Laces and Complexity

Recap: we've talked about basic knitting and colorwork

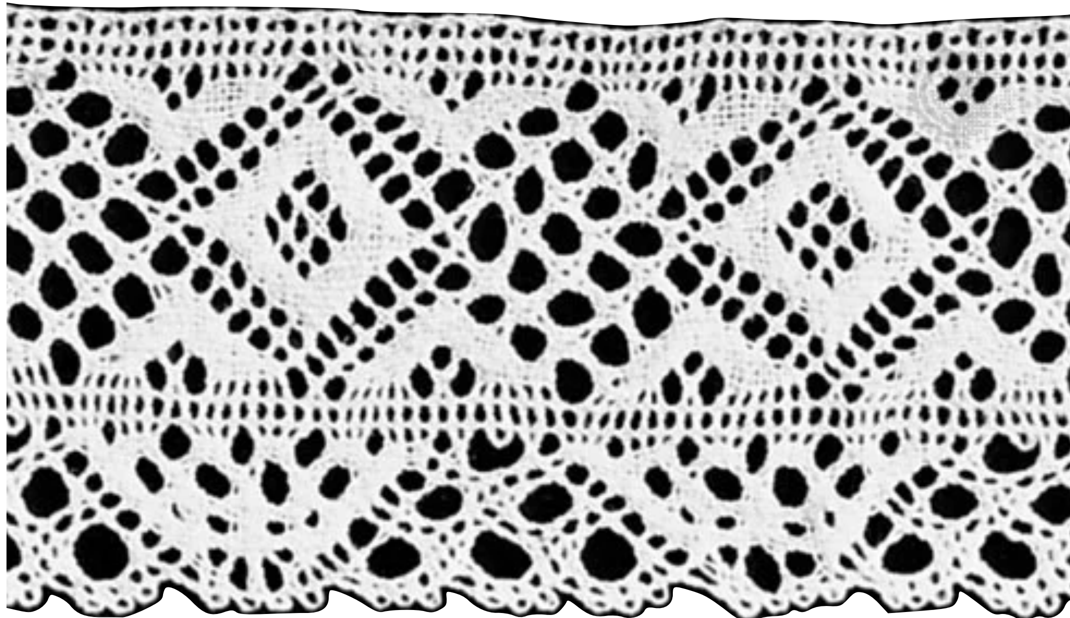
- What I hope you remember
  - knit on front bed, purl on back bed
  - colorwork involves more than one yarn and these techniques differ (most significantly) in how they handle floats (what unused yarns produce)
  - common knitout instructions (e.g., knit, xfer, inhook)

Agenda Today

- Laces
- Transfer algorithms
  - schoolbus for laces
  - what counts as a pass?
  - a pass in a program
  - another transfer algorithm CSE

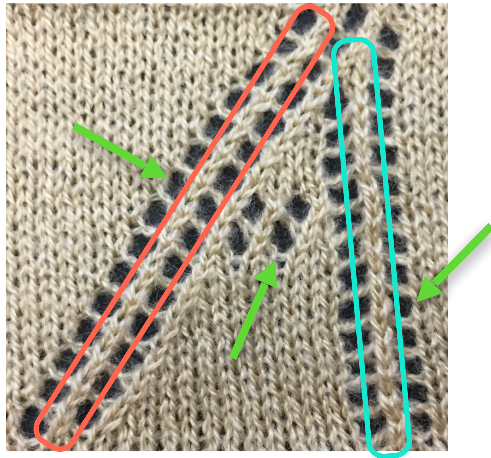
Lace in its original sense

- openwork fabric formed through looping, interlacing, twisting threads and braiding
- usually knitted or woven fabric is not considered lace, but knitting can be used to create lace-like structures that have open areas

