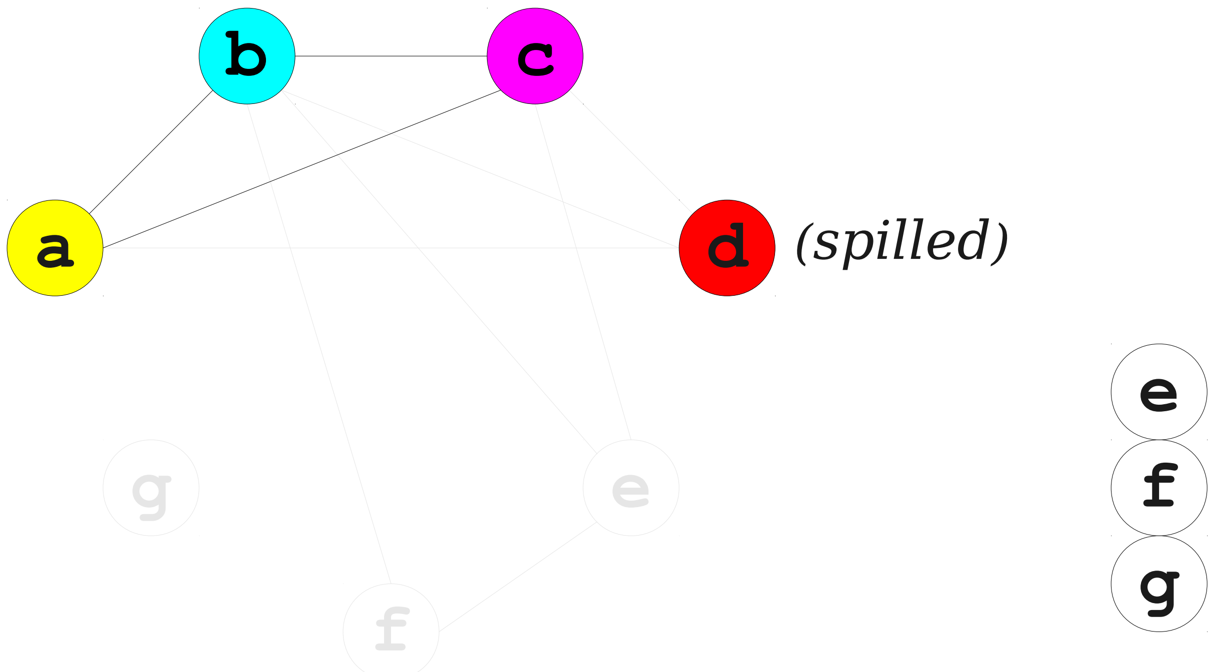
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



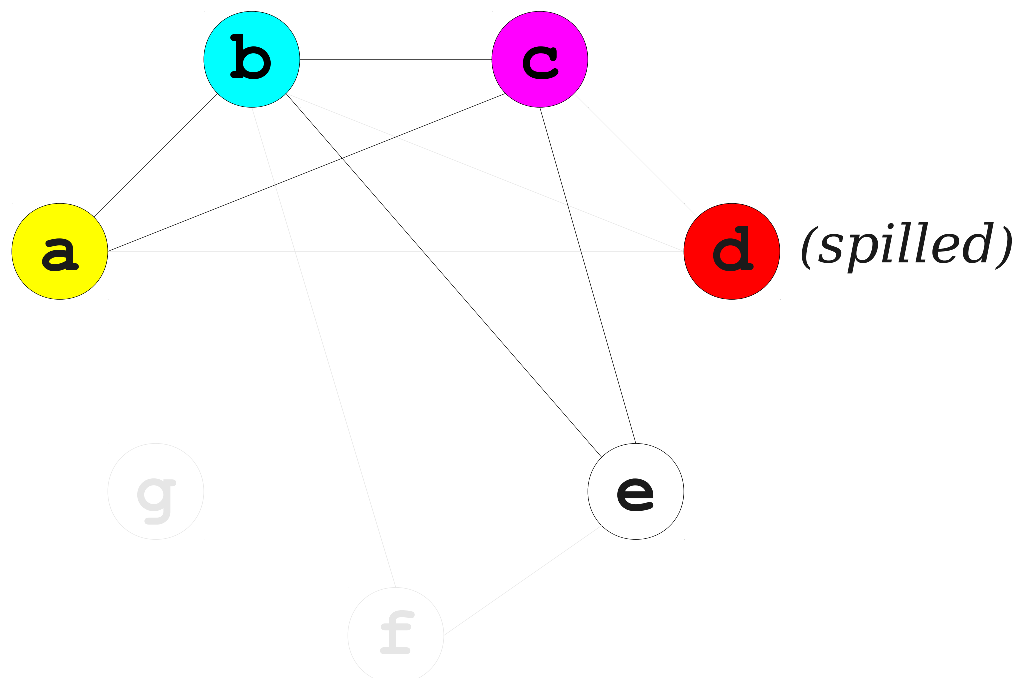
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

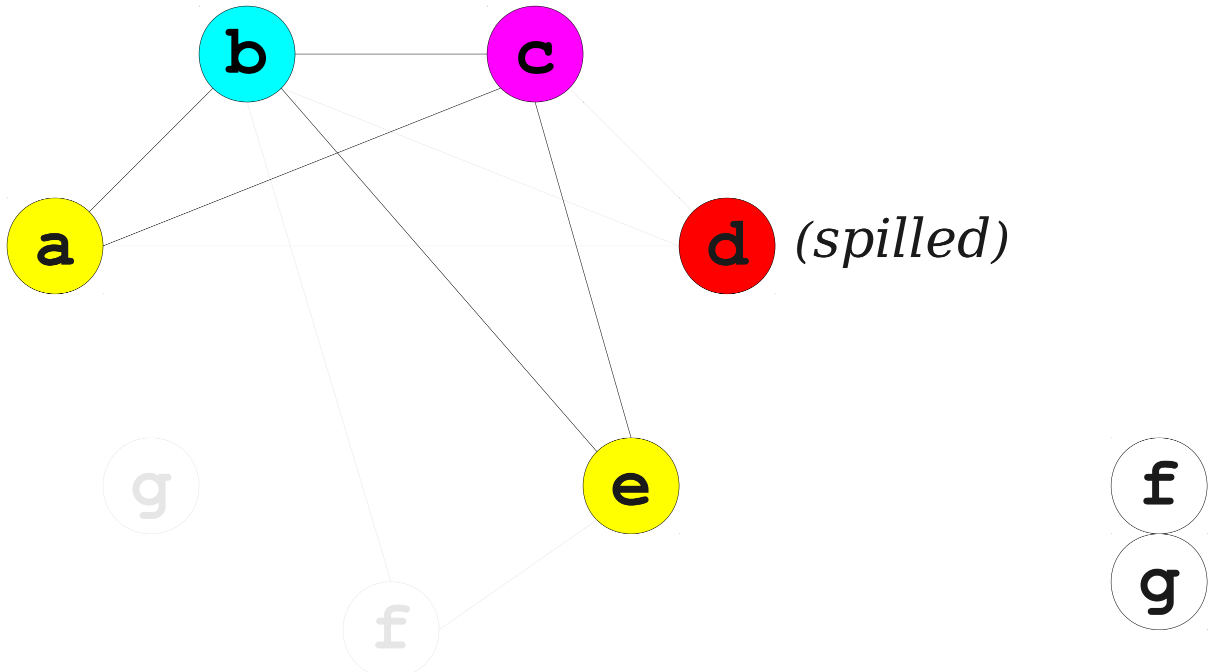
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

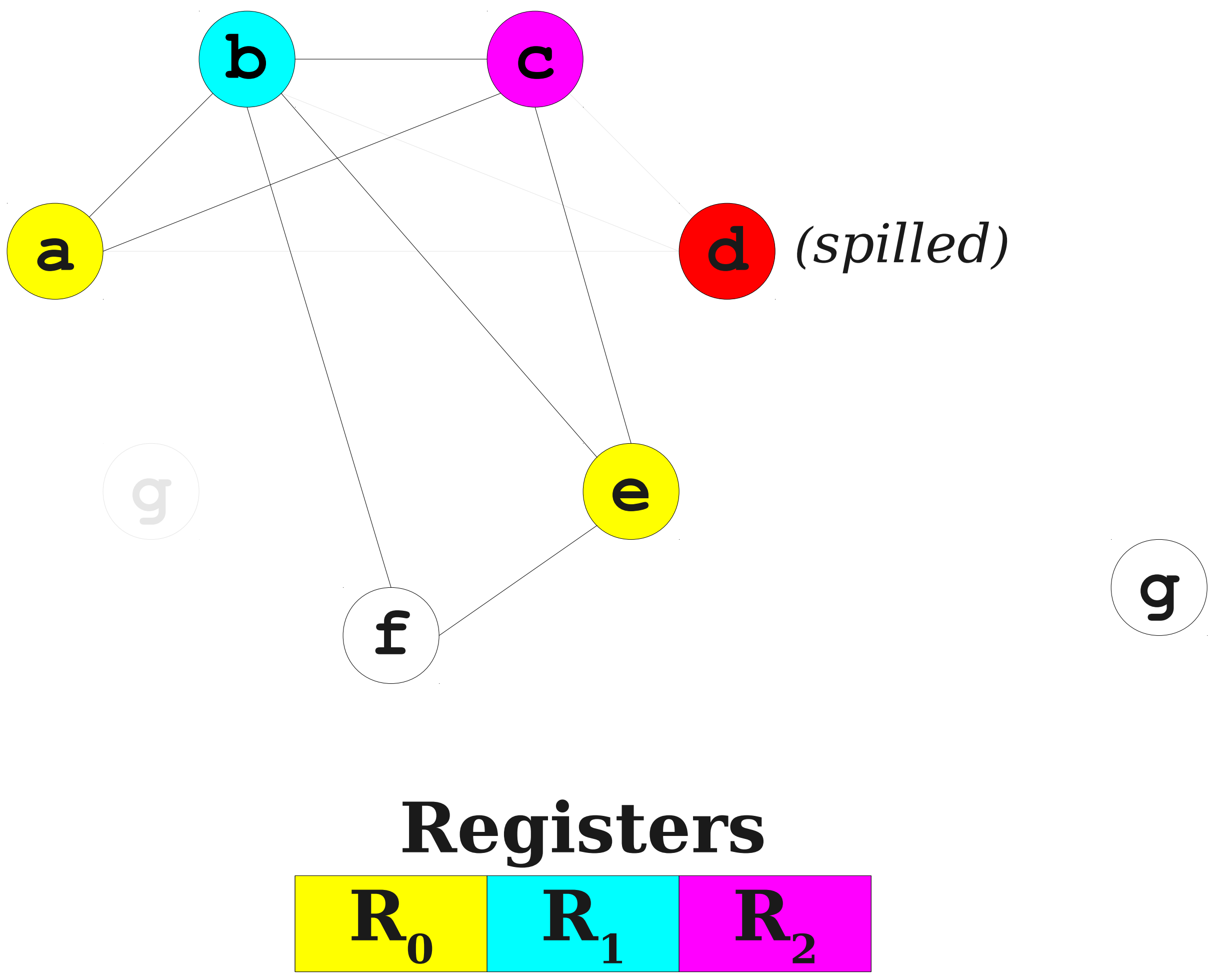
Chaitin's Algorithm Reloaded



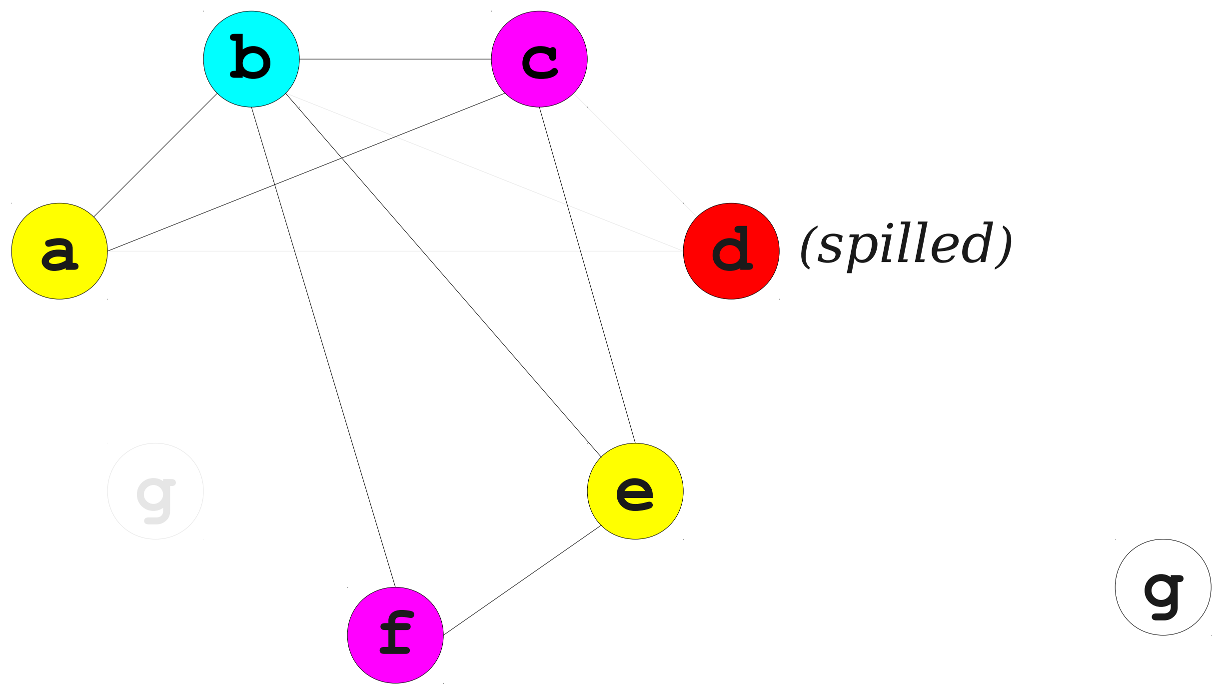
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



