Compilers - Chapter 5:
Register allocation for function calls

- Lecturers:
  - Paul Kelly (p.kelly@imperial.ac.uk)
  - Naranker Dulay (n.dulay@imperial.ac.uk)
- Materials:
  - materials.doc.ic.ac.uk, Panopto
  - Textbook
  - Course web pages (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza (https://piazza.com/class/kf7uelkyxk7aa)

Function calls

- E.g.
  price+lookup(type,qty/share)*tax
  where lookup is a user-defined function with integer arguments, and integer result
- Changing order of evaluation may change result because of side-effects
- Register targeting can be used to make sure arguments are computed in the right registers
- At the point of the call, several registers may already be in use
  - But a function might be called from different call-sites
  - Each call site might have different registers in use

Function calls: evaluation order

- Eg: f(a) + f(b) + f(c)
- Eg: g(f(a), f(b))

- Which sub-expression should we evaluate first?
  - This is a *correctness* issue

- In C++ the order is *undefined* – the compiler is free to choose an order of evaluation, even if f() has side-effects (https://en.cppreference.com/w/cpp/language/eval_order)

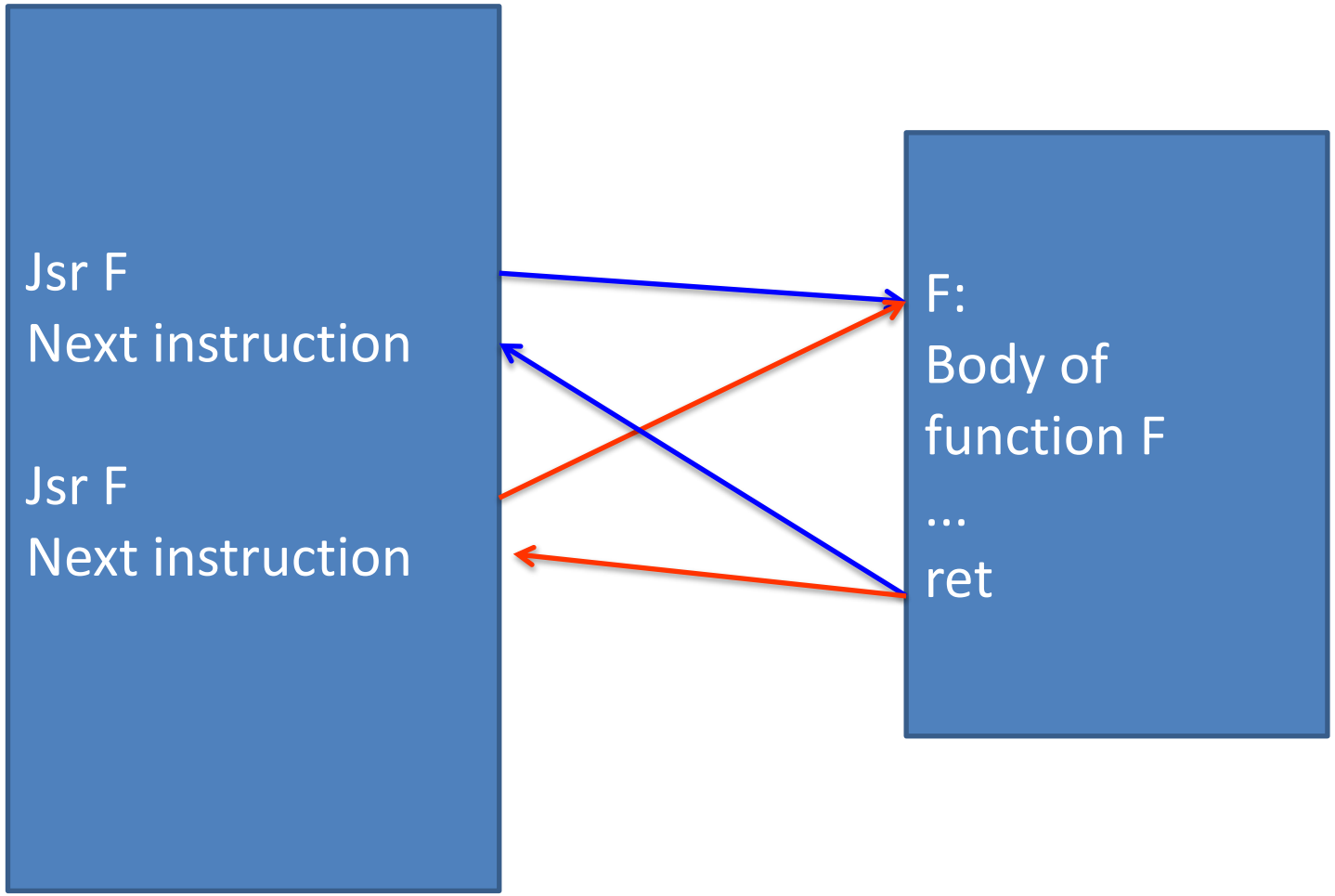
- In Java the order is left-to-right
- (https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html)

Function calls: evaluation order

- Eg: (1+f(x))*((2*y)+(3/u-z))