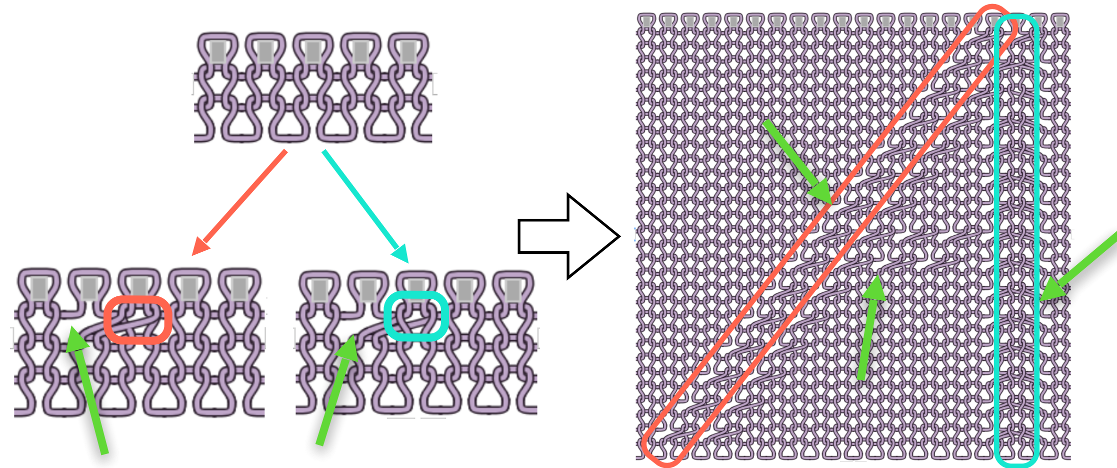


Torchon lace from Sweden, 19th century; in the Institut Royal du Patrimonie Artistique, Brussels. Courtesy of the Institut Royal du Patrimoine Artistique, Brussels; photograph, © IRPA-KIK, Brussels

Knit Lace



1. Have holes in particular locations
2. Have columns of stitches in particular lines



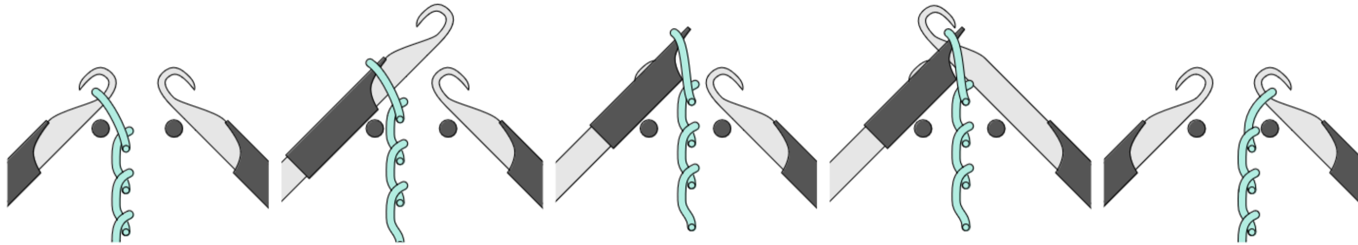
1. Leave empty needle to make hole
2. Control stacking order of loop stack

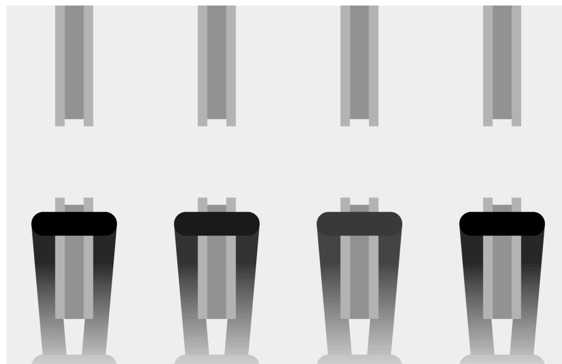
Source: Jenny Lin

How do we stack loops?