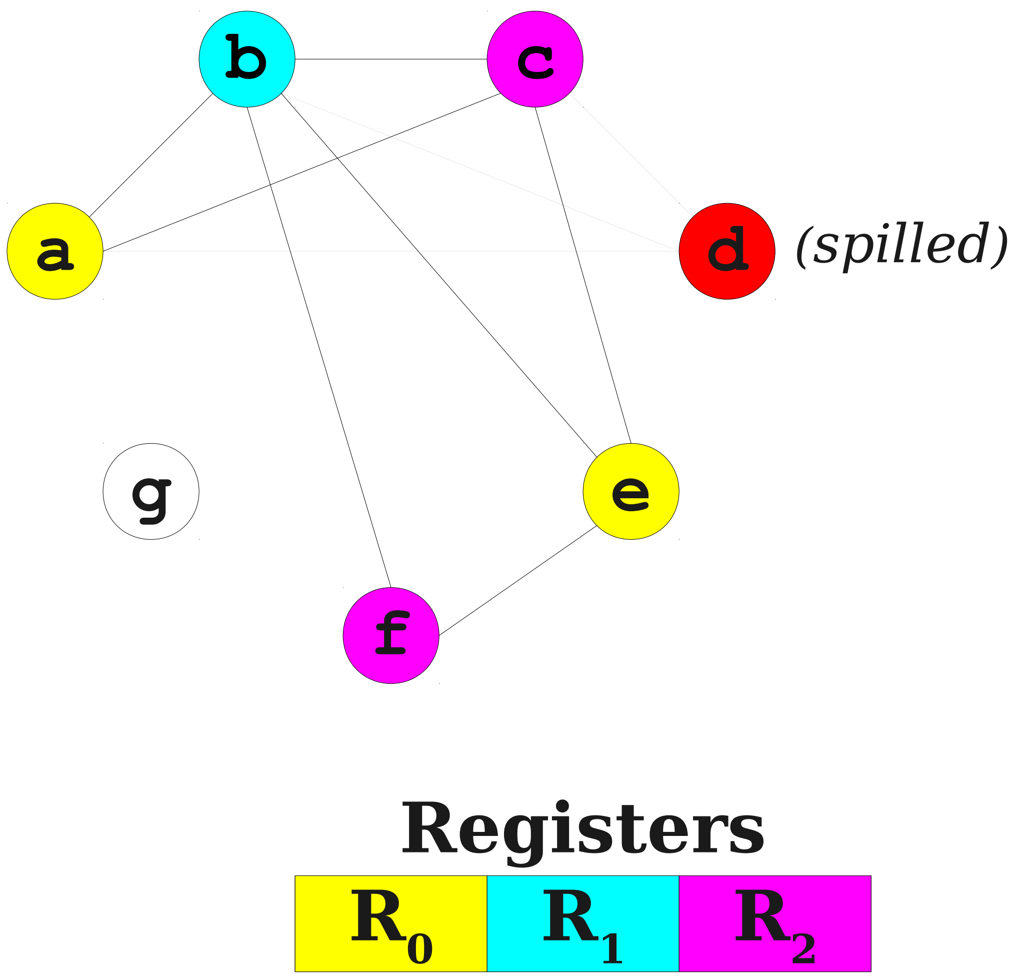
Chaitin's Algorithm Reloaded



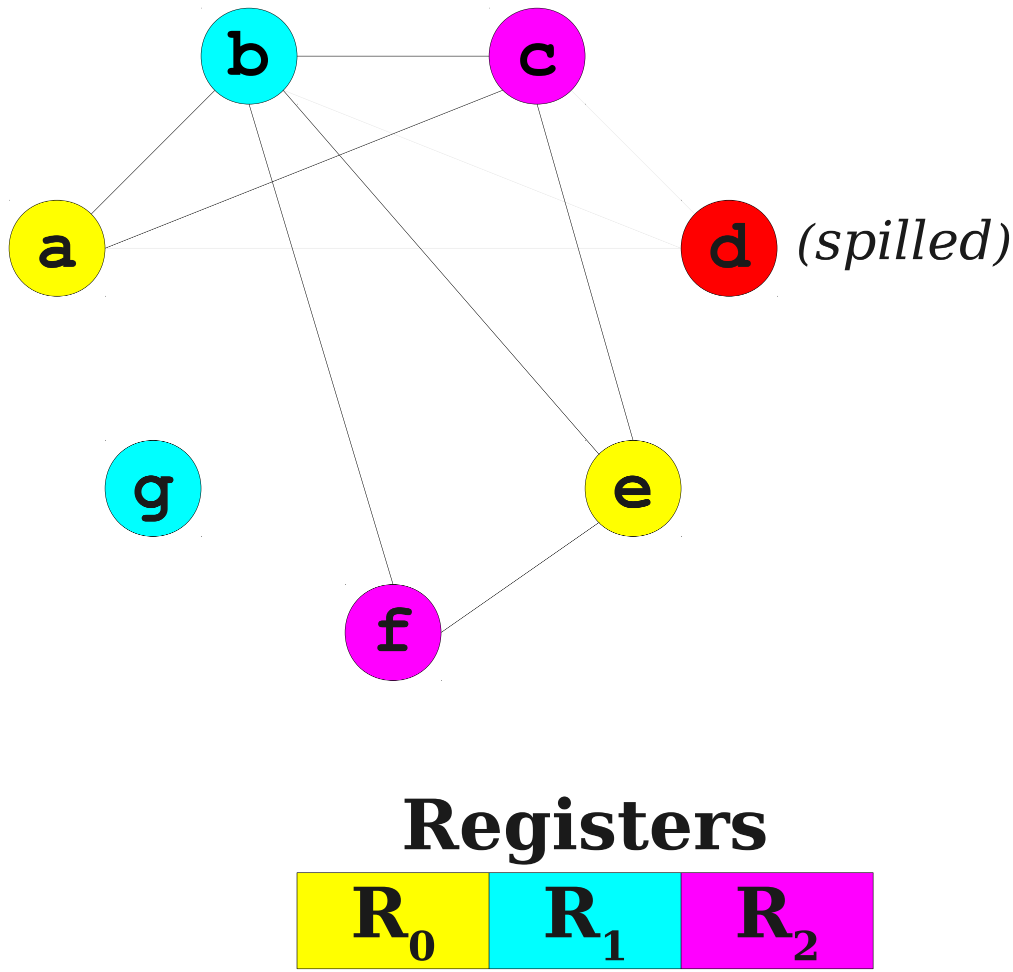
Registers

R_0	R_1	R_2

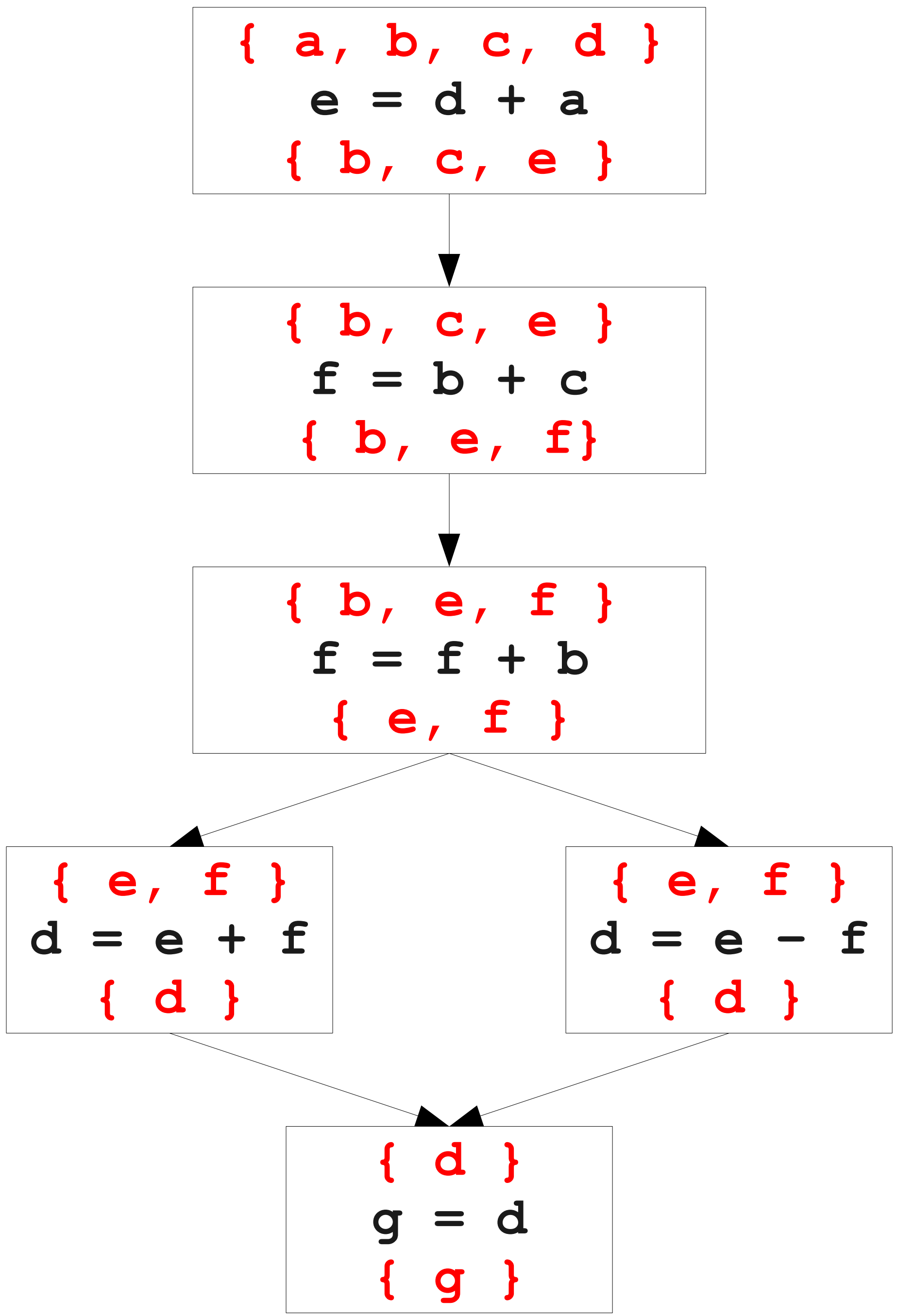
Chaitin's Algorithm Reloaded

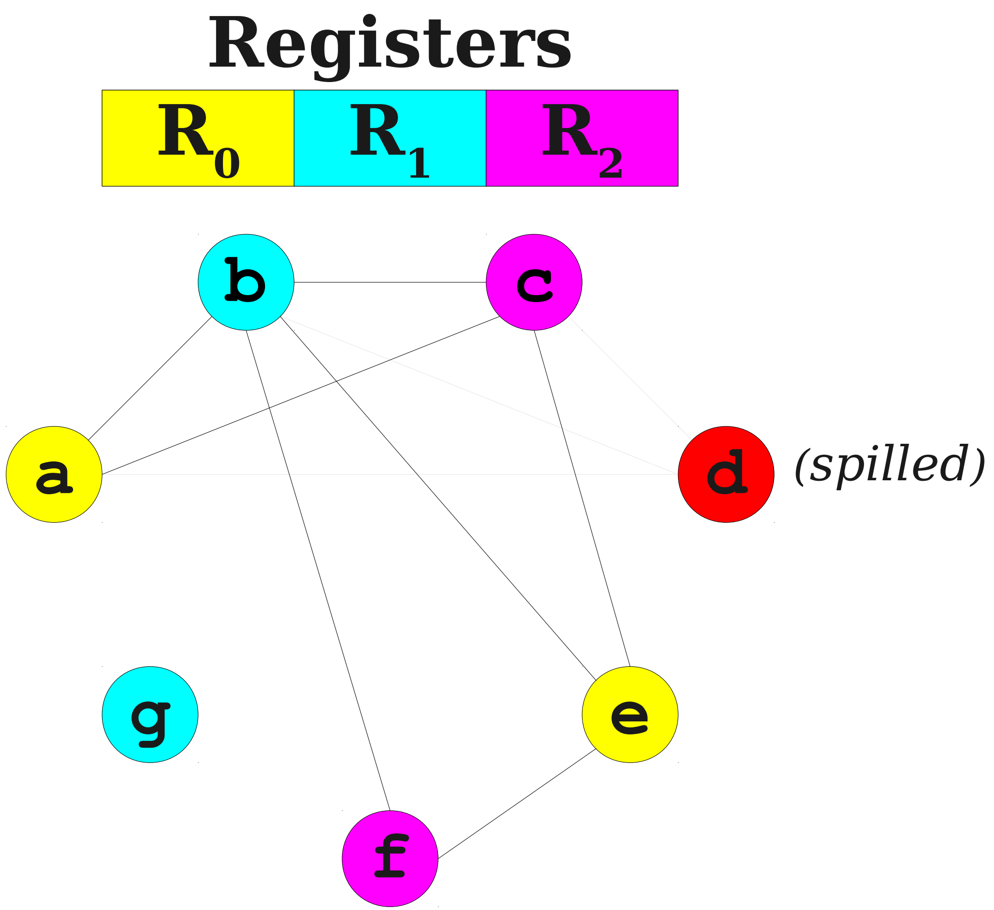


Chaitin's Algorithm Reloaded

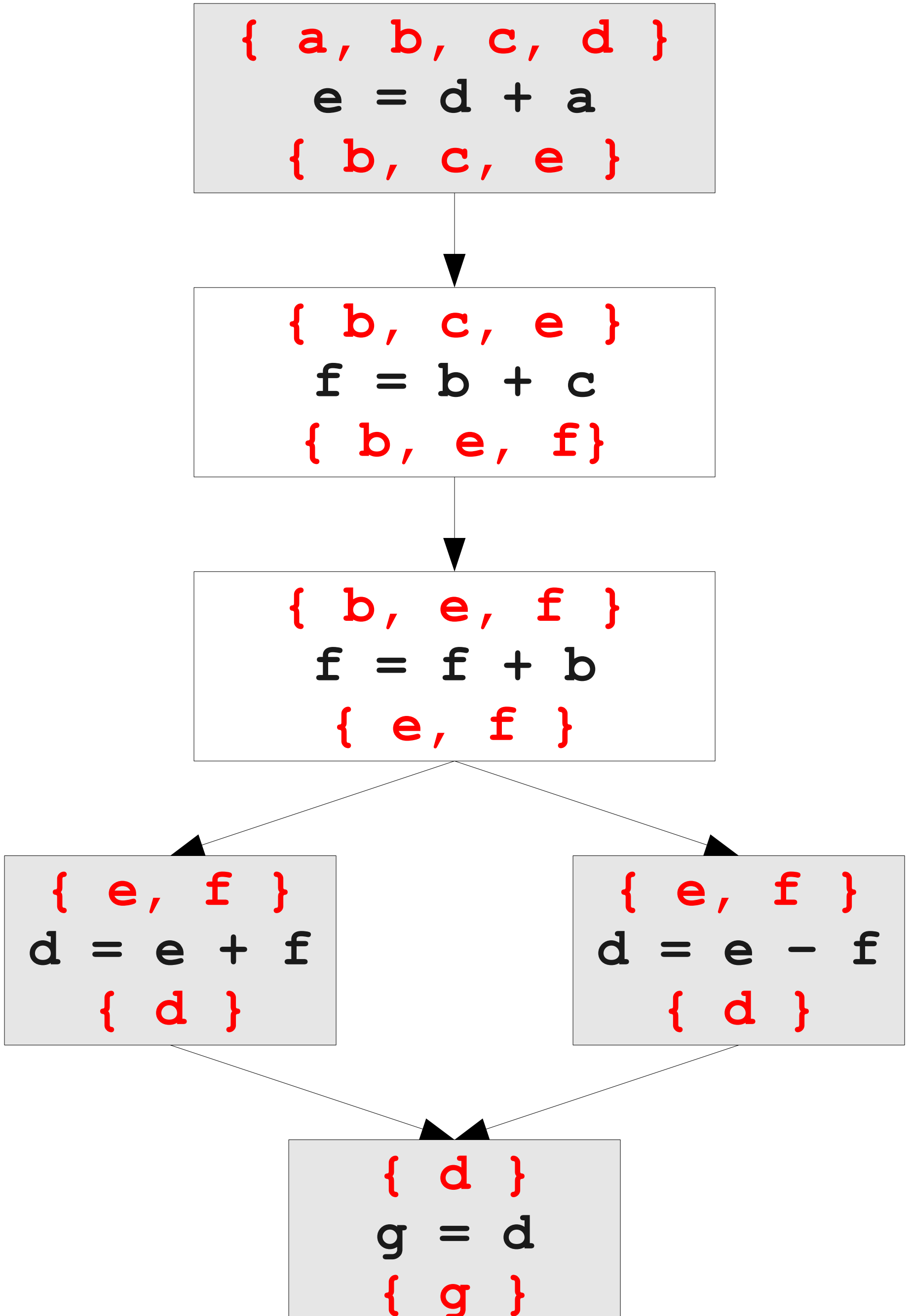


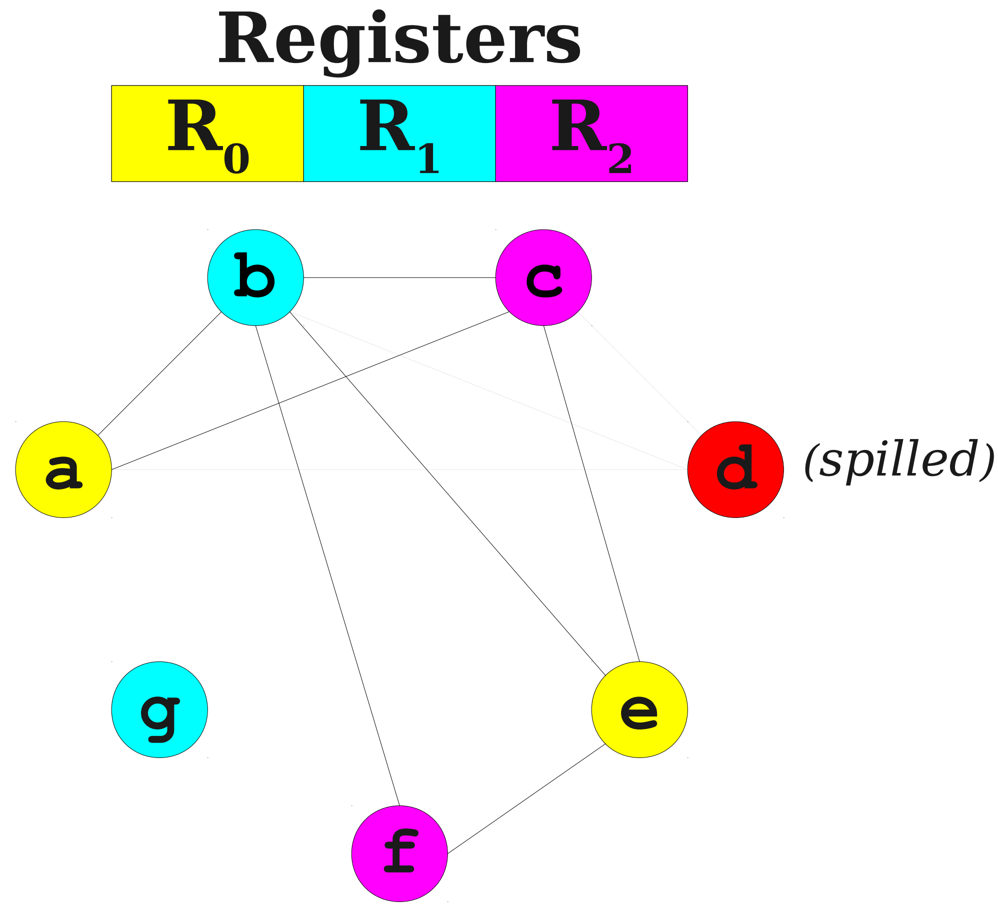
A Smarter Algorithm



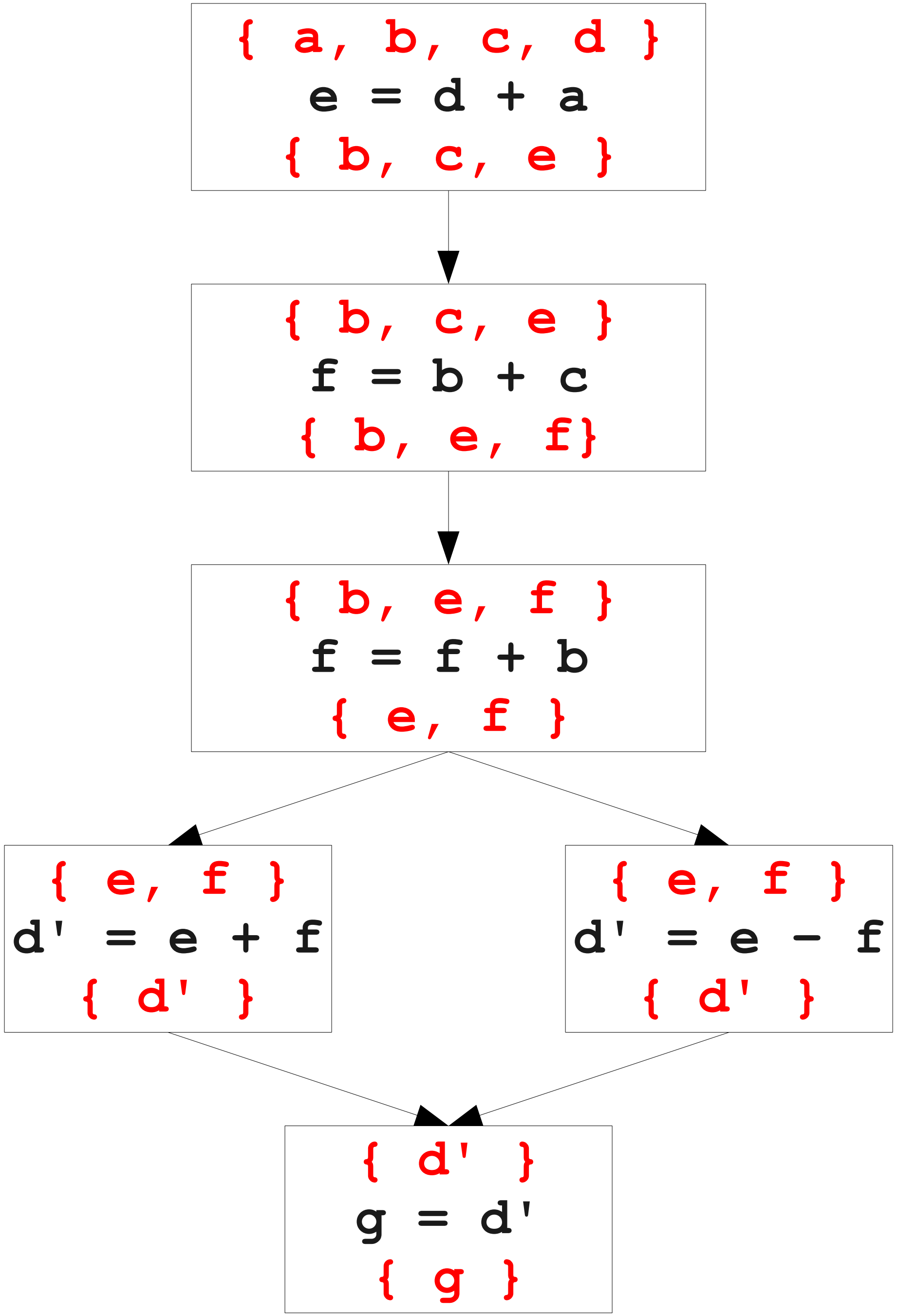


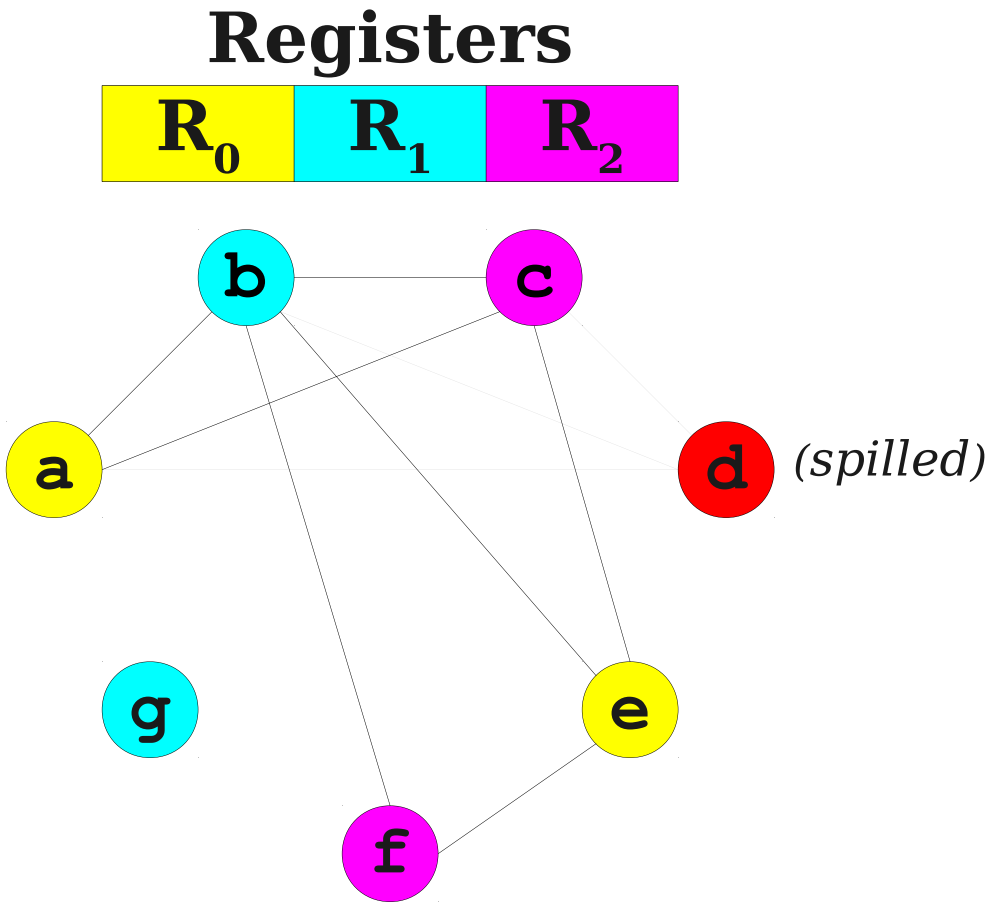
A Smarter Algorithm



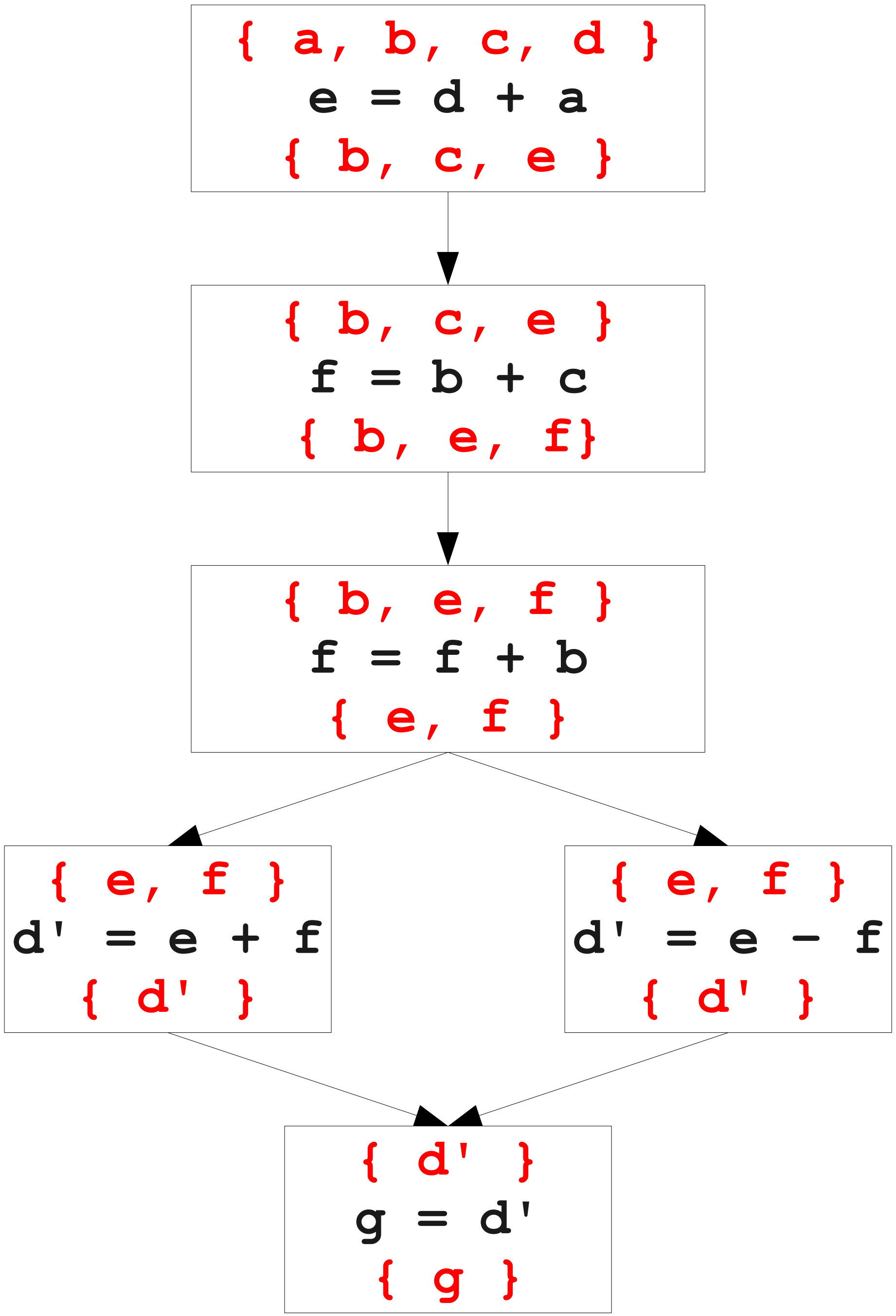


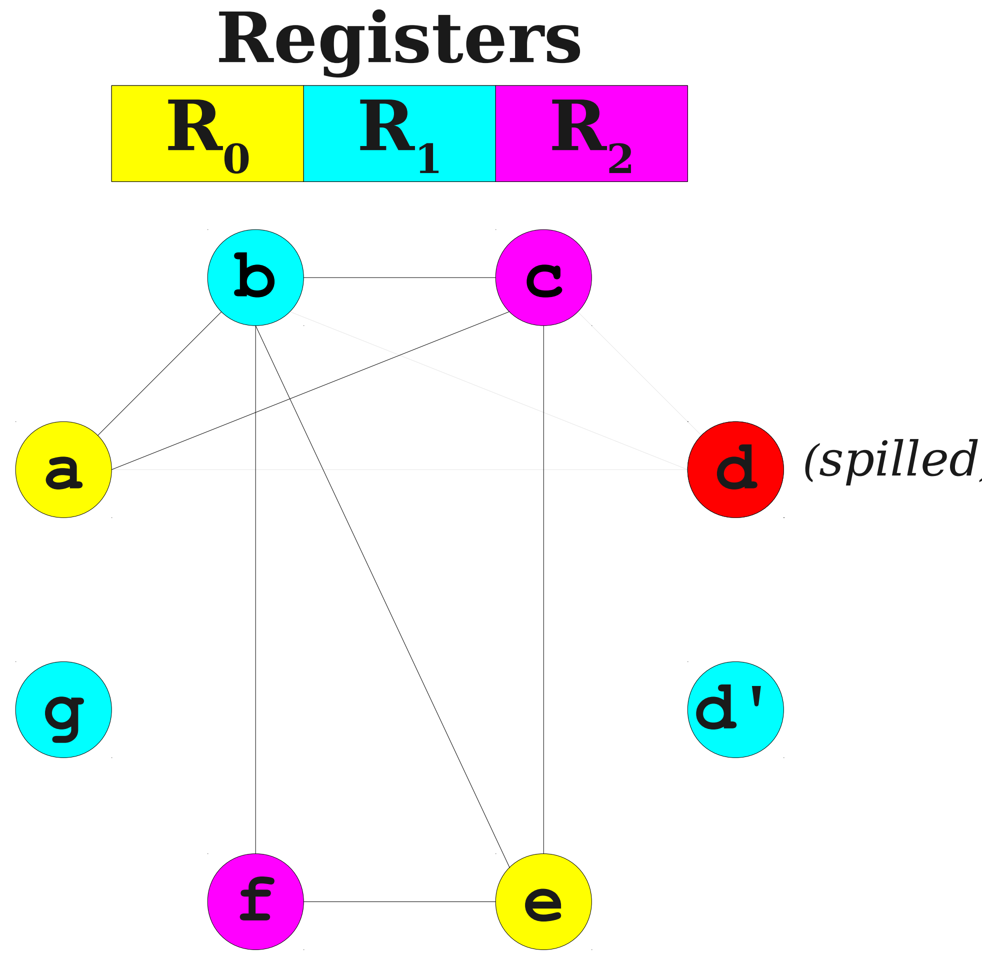
A Smarter Algorithm





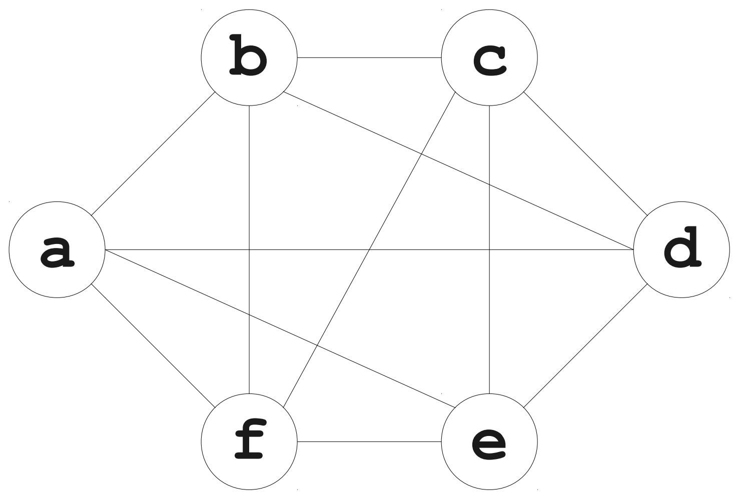
A Smarter Algorithm



