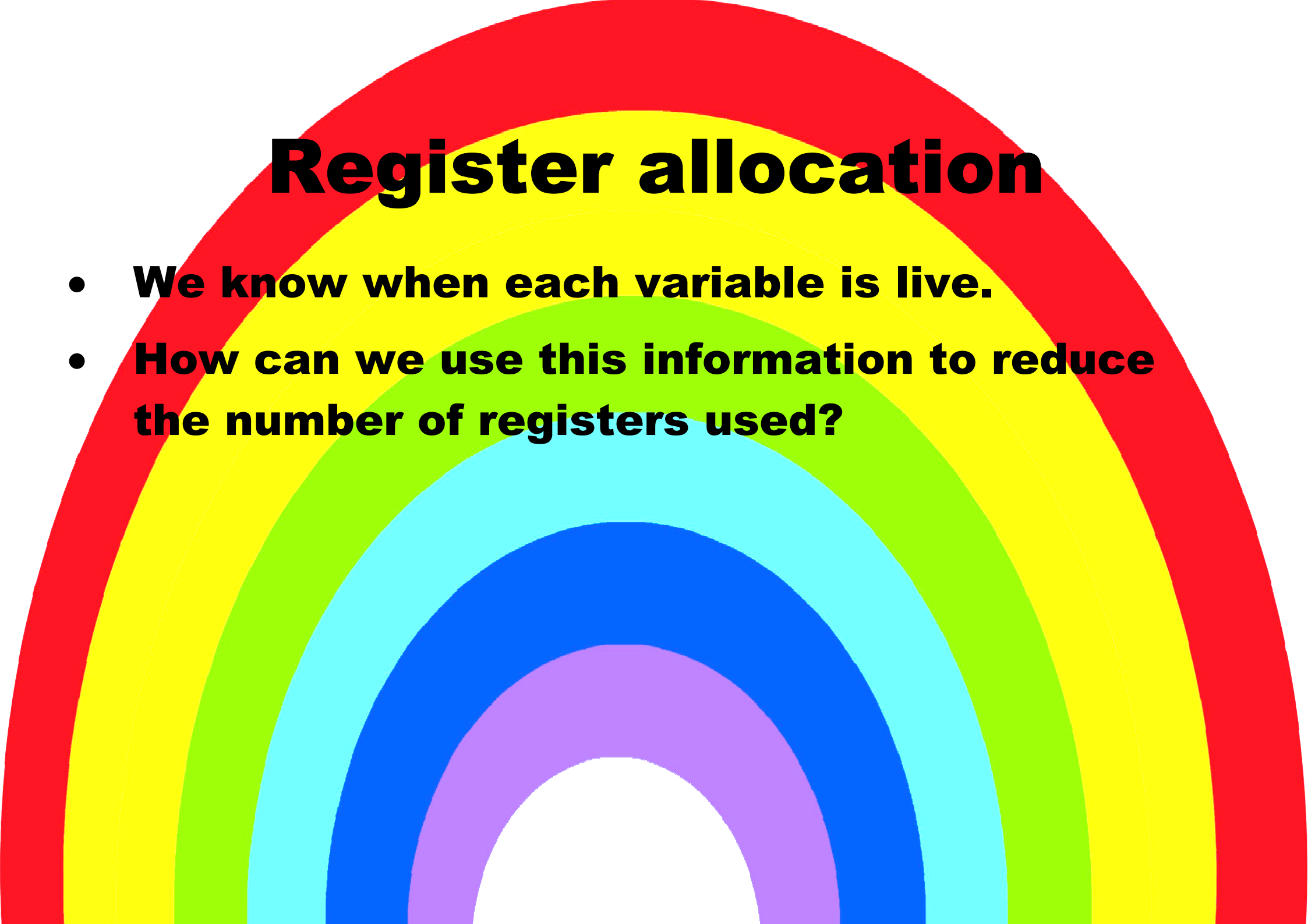
Register allocation

- We know when each variable is live.
- How can we use this information to reduce the number of registers used?



Register allocation

Real processors have limited number of registers.

Register allocation:

Allocating temporaries to machine registers so that:

- they fit in limited number (k) of registers

- moves between registers are avoided

Allocating temporaries to registers

Basic idea:

Can use same register for several temporaries if they are not live at same time

Algorithm:

1. Set up a *register interference graph*: undirected graph in which
   - nodes represent temporary variables
   - edges join temporaries that are live simultaneously.
2. *Colour* graph:
   - label nodes with numbers such that neighbouring nodes have different numbers.