- **Transfers!**
- Transfers are essential in machine knitting because the front bed only knits and the back bed only purls, so to create more complicated patterns it’s necessary to transfer some loops from back to front or the other way

Transfers



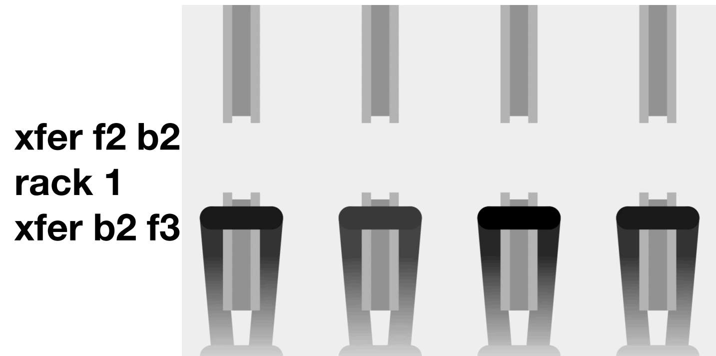


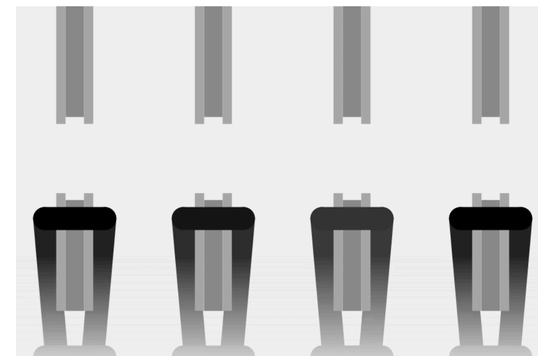
Transfers move loops from a needle to the needle across from it

xfer f3 b3
xfer b3 f3

Source: Jenny Lin

Stacking Loops





xfer f3 b3
rack -1
xfer f2 b3
rack 0
xfer b3 f3

Remember! Once two loops are on the same needle, they're stuck together forever!

Source: Jenny Lin

Knit Program Efficiency

Source: Jenny Lin

Transfers are not “cheap” operations

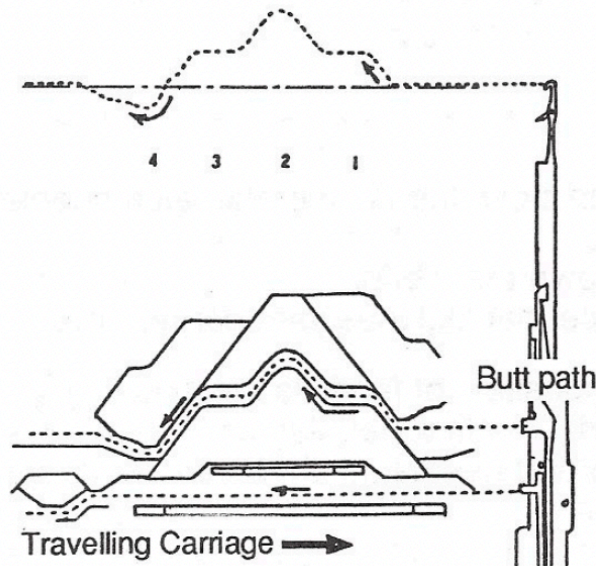
- time-wise, compared to plain knitting back and forth
- why?
  - because the CAM (carriage) moves back and forth to push needles up to perform operations, and the more needles it can control at the same time, the fewer passes of back and forth it needs to do

Carriage and CAM

2. Cam

CAM is a mechanical technical word. In the knitting machine the knit cam has a very important function in that it is the device that converts the left / right stroke operation of the carriage into the up /down operation of the needle, making the shape of the stitch. It is necessary to understand stitch shape responding to the cam operation.

(Fig.) Knit Cam Shape and Needle Axis



(1) The needle rises

(2) Clearing

(3) The needle descends

(4) Knock over

source: ABC of Flat Knitting

The knitting cam is called a single cam, double cam, 3cam, 4cam, depending on the number of cam aligned in a row. The efficiency of knitting improves as the number of cams increase.

Knit Program Efficiency $\approx$ # Passes

Source: Jenny Lin

What counts as a single pass?