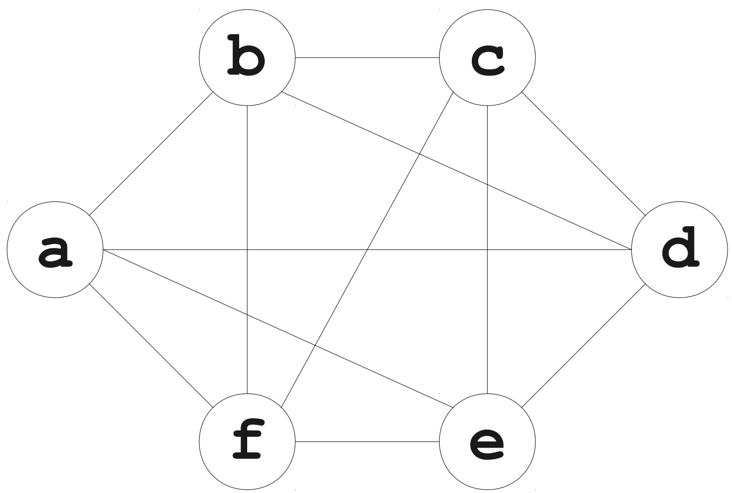


Another Example

Another Example



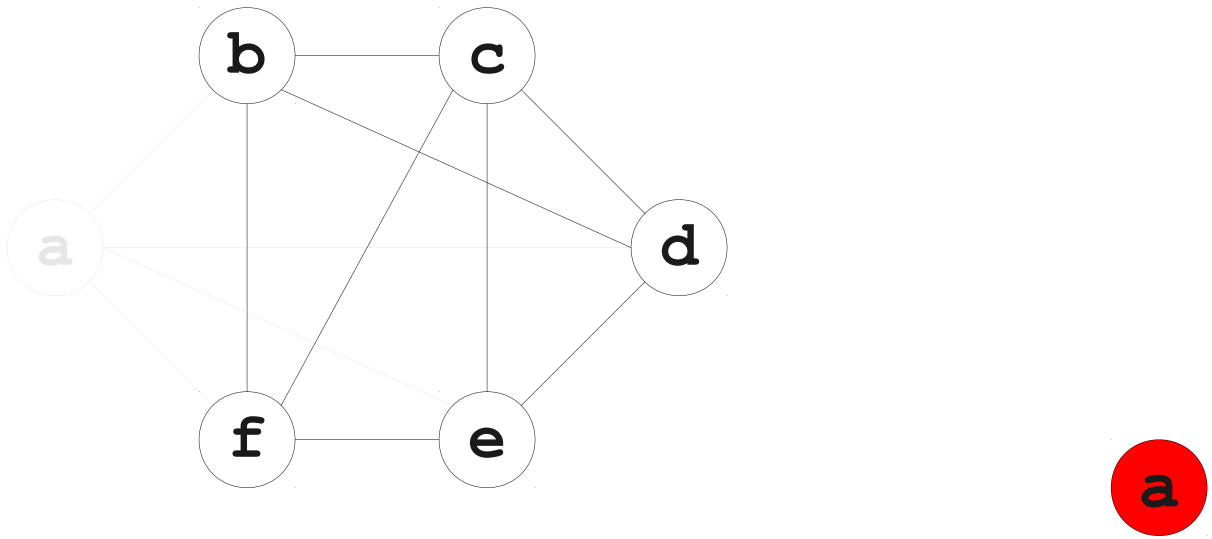
Another Example



Registers

R_0	R_1	R_2

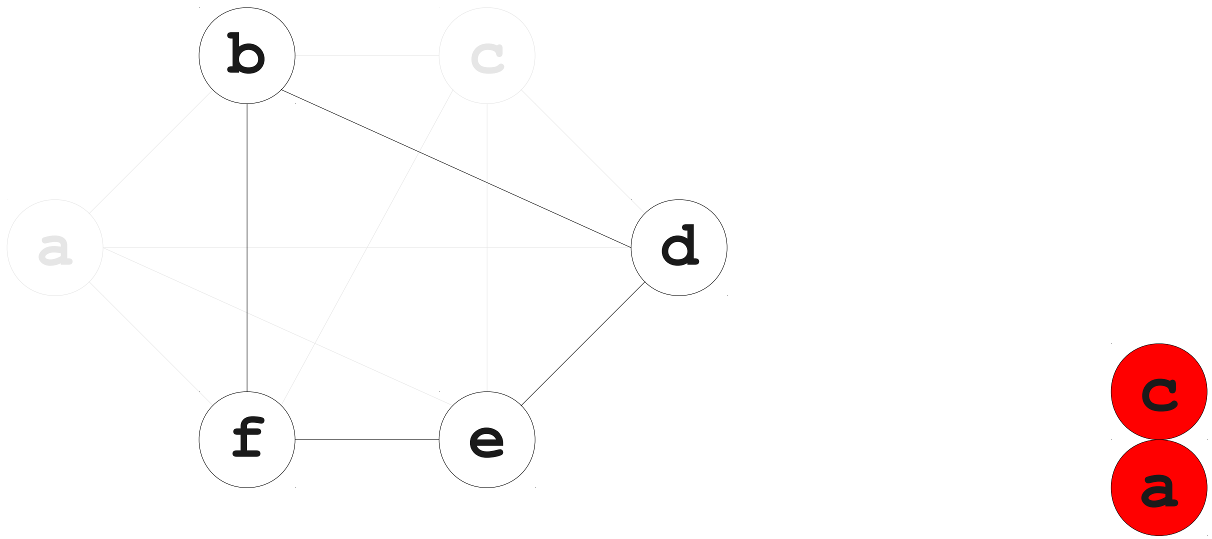
Another Example



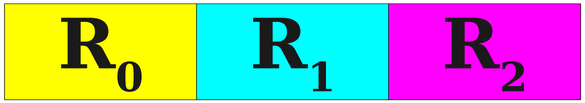
Registers

R_0	R_1	R_2

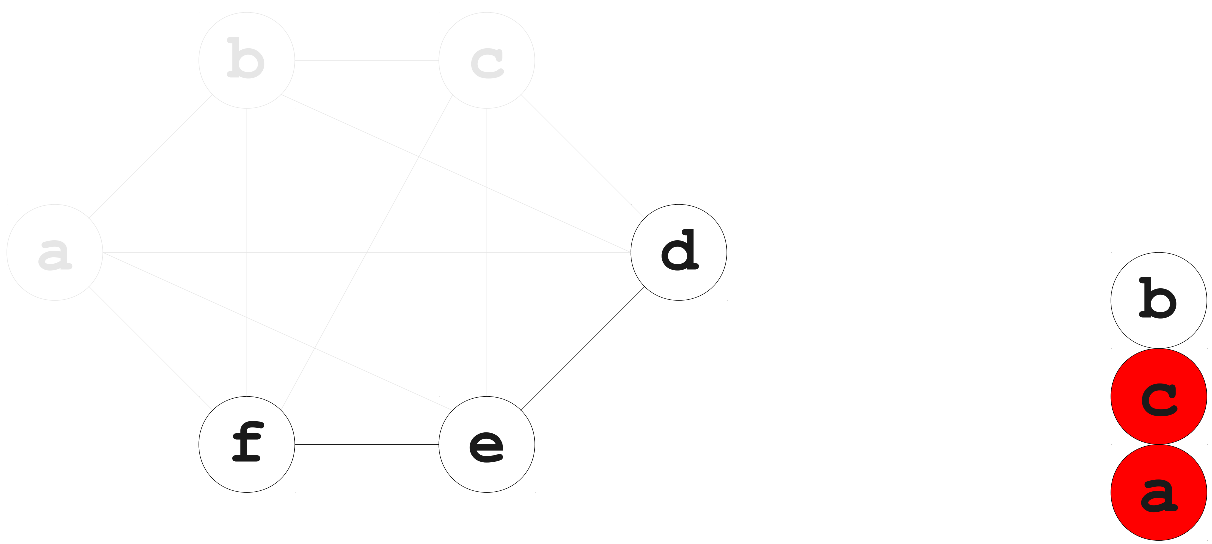
Another Example



Registers



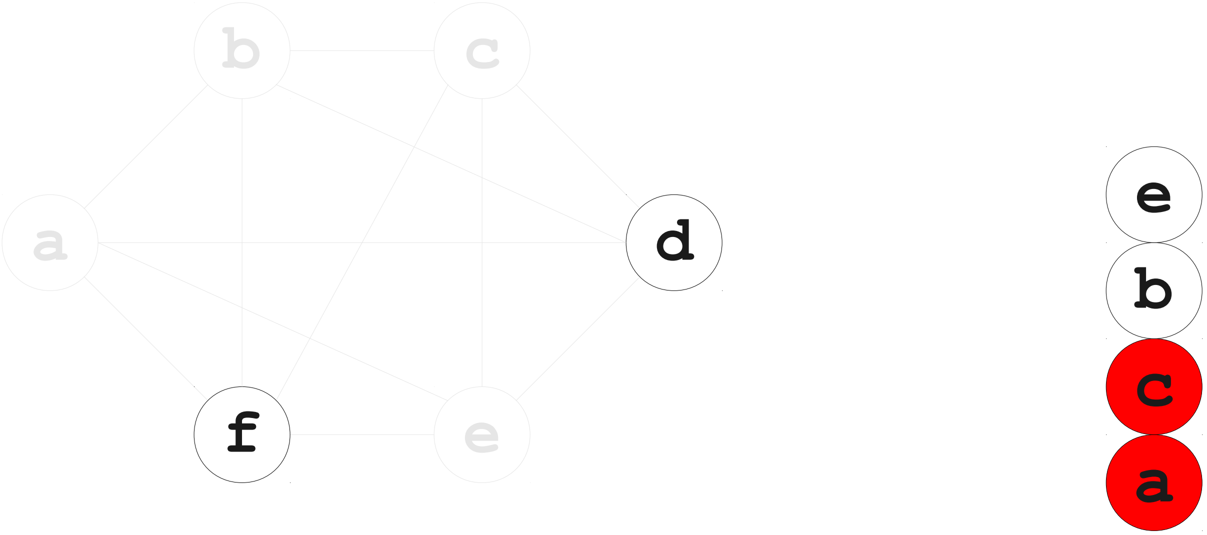
Another Example



Registers

R_0	R_1	R_2

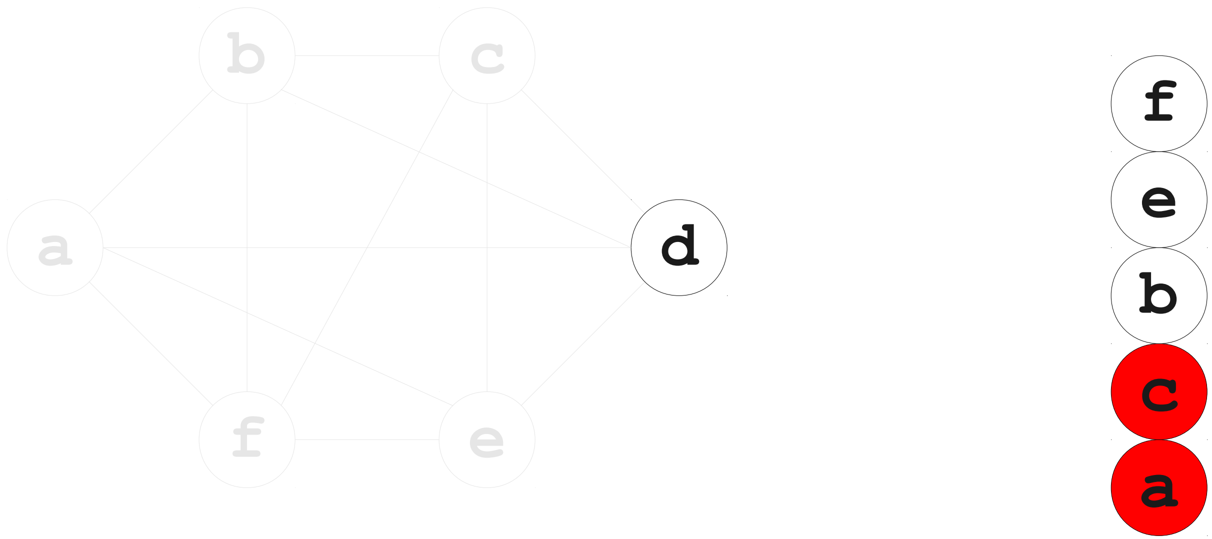
Another Example



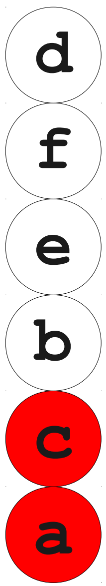
Registers

R_0	R_1	R_2

Another Example



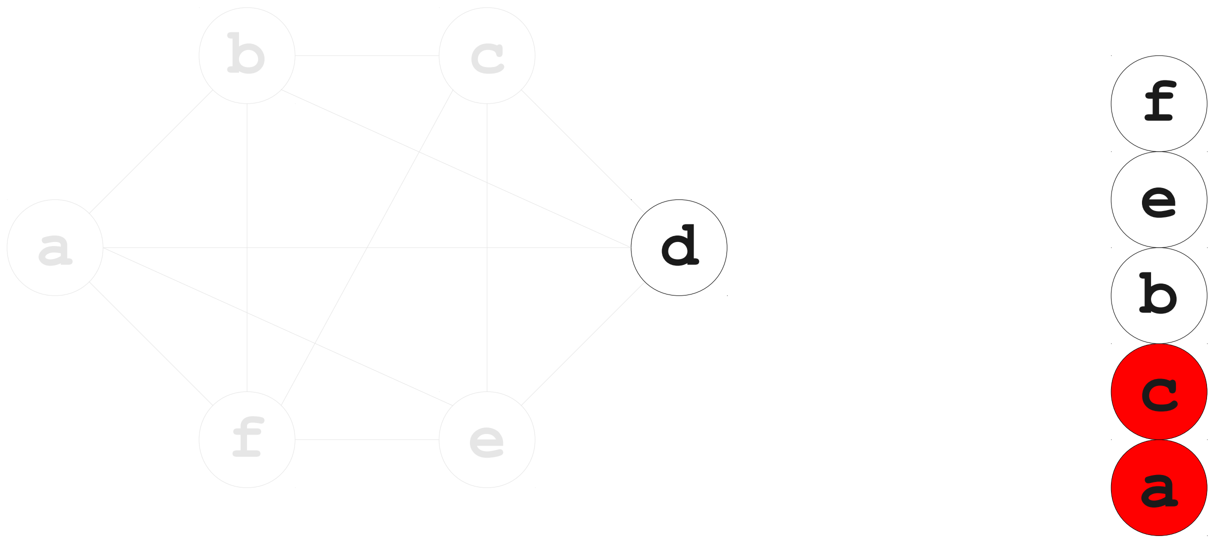
Another Example



Registers

R_0	R_1	R_2

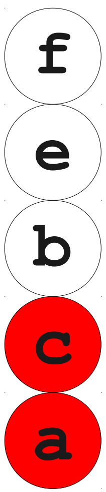