3. If graph can be coloured using at most *k* colours
   - use a different register for each colour

   Else, if more than *k* colours are needed:

   - rewrite program to use fewer temporaries
   - *spill* values to memory
   - try colouring again

(Register) interference graph

Interference edges join temporaries that cannot be allocated to same register. Because:

1. they are live simultaneously

2. one is assigned to (but not live) while the other is live

3. machine specific: some registers not available for instruction

Constructing a (register) interference graph

- Nonmove instruction `x = y op z`
- Move instruction `x = y`

Algorithm:

For each instruction *s*:

- If *s* is nonmove instruction `x = y op z`:

  For each variable `v` that is in *out(s)*, add edge `(x,v)`

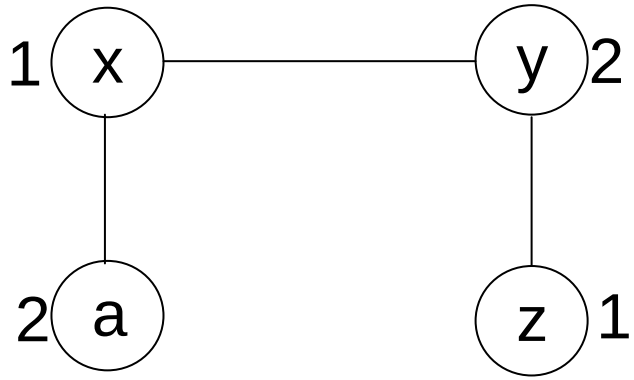
- If *s* is move instruction `x = y`:

  For each variable `v` that is in *out(s)*, **except `y`**, add edge `(x,v)`

Example 1:

	live	*edge*
`x = 2`	{x}	
`a = 1`	{x, a}	ax
`y = x + a`	{x, y}	xy
`z = y + x`	{y, z}	yz
`z = y + z`	{z}	
`a = z`	{a}	
`x = a + a`	{x}	

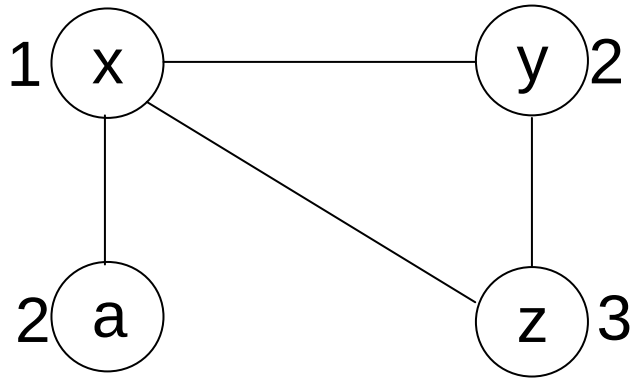
Register interference graph:



Example 2:

	live	*edge*
`x = 2`	{x}	
`z = 3`	{x}	xz
`a = 1`	{x, a}	ax
`y = x + a`	{x, y}	xy
`z = y + x`	{y, z}	yz
`z = y + z`	{z}	
`a = z`	{a}	
`x = a + a`	{x}	

Register interference graph:



Graph colouring

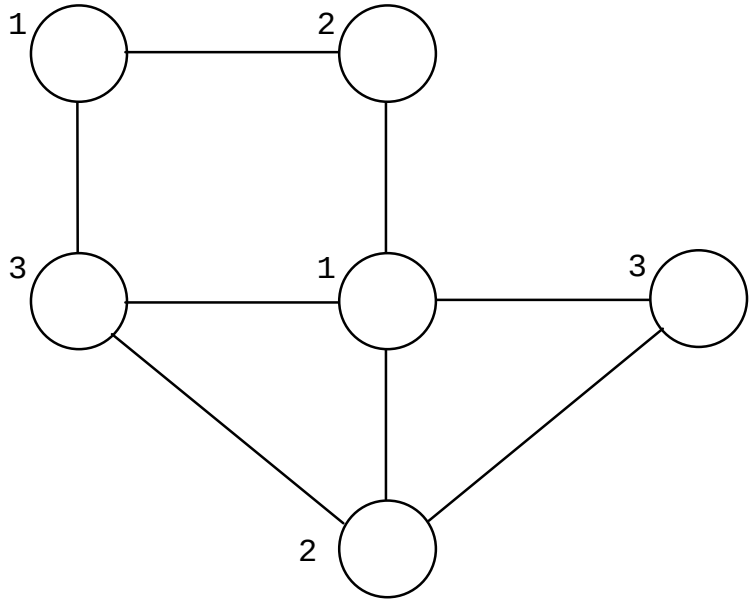
The graph colouring problem:

- Label each node of a graph with *colours* so that
  - neighbouring nodes are labelled with different colours
  - *and* the number of colours is minimized
    (or the number of colours is no more than k).