- Knits and tucks can share a single pass (if they are in the right order)
- Transfers can share a single pass if they happen at the same racking

```
;;carrier starts at f0
knit + f1 6
tuck + f2 6
knit + b3 6
```

1 pass

```
;;carrier starts at f0
knit + f1 6
tuck - f2 6
knit + b3 6
```

3 passes

Are these patterns different?

These are two different patterns

```
xfer f1 b0
xfer f2 b3
xfer f3 b2
```

3 passes

```
xfer f1 b0
xfer f3 b2
xfer f2 b3
```

2 passes

Are these patterns different?

These are the same!

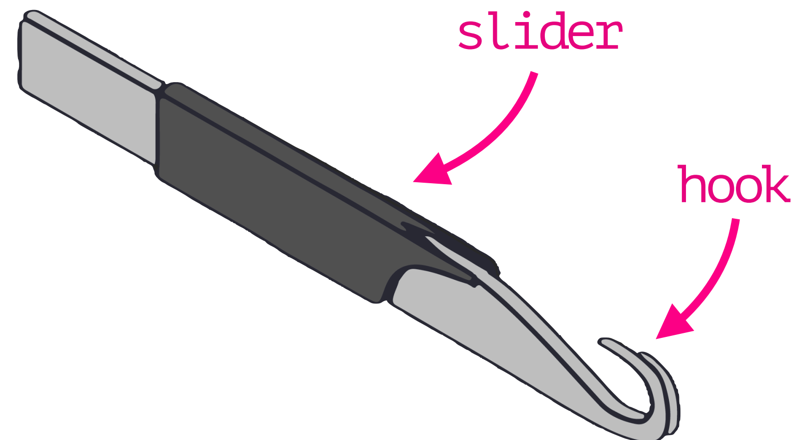
Source: Jenny Lin

Segue into transfer planning!

- **Transfer planning** is the problem that takes as input a starting needle-loop configuration and a target needle-loop configuration, and outputs a set of transfer instructions that achieves transforming from the starting configuration to the target
- Efficient transfer planning is important because machine knitting's efficiency is approximately measured by #passes and badly-planned transfers will cause a lot more **passes** than necessary and thus poor efficiency

Review the knitout language (plus one new thing, sliders)

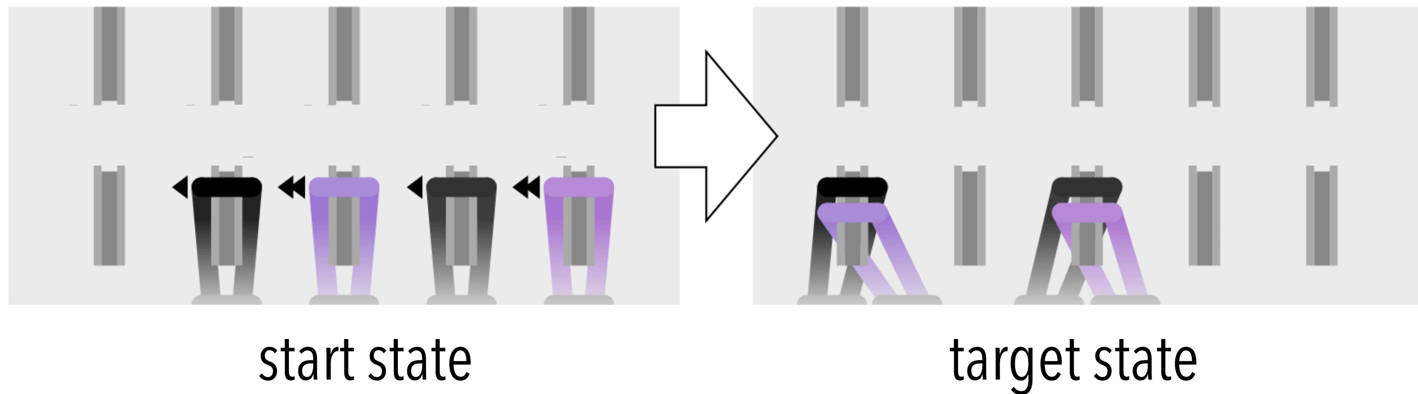
- Basic opcodes
  - tuck, knit, split
  - amiss, drop, xfer (tuck, knit, and split with **no yarns**)
- Needle reference
  - front/back bed needles: f1, f2, ...; b1, b2, ...
  - front/back bed sliders: fs1, fs2, ...; bs1, bs2, ...
- headers and setup opcodes