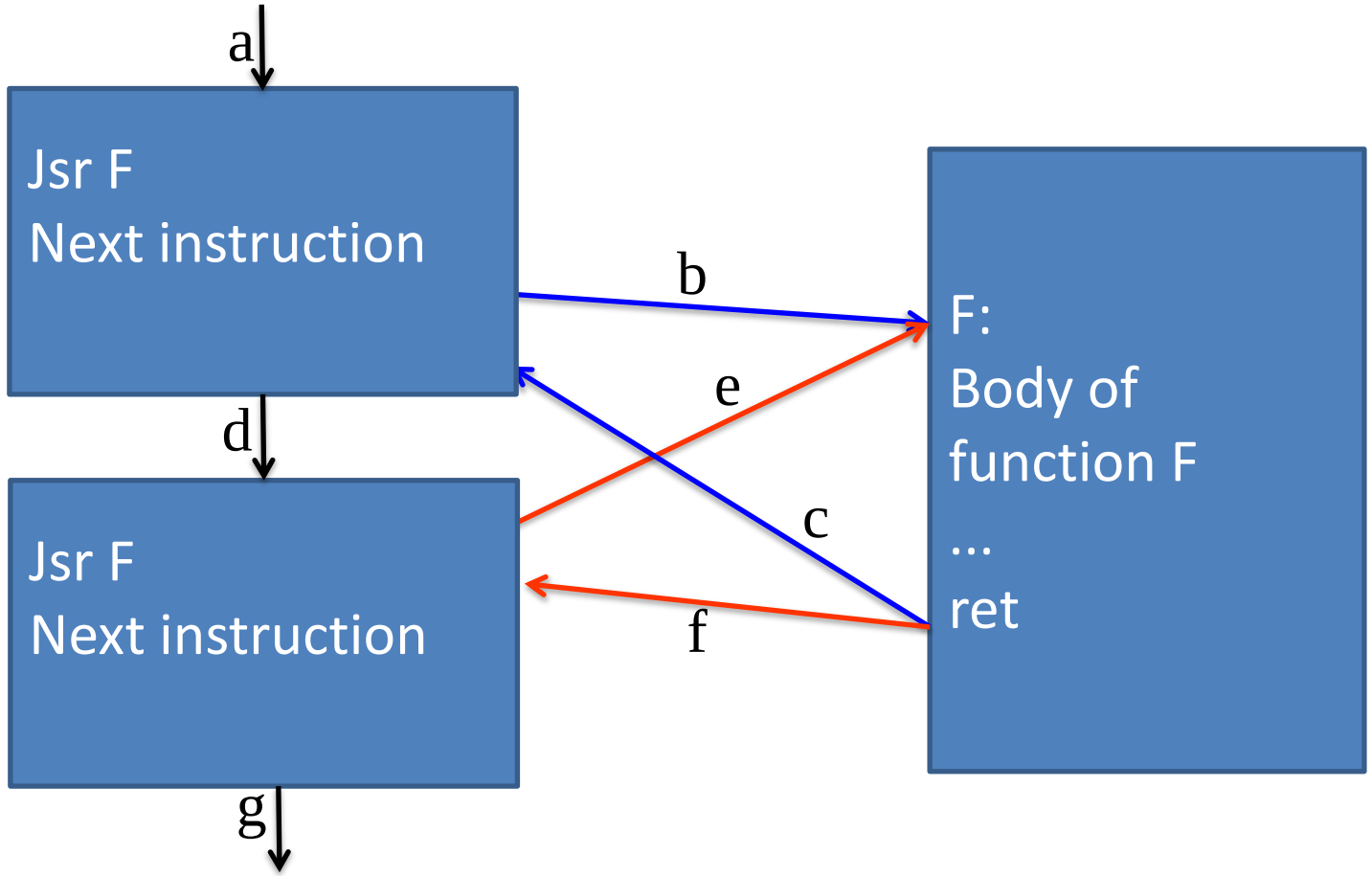
- Which one should we evaluate first?
  - This is a *performance* issue

Calling a function/method



- A function might be called from different places
- In each case it must return to the right place
- Address of next instruction must be saved and restored

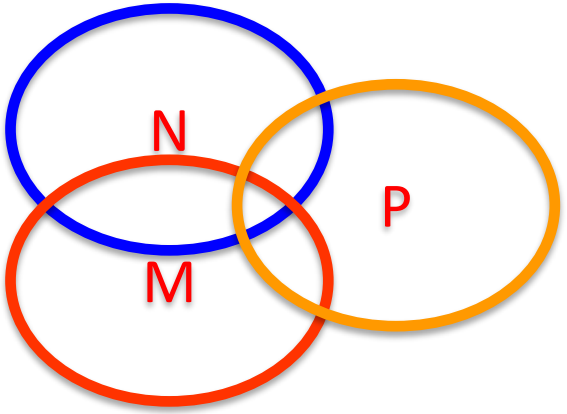
Side-observation: infeasible control-flow paths



- a,b,c,d,e,f,g is a feasible path
- a,b,f,g is a path in the graph, but is not feasible
- Control-flow graphs correctly capture control flow *inside* functions/methods, but not *between* them
- We will return to control-flow graphs shortly

Save the registers…

- We must ensure that the function being called doesn't clobber any registers which contain intermediate values:
  - *Caller saves*: save registers being used by caller in case they are clobbered by callee
  - *Callee saves*: save only the registers which callee actually uses
- Neither protocol is always the best (examples?)