Chromatic number of a graph:

minimum number of colours needed to colour the graph.

Example:



Graph colouring algorithms

Graph colouring problem is *NP-complete*:

- Any algorithm must take exponential time to solve it.
- Approximate algorithms exist, with linear complexity.

Finding optimal colouring

Assume graph has *n* nodes.

Worst case: chromatic number is *n*.

To find optimal colouring:

```
col = colour_graph(n);
while (col != NONE) {
   best = col;
   bestn = number_of_colours(col);
   col = colour_graph(bestn-1);
}
```

where `colour_graph(`*k*`)` finds a colouring using no more than *k* colours.

Simple exact backtracking algorithm

Algorithm to colour graph (*n* nodes) using at most *k* colours:

`node[1], ..., node[n] =` list of nodes in graph;

`colour[1], ..., colour[n] =` list of colours used;

`colour_graph(k)`:

```
ac = {1,...,k};  /* set of available colours
i = 0;
while (1) {
  i = forward(i);
  if (i == n) return(colour[1,...,n]);
  i = backward(i);
  if (i == 0) return(NONE);
}
```

forward(i):

```
while (i < n) {
  i = i+1; v = node[i];
  fc[v] = ac;
  for (j = 1; j < i; j++)
    if (neighbour(v,node[j]) remove(colour[node[j]],fc[v]);
  if (empty(fc[v])) return(i-1);
  colour[v] = min(fc[v]);
  remove(colour[v],fc[v]);
}
return(n);
```

backward(i):

```
while (i >= 1) {
  v = node[i];
  if (fc[v] == empty) i = i-1;
  else {
    colour[v] = min(fc[v]);
    remove(colour[v],fc[v]);
    return(i);
  }
}
return(0);
```

Fast approximate graph colouring algorithm

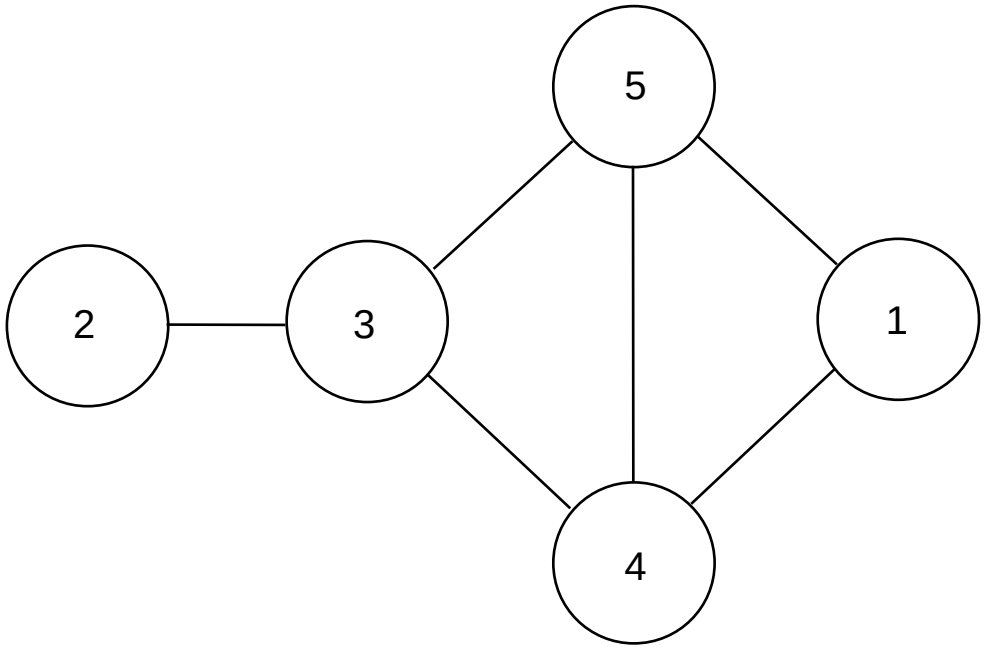
Algorithm to colour graph (*n* nodes) using at most *k* colours.
Approximate: might not succeed even if there is a solution.
Complexity: linear, not exponential.