Transfer Planning Problem Statement

- Given a start machine state, and a target machine state, generate a sequence of instructions that can bring the machine from the start to the target.

Flat Lace Transfer Planning Problem



Machine State Representation

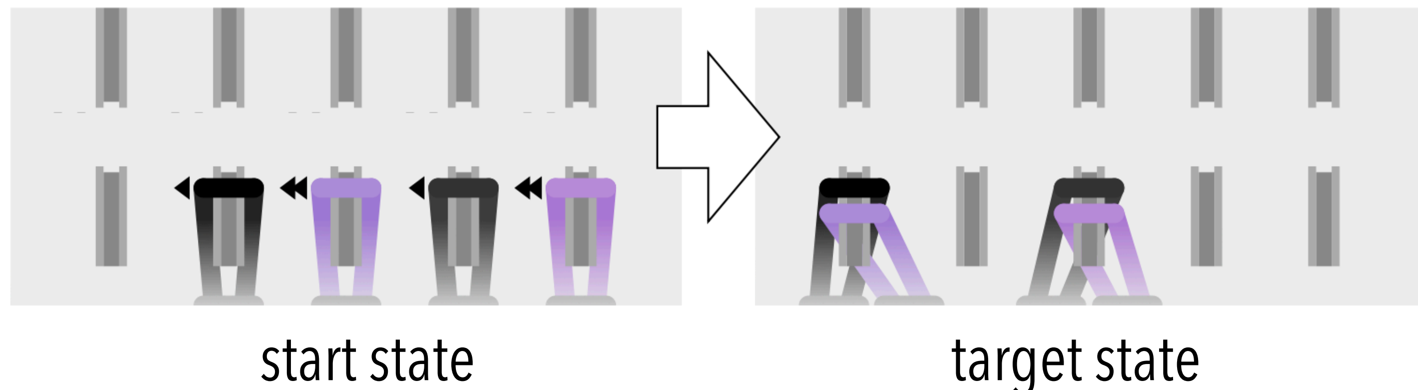
- We represent the machine state as a mapping from needle numbers to loops.
- For example, if we have five stitches on needle f1 to f5, we can write it as:

```
f1 -> n1, f2 -> n2, f3 -> n3, f4 -> n4, f5 -> n5
```