N registers in use

M registers in use

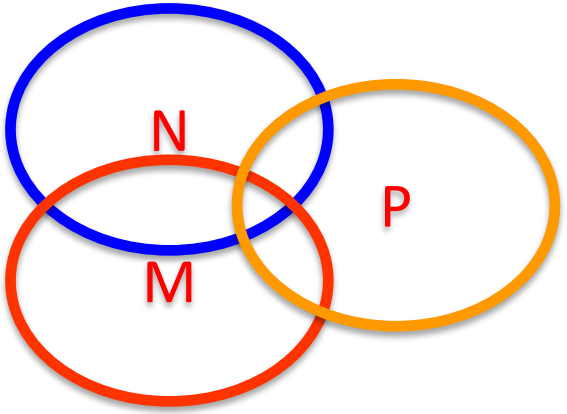
Jsr F
Next instruction

Jsr F
Next instruction

F:
Body of function F
…
ret

Needs P registers

Caller saves



N registers in use

M registers in use

Save N ∩ **P**
Jsr F
Next instruction
Restore N ∩ **P**

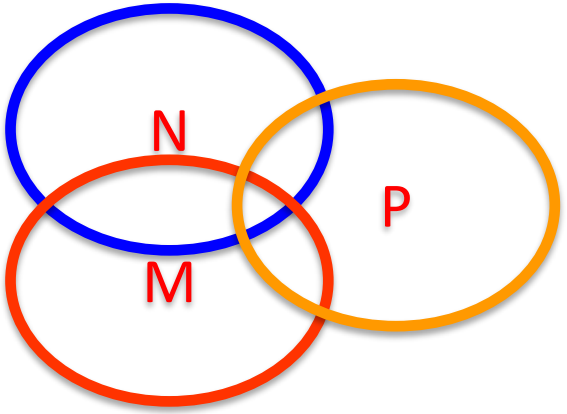
Save M ∩ **P**
Jsr F
Next instruction
Restore M ∩ **P**

F:
Body of function F
…
ret

Needs P registers

- Small problem: the **caller** doesn't know which registers the **callee** will need – it has to save any register that *might* be used

Callee saves



N registers in use

M registers in use

Jsr F
Next instruction

Jsr F
Next instruction

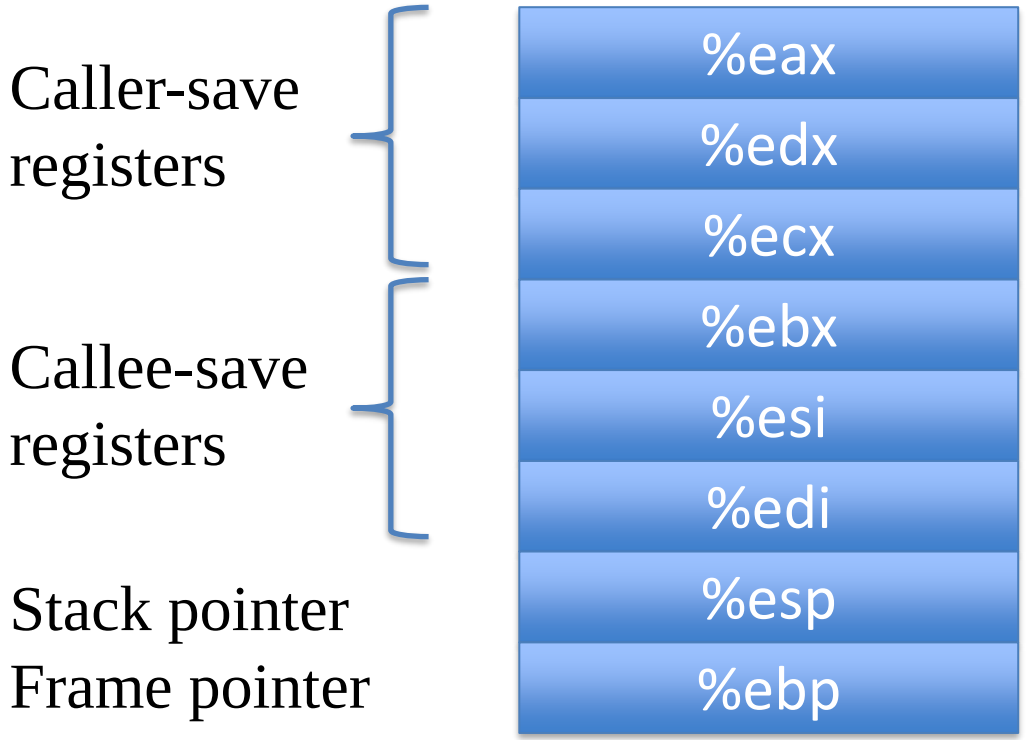
F:
Save (NUM) ∩ P
Body of function F
Restore (NUM) ∩ P
ret

Needs P registers

- Small problem: the **callee** doesn't know which registers the **callers** will need – it has to save any register that *might* be in use

Intel IA32 register saving convention

- In general you can generate any code you like, but if you want it to work with other people's libraries, you have to obey the platform's Application Binary Interface (ABI)



- Eg for Intel IA32 running Linux
- Actually there are many more rules
- Eg parameter passing, stack frame layout
- Eg virtual function tables for C++

ARM (32-bit) register saving convention

- "A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP"
- Similar rules apply to floating-point registers: s0-15 are caller-saves, s16-s31 are callee-saves

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf

MIPS register saving convention

- R2-R3: function result
- R4-R7: incoming parameters (if there are >4 parameters, they are passed on the stack)
- R16-R23: callee-save
- R8-R15: caller-saves temporaries
- R16-R25 callee-save
- R31: procedure return address
- R30: stack pointer
- R29: Frame pointer (may be used for data in leaf procedures)
- R26, R27 reserved for operating system
- R28: pointer to global area
- (R0 is always zero)

Summary

- Function calls can occur within expressions
- In some languages, the order in which such functions are called is strictly defined
- The same issue arises with the order of evaluation of function *arguments*
- In some languages it is up to the compiler

- A function may be called from several *call sites*
- At each call site, some set of registers may be in use
- The function itself may need to use some registers
- The compiler needs to produce *one* implementation of the function body that can be used from different call sites
- Each processor type/OS has a Application Binary Interface (ABI) that specifies how function arguments and results are passed, and which registers must be preserved
- In the caller-saves protocol, the caller saves all the registers it is using
- In the callee-saves protocol, the callee saves all the registers it might use
- Typical ABIs are a hybrid – some registers are caller-saves, some callee-saves