Another Example



Registers

R_0	R_1	R_2

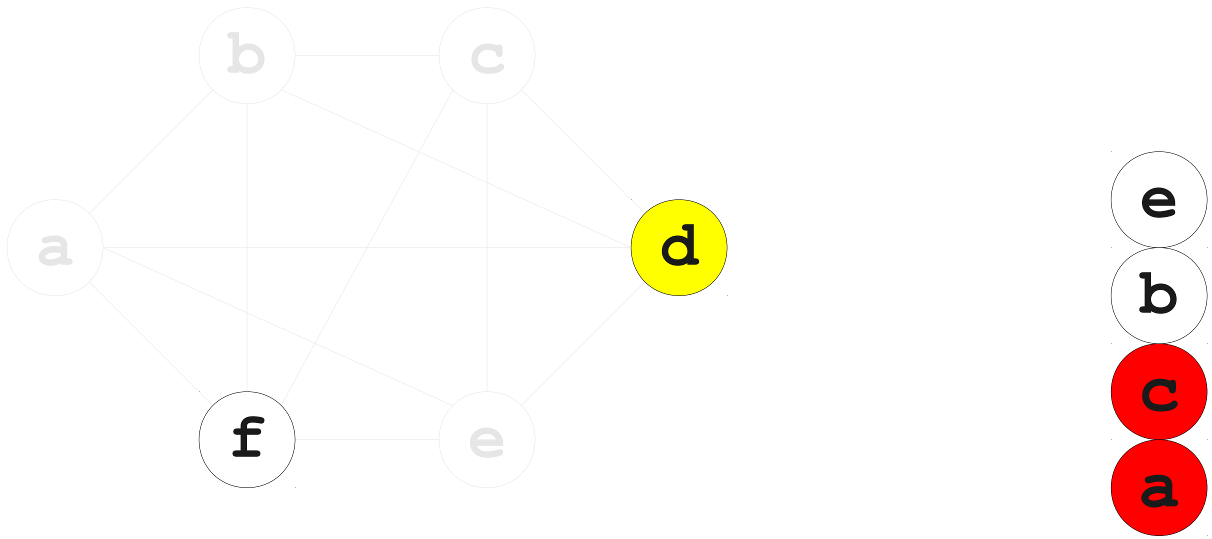
Another Example



Registers

R_0	R_1	R_2

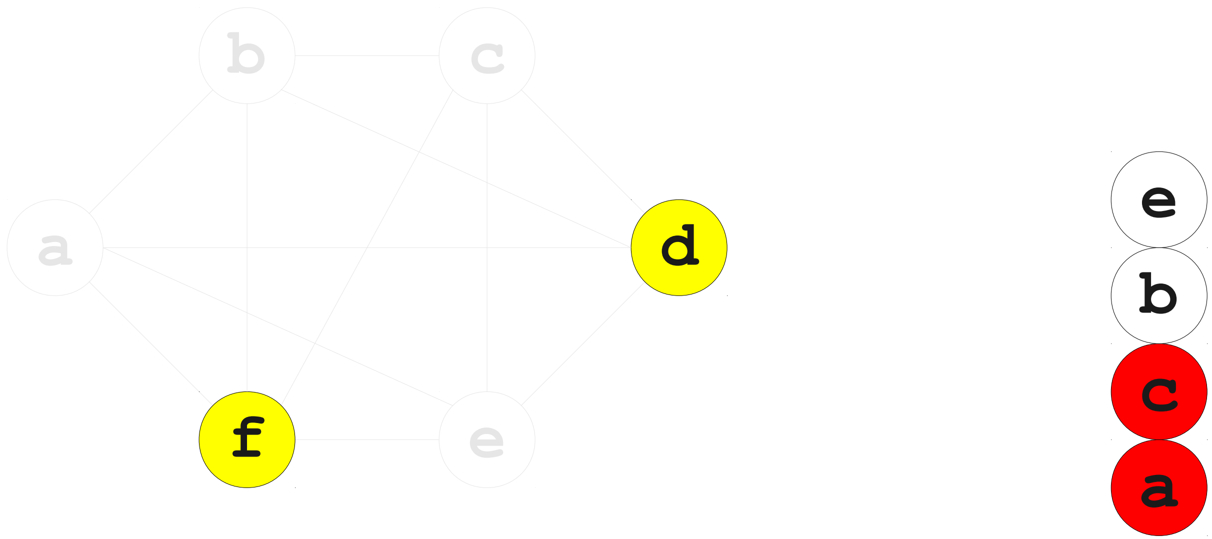
Another Example



Registers

R_0	R_1	R_2

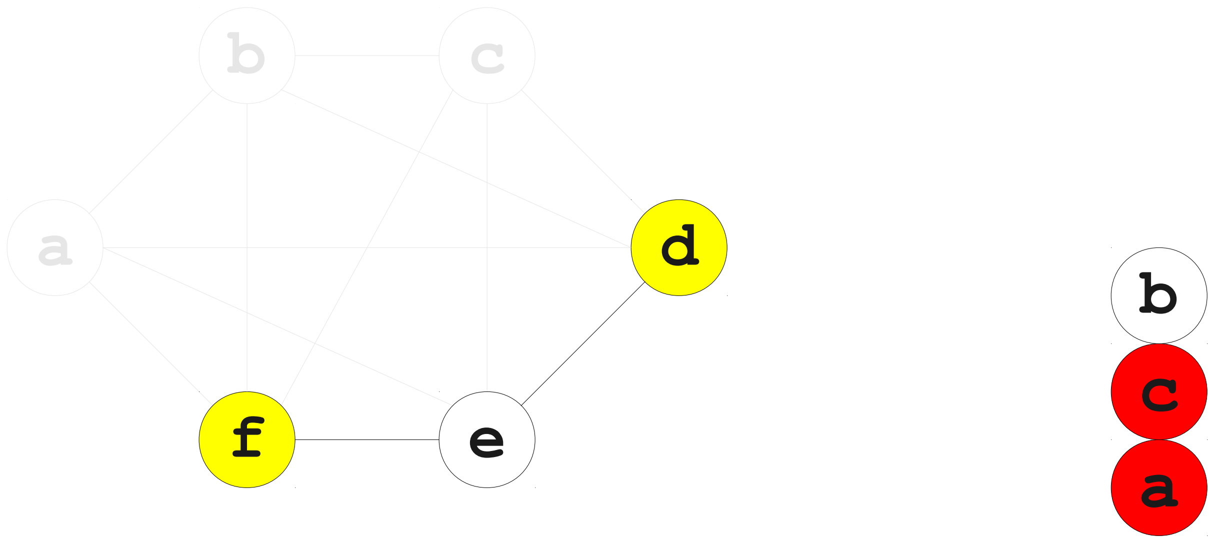
Another Example



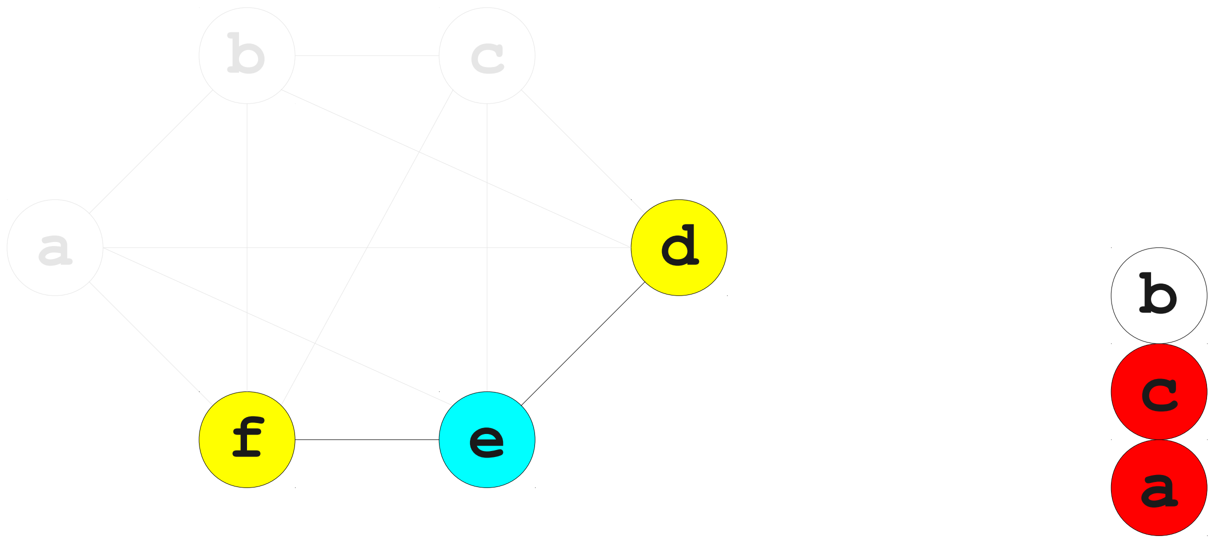
Registers

R_0	R_1	R_2

Another Example



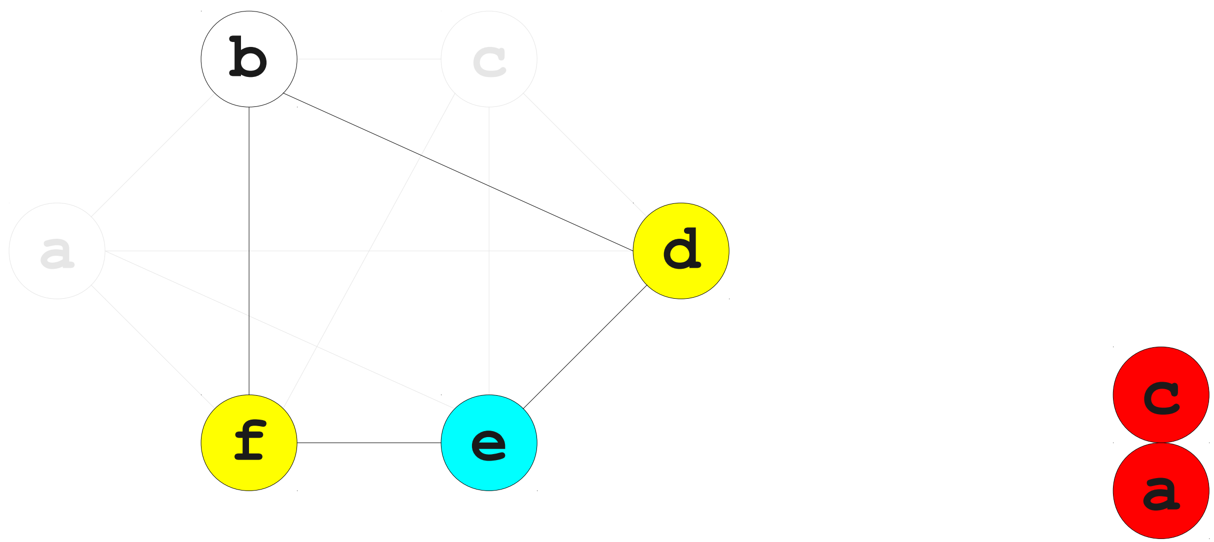
Another Example



Registers

R_0	R_1	R_2

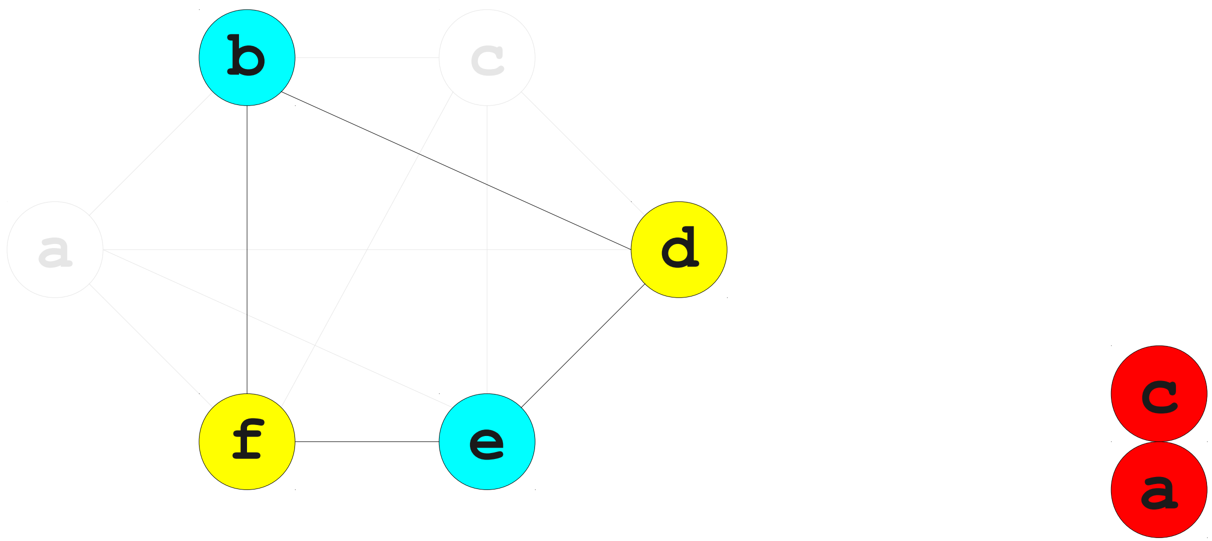
Another Example



Registers

R_0	R_1	R_2

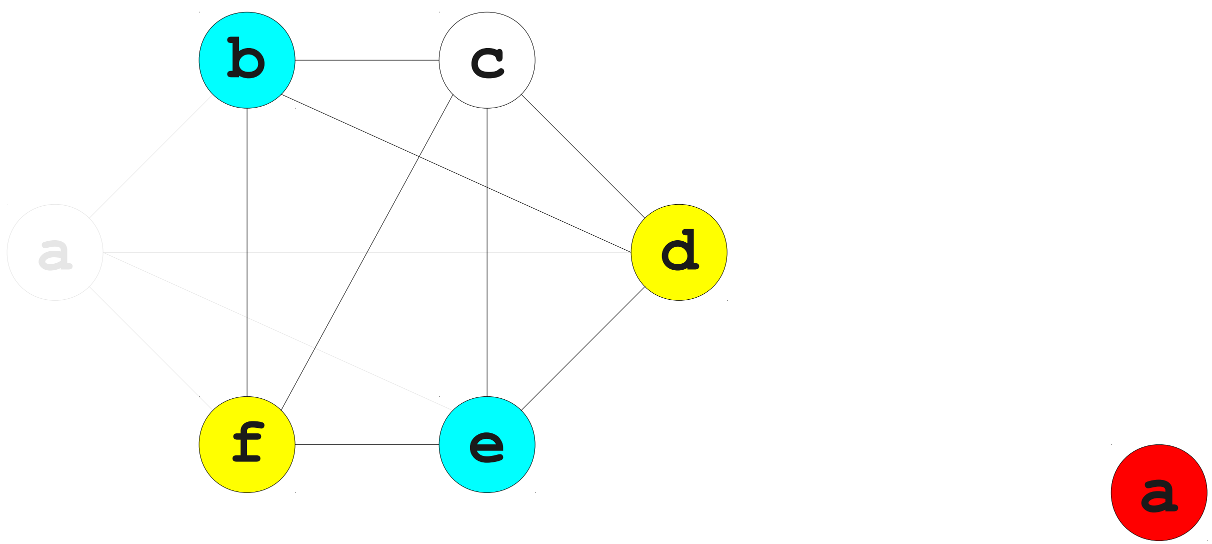
Another Example



Registers

R_0	R_1	R_2

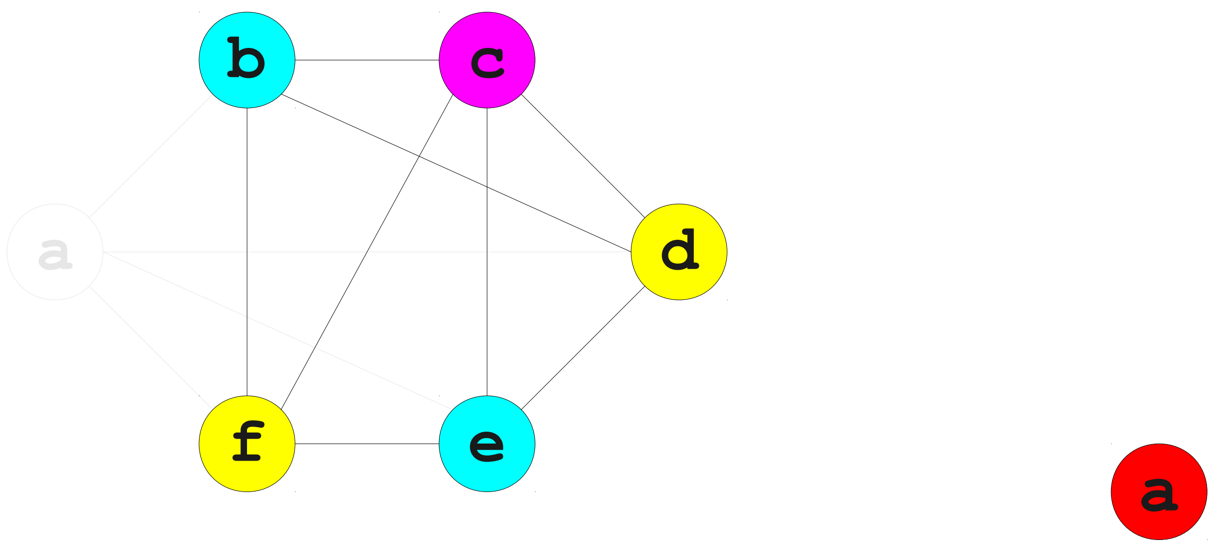
Another Example