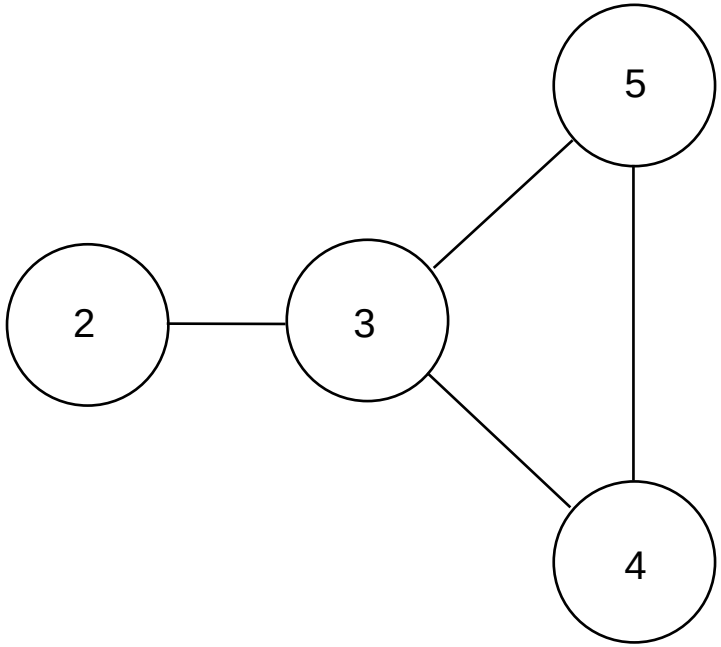
`colour_graph(k)`:

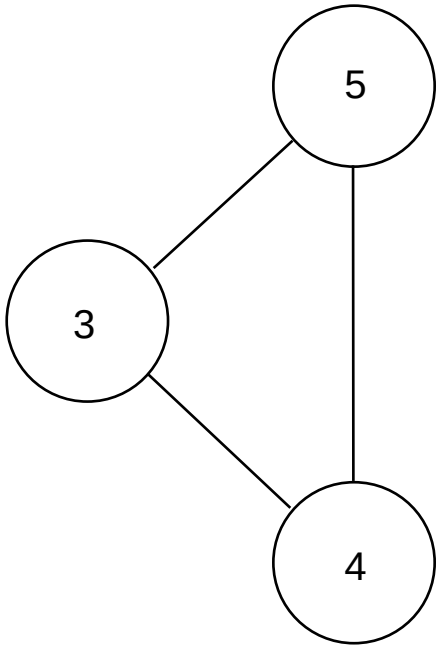
```
if (graph not empty) {
  if (graph contains node u with degree < k)
    v = u;
  else v = any node;           // potential spill
  remove node(v) from graph;
  colour_graph(k);
  add node(v) to graph;
  if (v's degree < k)
    colour[v] = any colour from 1..k not used by neighbours;
  else
  if (some colour c from 1..k is not used by neighbours)
    colour[v] = c;
  else                         // actual spill
    colour[v] = NONE;
)
```

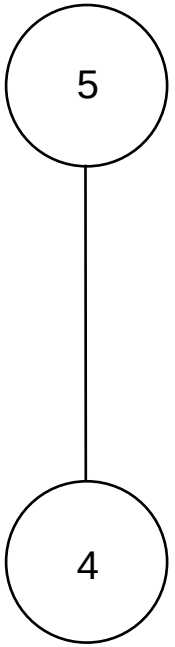
Graph colouring example

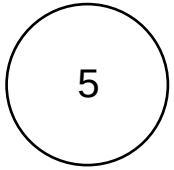
Try to colour graph with at most 3 colours:

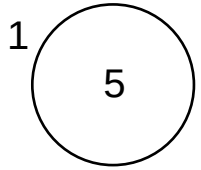




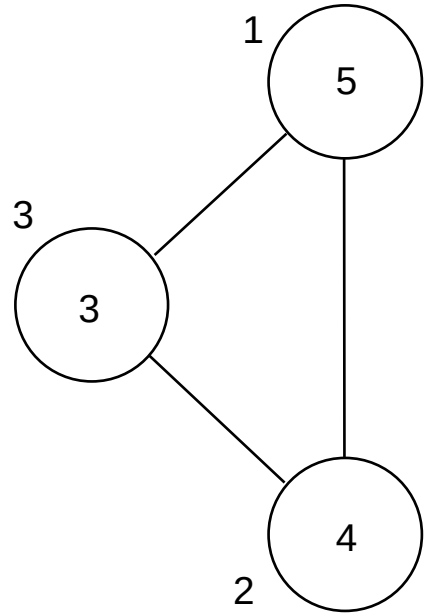

5
3
4

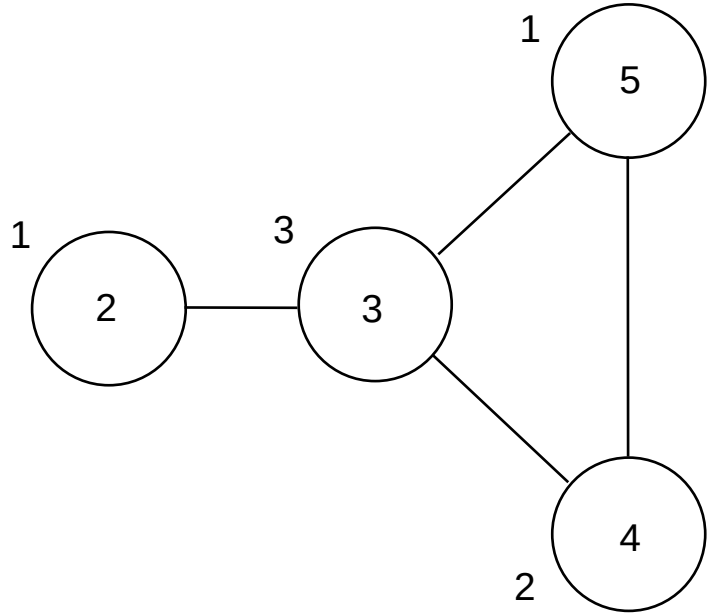

5
4

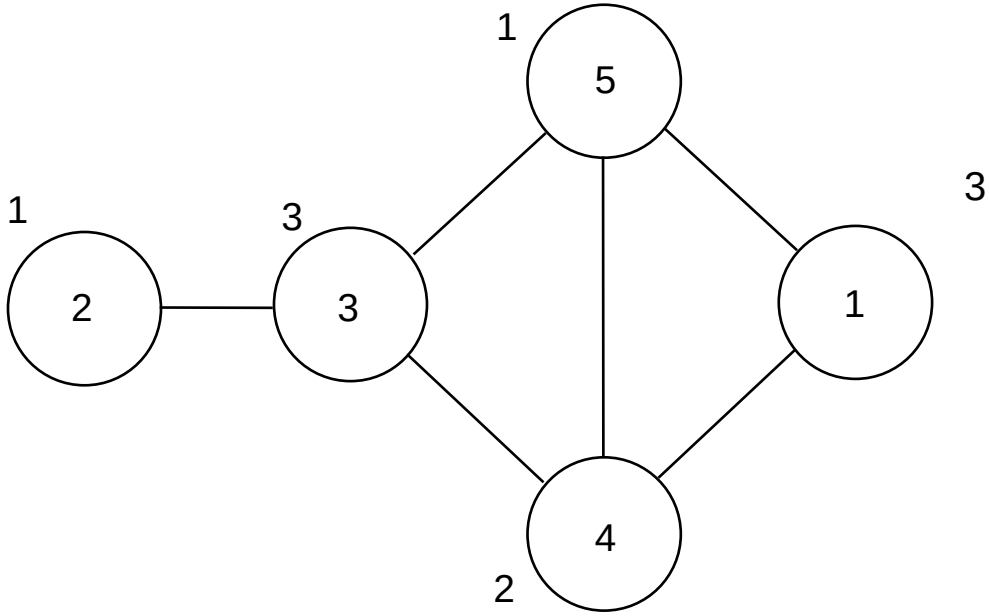

5


1
5


1
5
4
2


1
5
3
3
4
2


1
5
1
3
2
3
4
2


1
5
3
1
3
2
3
1
4
2