Textbooks

- EaC covers the Sethi-Ullman algorithm briefly – see Section 7.3.1 (page 315)
- EaC's recommended solution is more complex and ambitious: they separate the problem into three stages:
  - instruction selection (Chapter 11), by tree pattern matching
  - instruction scheduling (Chapter 12), accounting for pipeline stalls
  - register allocation (Chapter 13) by graph colouring (Section 13.5)
- Appel covers the Sethi-Ullman algorithm in section 11.5 (page 260)
- Appel concentrates on graph colouring – see Chapter 11
- Graph colouring relies on live range analysis, which is covered in Chapter 10

Code generation in Appel's textbook

- In Appel's compiler, the input for the code generation phase is a low-level intermediate representation (IR), which is defined in Figure 7.2 (page 157)
- The translation from the AST to the IR Tree is where:
  - 'For' loops are translated into 'Label' and jump
  - Array access is translated into pointer arithmetic + bounds checking
- The IR tree does not specify how the low-level work should be packed into machine instructions, nor which registers should be used; two phases follow:
  1. Instruction selection (Chapter 9)
  2. Register allocation (Chapter 11)
- Instruction selection works with 'temporaries' – names for locations for storing intermediate values; register allocation determines how temporaries are mapped to registers

Code generation in Appel's textbook…

- The data type for Instructions is defined on page 210 (section 9.3)
- In an instruction set like Intel's IA-32, an instruction may often perform several elementary operations in the IR Tree
- Instruction selection consists of pattern-matching – finding chunks of IR Tree corresponding to an instruction; Appel implements this in much the same way as we did in Haskell – see Programs 9.5 and 9.6 (page 213-4)
- Appel's 'translate_exp' function 'munchExp' has the effect of emitting the assembler instructions, and returns as its result the temporary in which the result will be stored

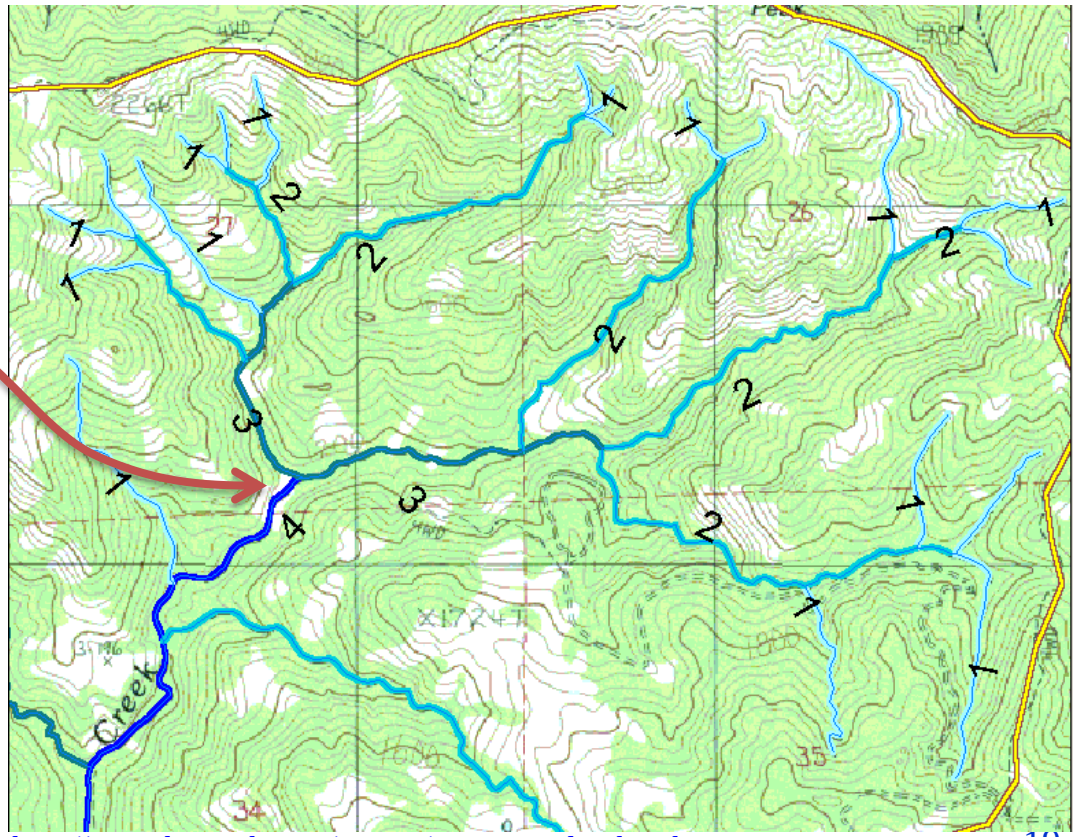
Code generation in Appel's textbook…

- We allocate registers *before* (or at the same time as) instruction selection
- Appel selects instructions first, *then* allocates registers – see discussion Section 9.3 (page 209)
- This is not a straightforward decision…
  - You may be able to avoid using a register for an intermediate value if you can use a sophisticated addressing mode
  - You may run out of registers, so some temporaries will have to be stored in memory – which changes the addressing modes you will be able to use
- Our approach leads to a much simpler compiler; see discussion on page 269

Feeding curiosity… (*way* off-topic)