Registers

R_0	R_1	R_2

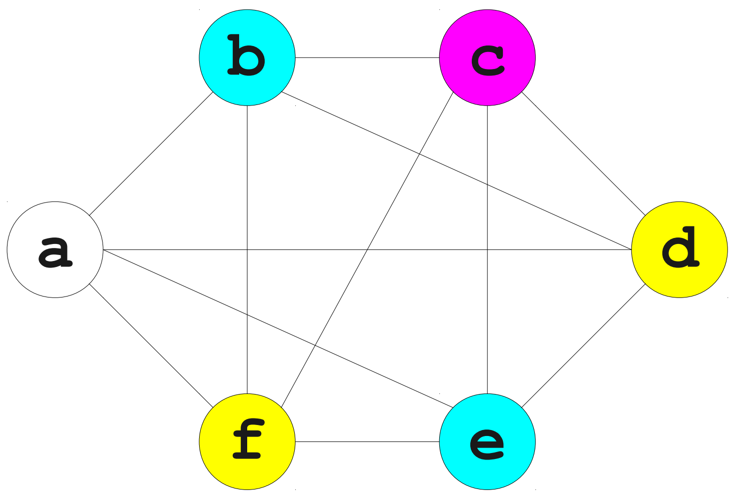
Another Example



Registers

R_0	R_1	R_2

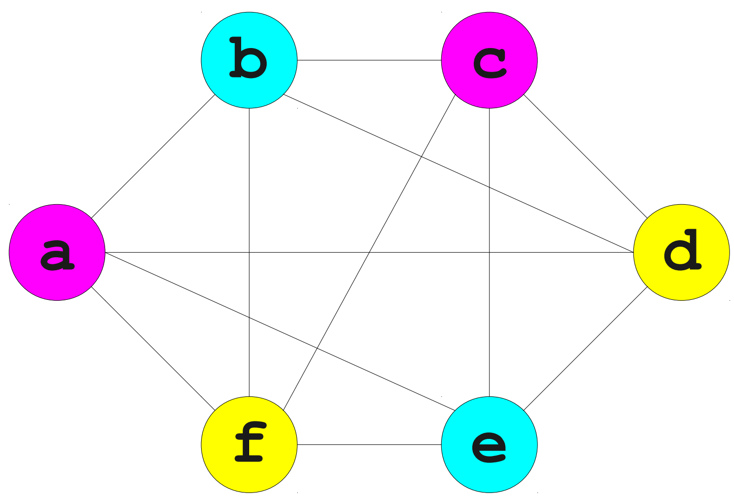
Another Example



Registers

R_0	R_1	R_2

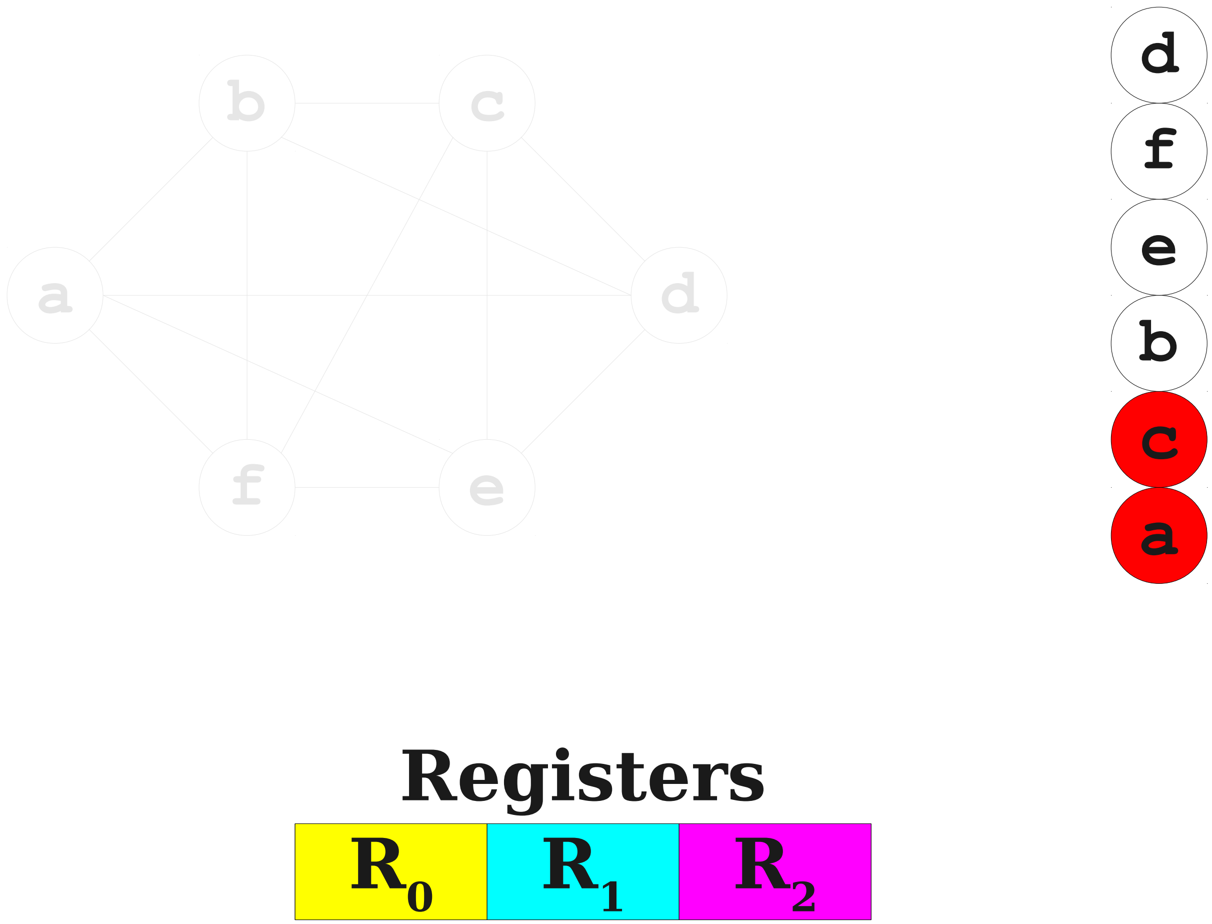
Another Example



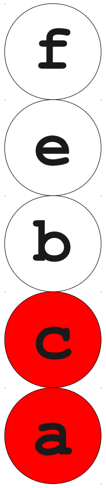
Registers

R_0	R_1	R_2

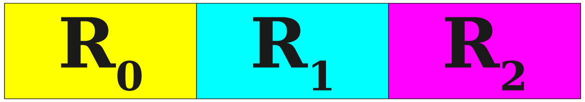
Another Example



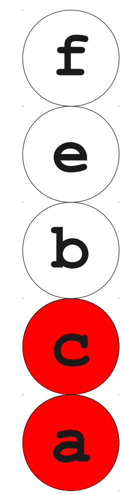
Another Example



Registers



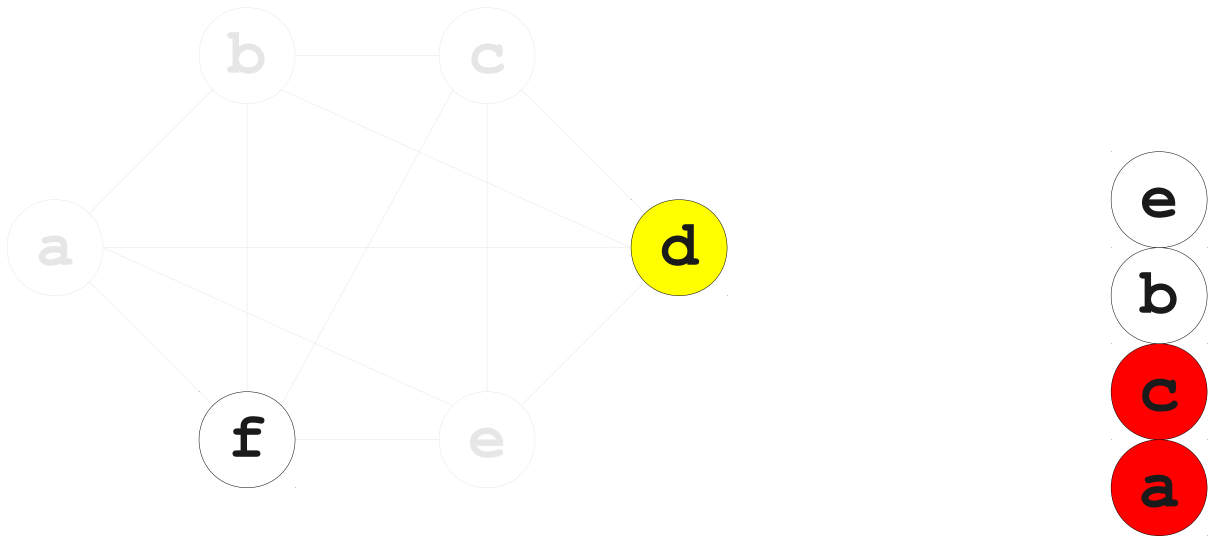
Another Example



Registers

R_0	R_1	R_2

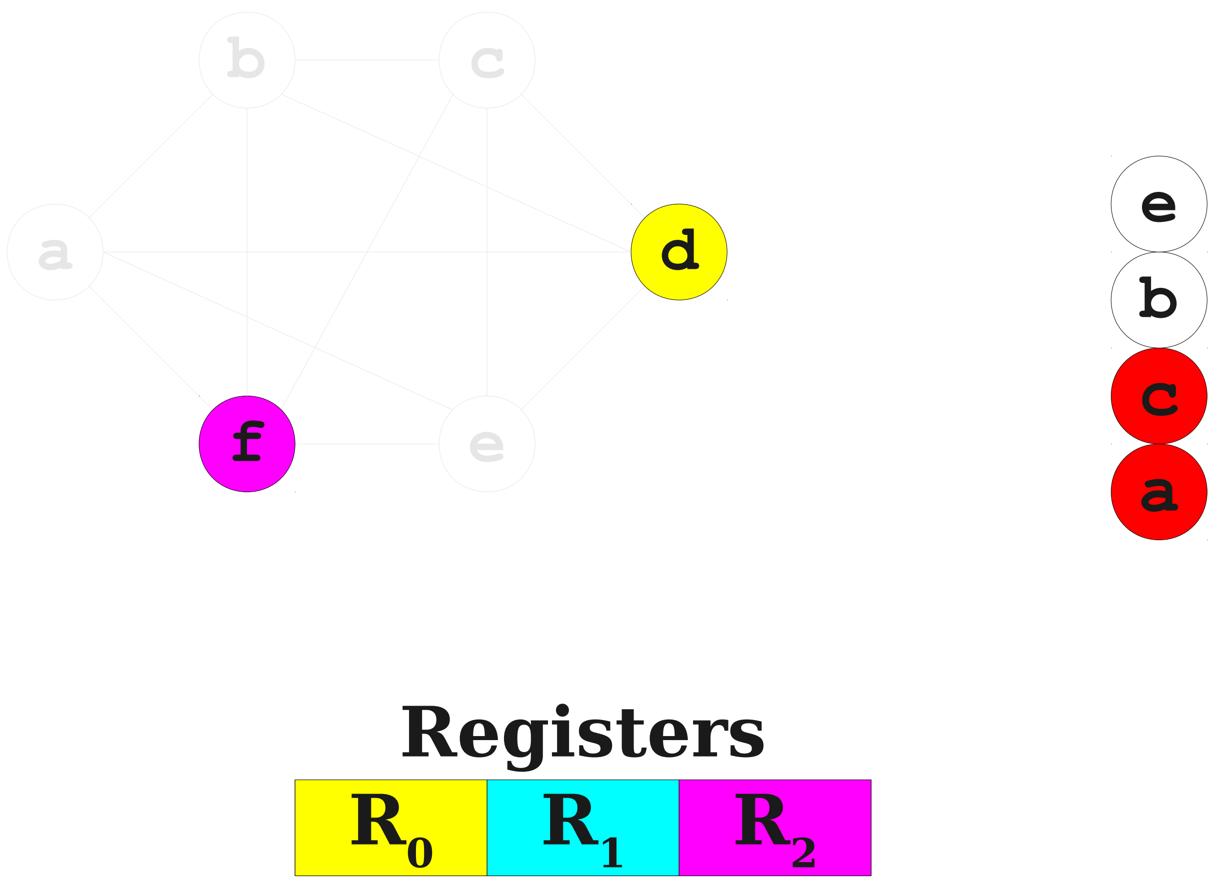
Another Example



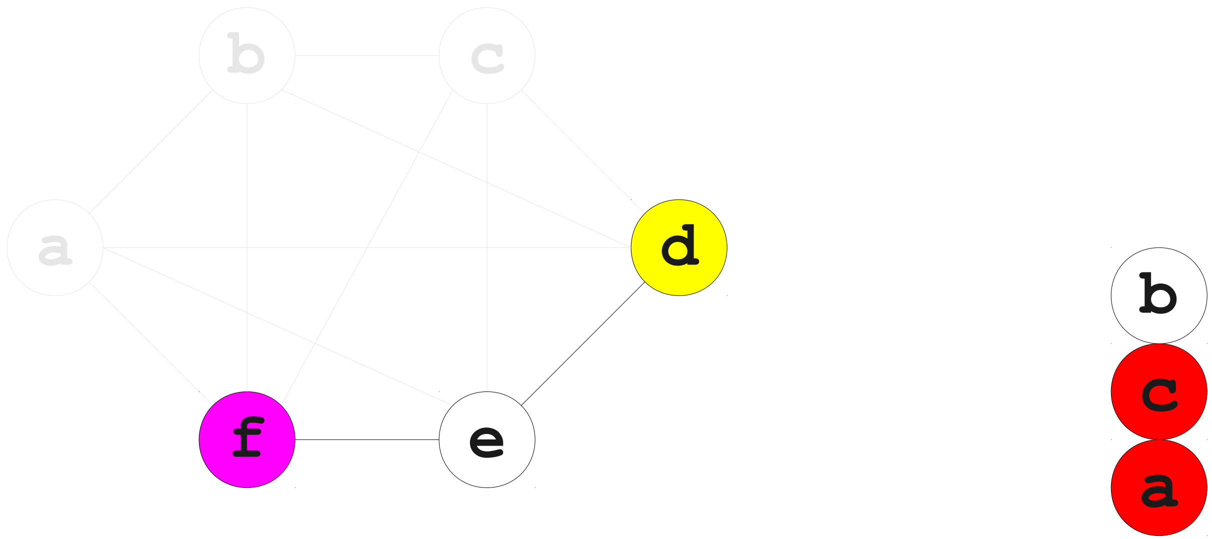
Registers

R_0	R_1	R_2

Another Example