b					
bs					
fs					
f	n1	n2	n3	n4	n5
	1	2	3	4	5

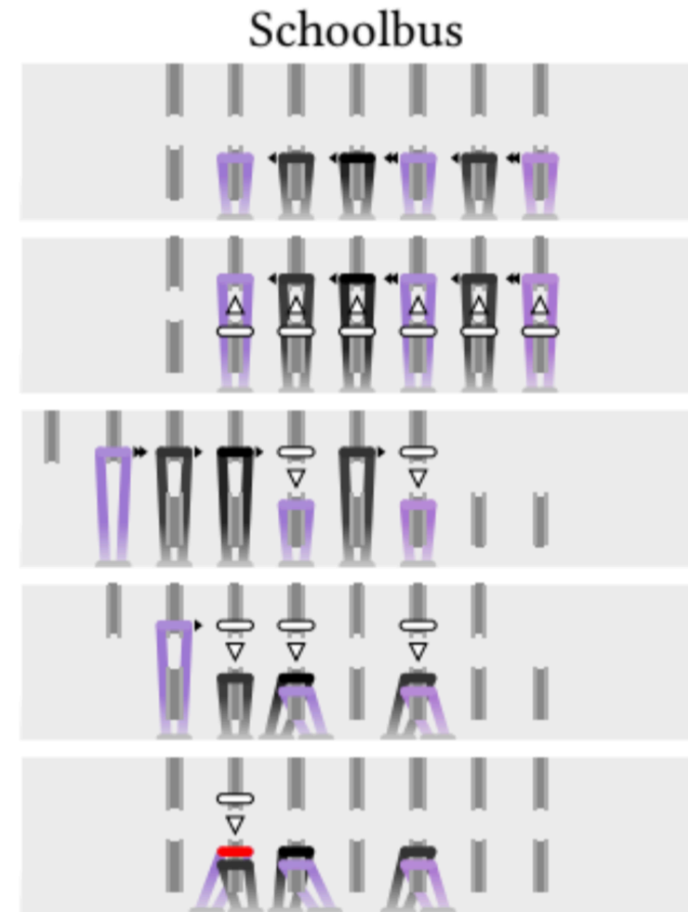
Schoolbus Algorithm

- Transport the stitches (students) that need to get off at the same offset (stop)
  - For this example, it means we transfer loops 2 and 4 first, and then loops 1 and 3