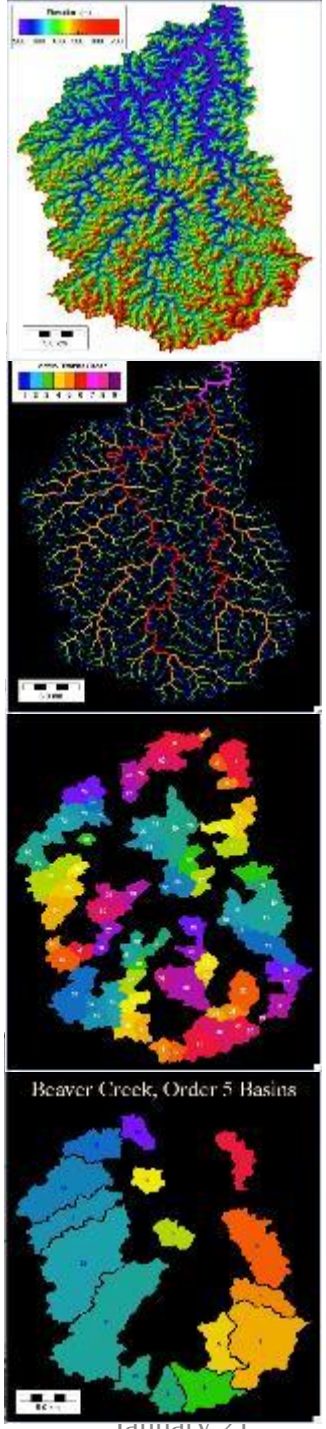
- The Sethi-Ullman-Ershov "weights" idea can be found elsewhere – where it goes by the name of Horton-Strahler number (https://en.wikipedia.org/wiki/Strahler_number). In the 1940s-50s, Horton and Strahler were studying the trees formed by rivers and their tributaries

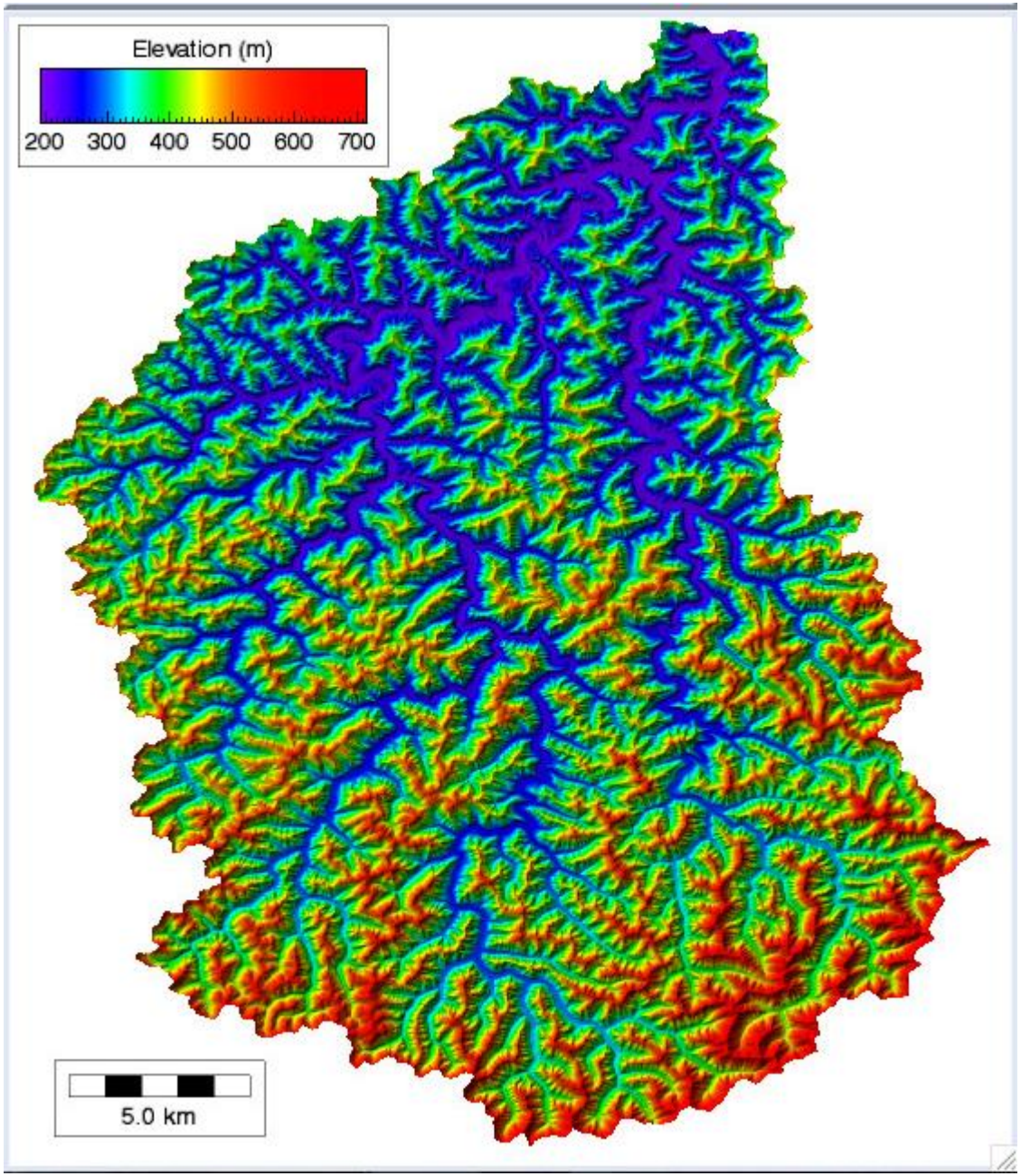
- Rivers and their tributary streams form a tree
- Here, for example, the H-S order 4 "Creek" has two order-3 tributaries
- Each river segment has a catchment area, as illustrated in the following slides
- Similar phenomena and analyses also apply to actual trees, to the vortices in fluid flow, and the development of cities