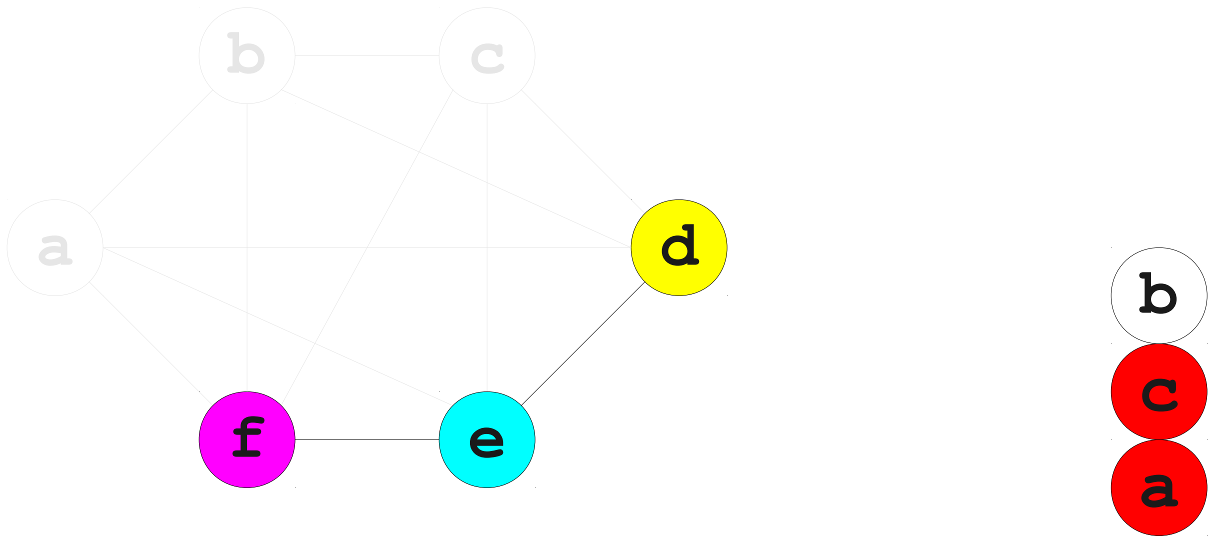
Another Example



Registers

R_0	R_1	R_2

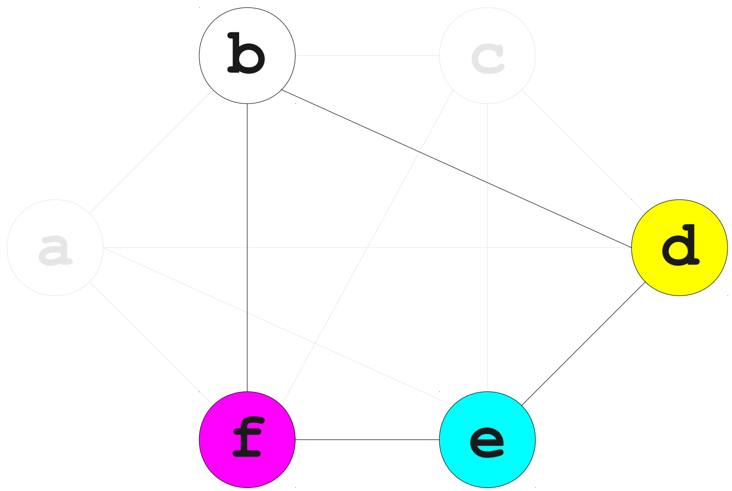
Another Example

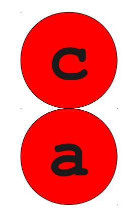


Registers

R_0	R_1	R_2

Another Example

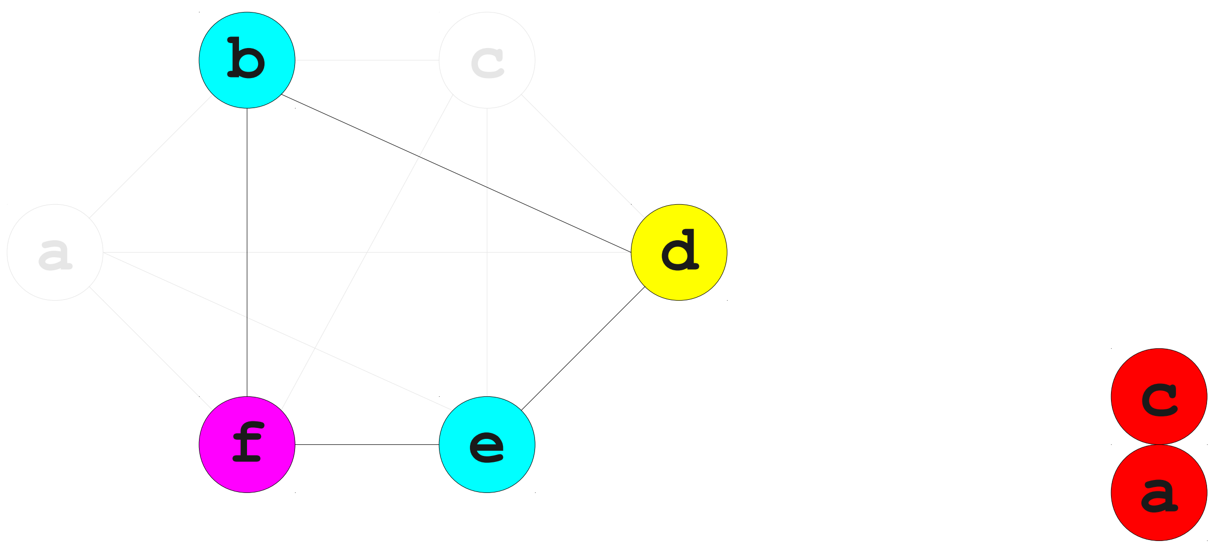




Registers

R_0	R_1	R_2

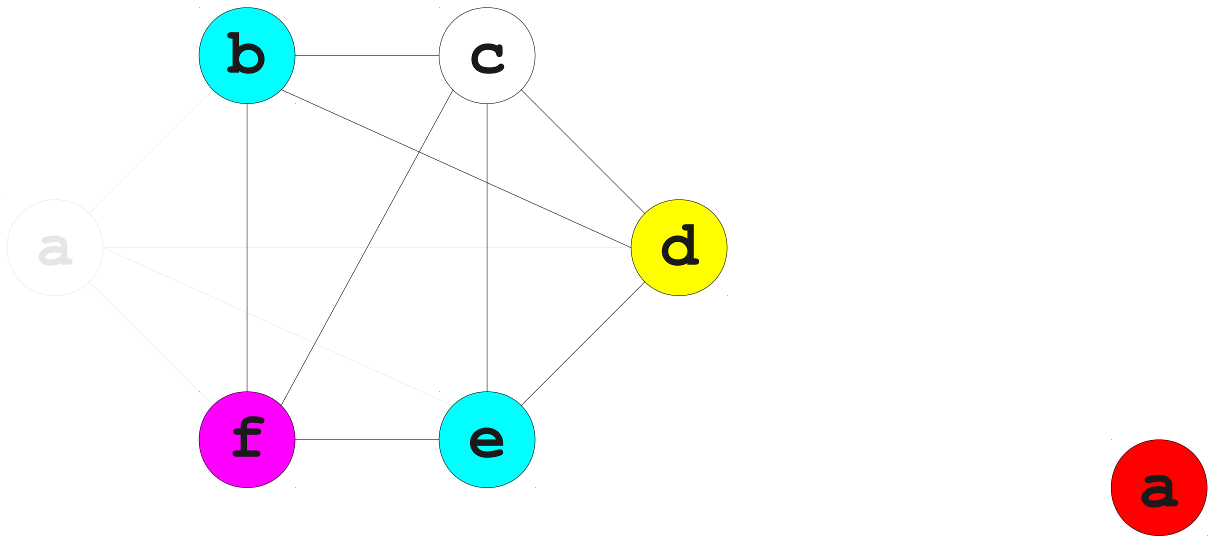
Another Example



Registers

R_0	R_1	R_2

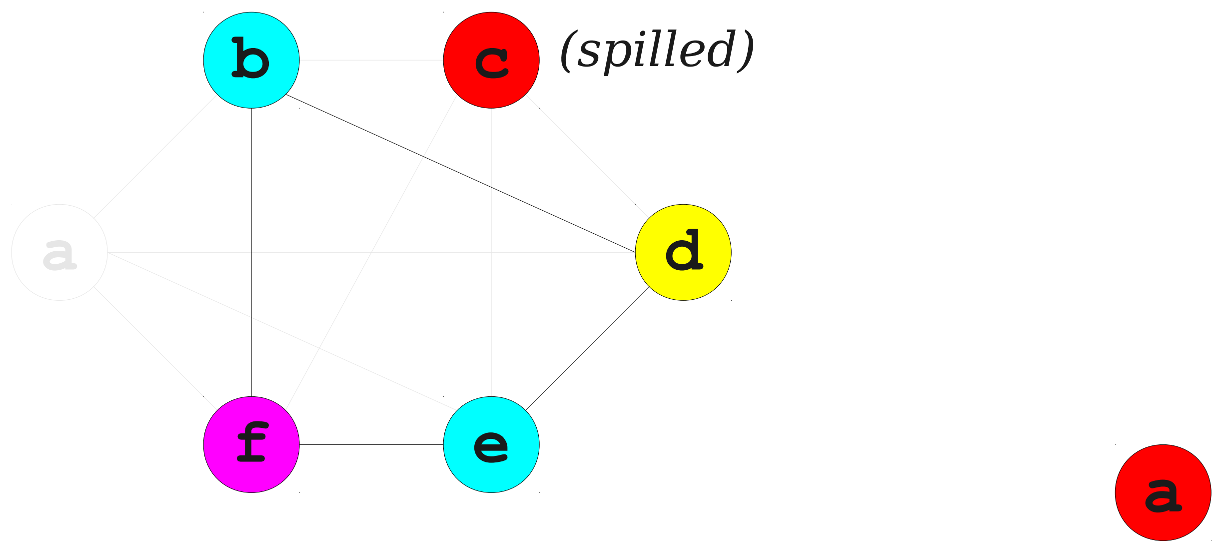
Another Example



Registers

R_0	R_1	R_2

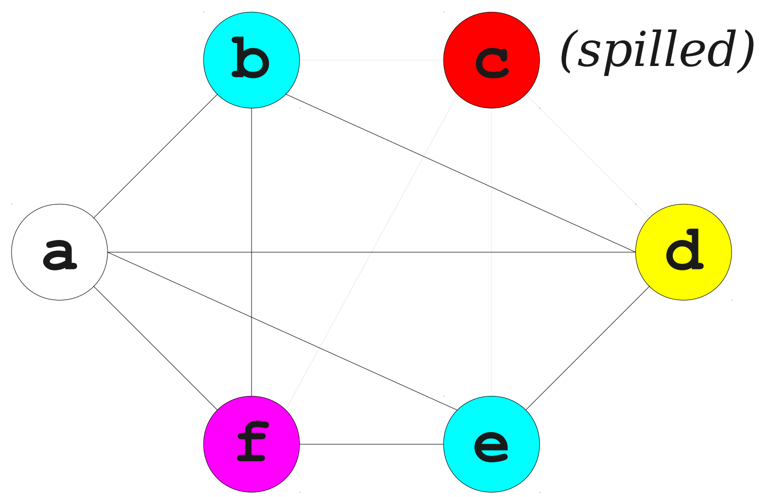
Another Example



Registers

R_0	R_1	R_2

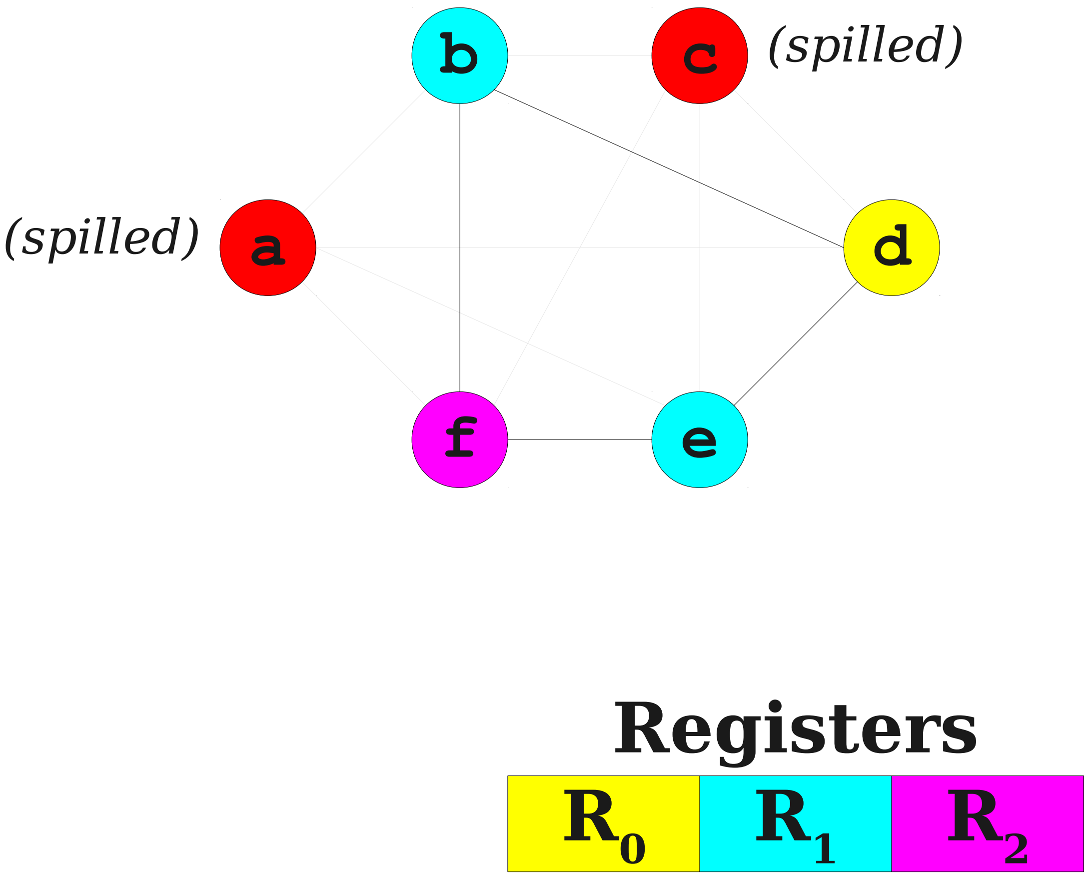
Another Example



Registers

R_0	R_1	R_2

Another Example



Chaitin's Algorithm

- Advantages:
  - For many control-flow graphs, finds an excellent assignment of variables to registers.
  - When distinguishing variables by use, produces a precise RIG.
  - Often used in production compilers like GCC.
- Disadvantages:
  - Core approach based on the NP-hard graph coloring problem.
  - Heuristic may produce pathologically worst-case assignments.

Correctness Proof Sketch

- No two variables live at some point are assigned the same register.
  - Forced by graph coloring.
- At any program point each variable is always in one location.
  - Automatic if we assign each variable one register.
  - Requires a few tricks if we separate by use case.

Improvements to the Algorithm

- Choose what to spill intelligently.
  - Use heuristics (least-commonly used, greatest improvement, etc.) to determine what to spill.
- Handle spilling intelligently.
  - When spilling a variable, recompute the RIG based on the spill and use a new coloring to find a register.

Summary of Register Allocation

- Critical step in all optimizing compilers.
- The **linear scan** algorithm uses **live intervals** to greedily assign variables to registers.
  - Often used in JIT compilers due to efficiency.
- **Chaitin's algorithm** uses the **register interference graph** (based on **live ranges**) and **graph coloring** to assign registers.
  - The basis for the technique used in GCC.