http://www.krisweb.com/stream/stream_order_kris.htm

Horton-Strahler stream order for Beaver Creek, Kentucky

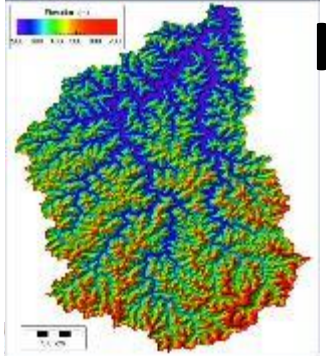


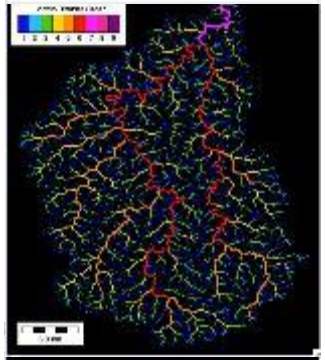


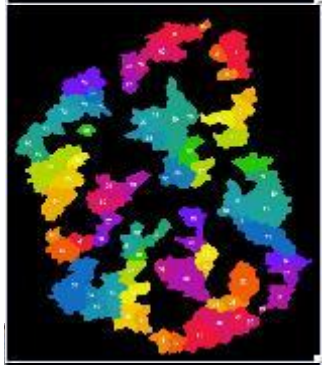
We start with the elevation map, showing the river network

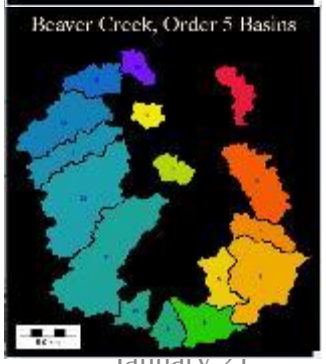
From the examples page illustrating the RiverTools software: https://rivix.com/gallery_main.php

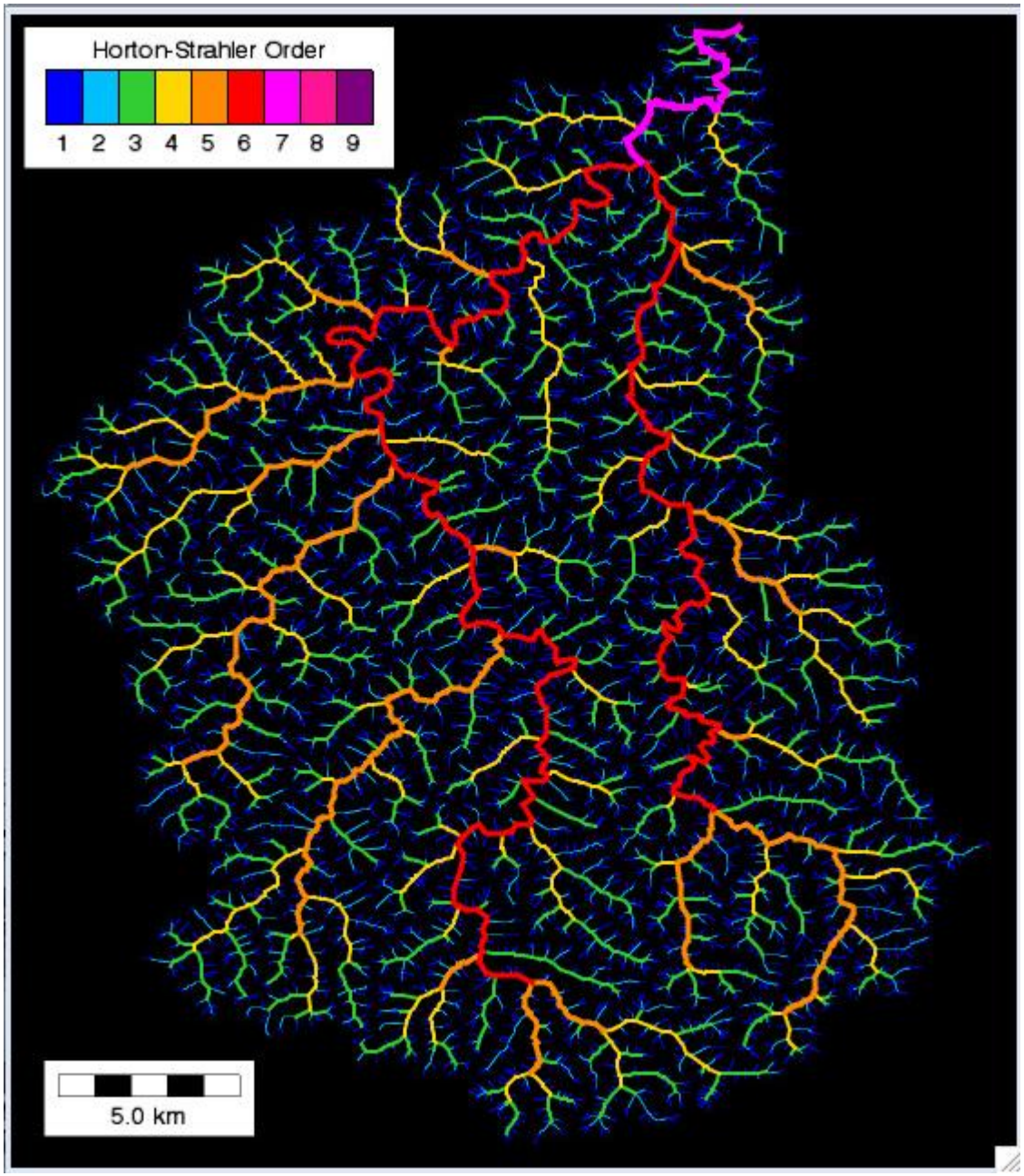
Horton-Strahler stream order for Beaver Creek, Kentucky







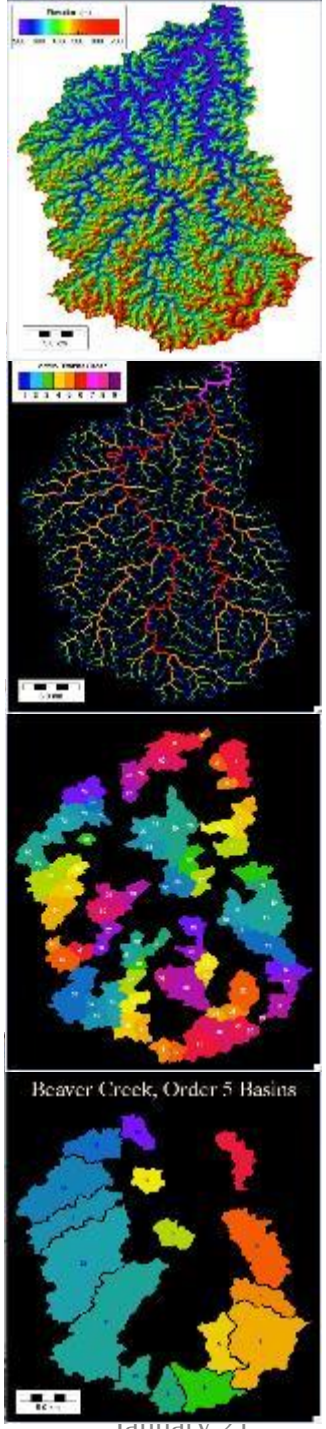


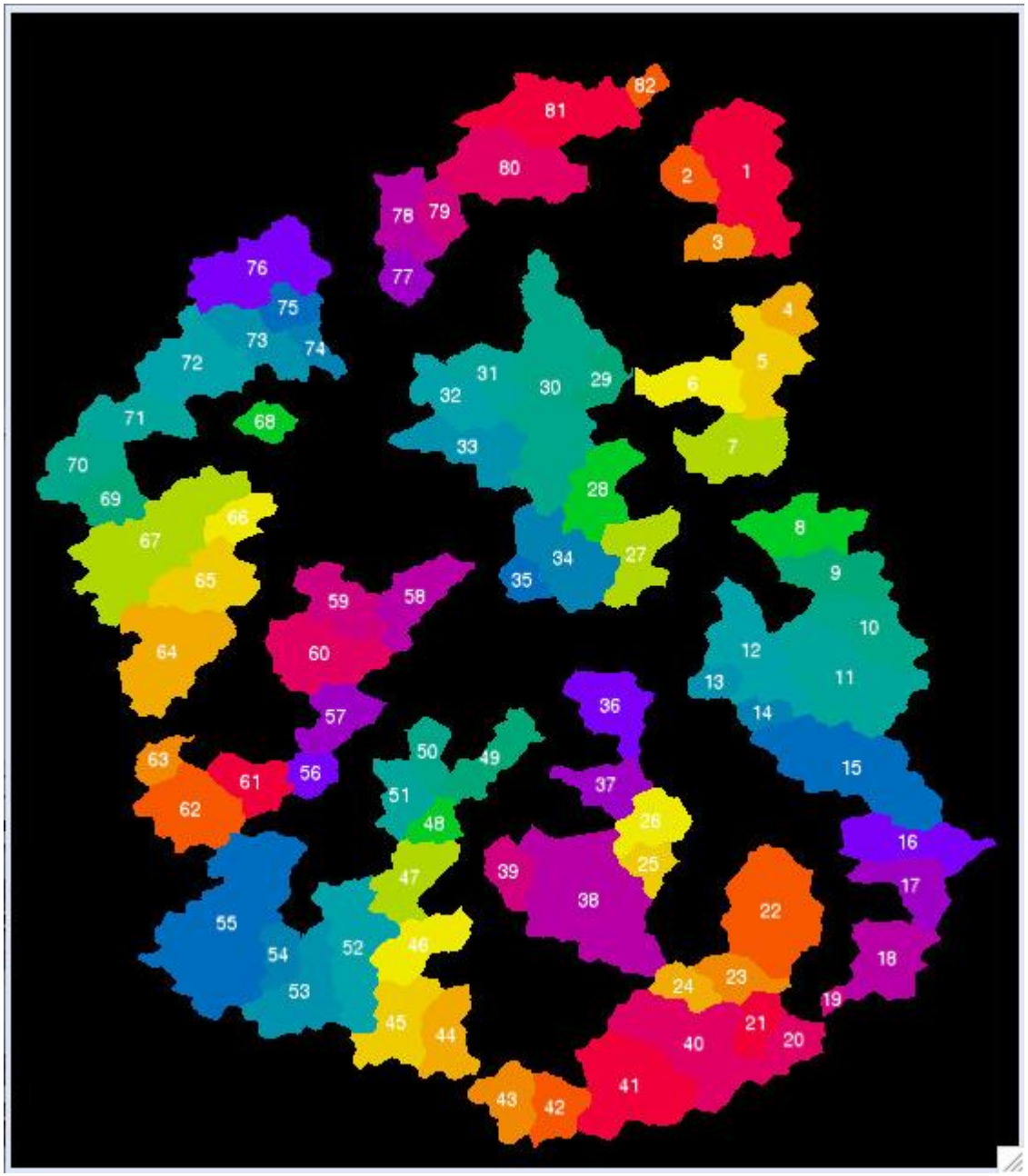


Then compute the Horton-Strahler order for each segment

From the examples page illustrating the RiverTools software: https://rivix.com/gallery_main.php

Horton-Strahler stream order for Beaver Creek, Kentucky

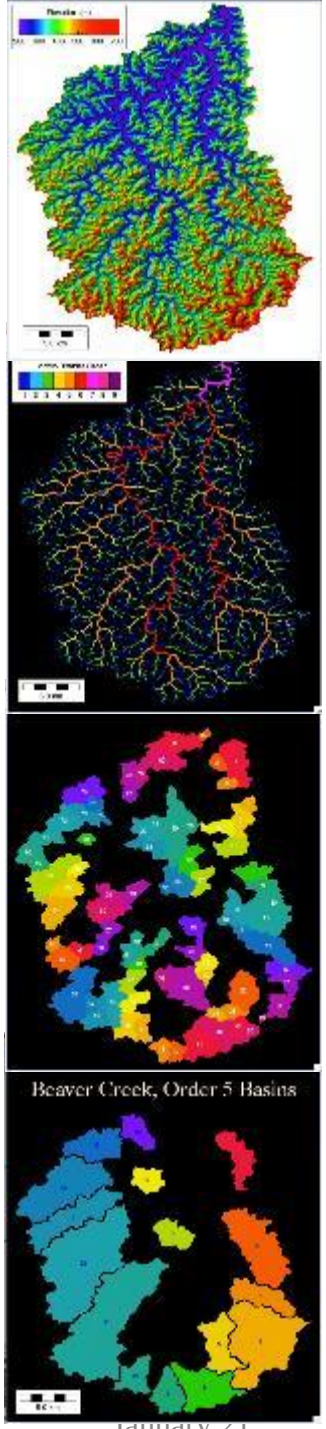


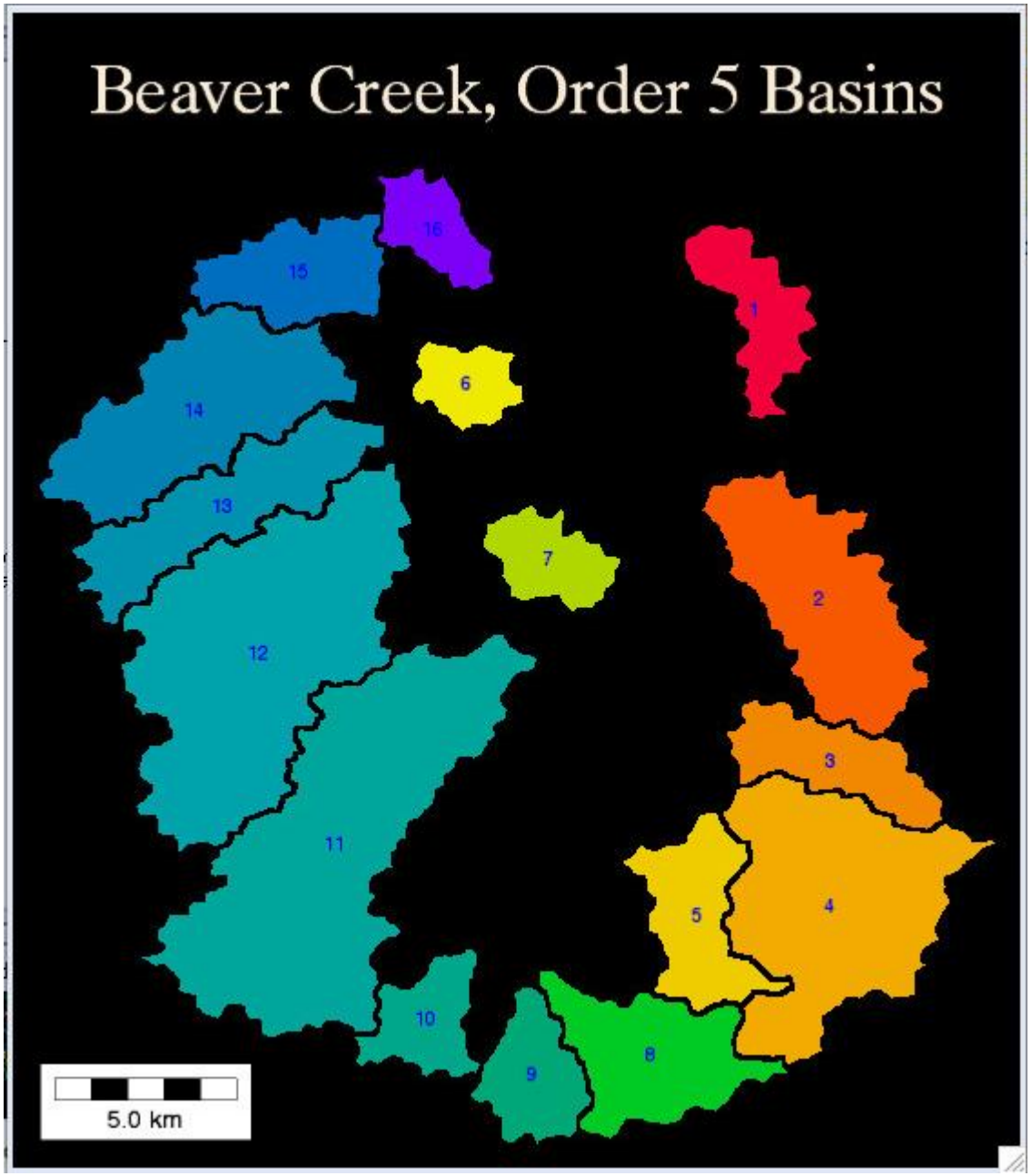


Now we can visualise the catchment areas for all the **order four** segments

From the examples page illustrating the RiverTools software: https://rivix.com/gallery_main.php

Horton-Strahler stream order for Beaver Creek, Kentucky





And here **order 5**

From the examples page illustrating the RiverTools software: https://rivix.com/gallery_main.php