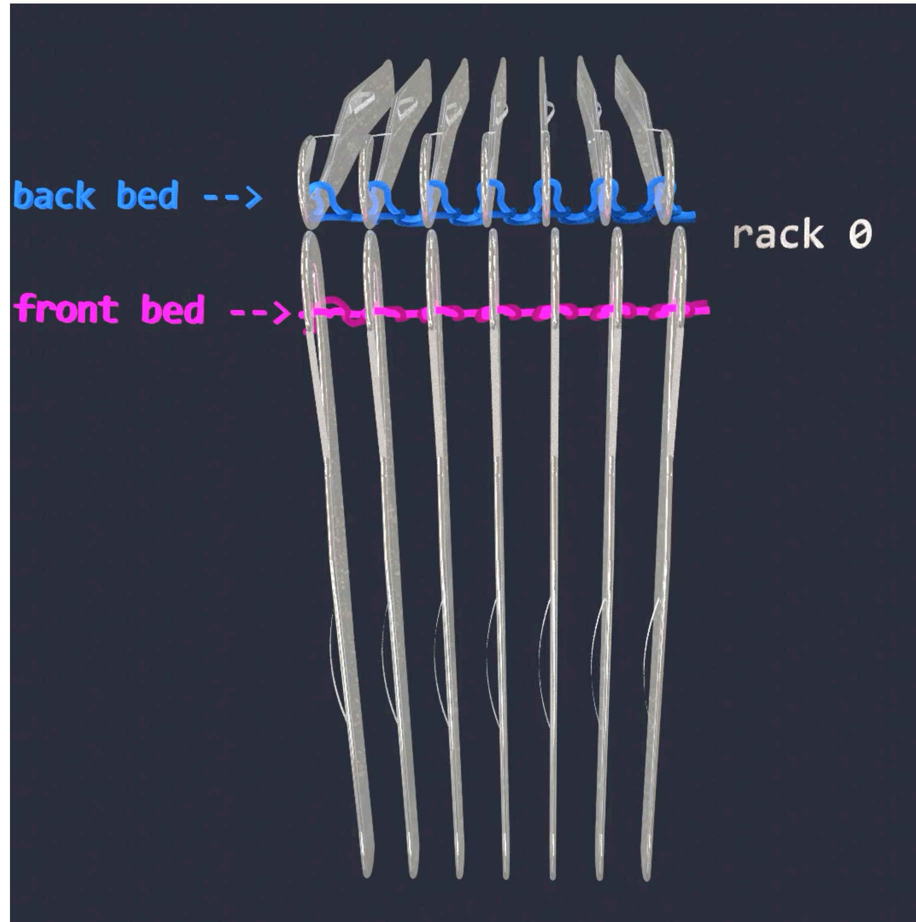


Example transfer plan for schoolbus

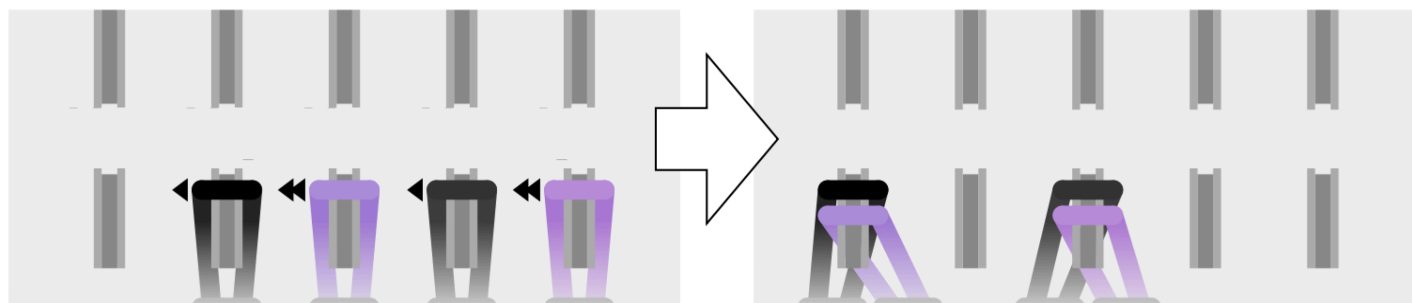
- Insight: for flat laces, knitting is flat, one bed is completely free (the back bed)
- Main idea: dropping loops off by offsets
  - move all loops to back and rack
  - move the same-offset loops back to front
  - continue until everything's moved to desired place



Recap: Racking (back bed only)



Step-by-Step Schoolbus: Setting the Stage



start state

	1	2	3	4	5
b					
bs					
fs					
f		n1	n2	n3	n4

```
f2 -> n1, f3 -> n2,
f4 -> n3, f5 -> n4
```

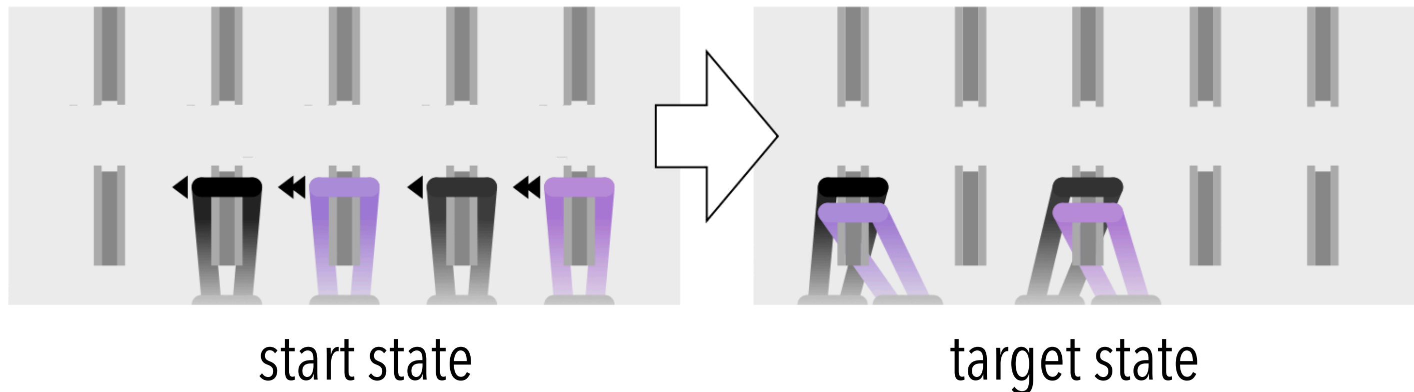
target state

	1	2	3	4	5
b					
bs					
fs					
f	n1 n2		n3 n4		

```
f1 -> (n2,n1)
f3 -> (n4,n3)
```

Let's say the convention is that the name that comes first is the loop that is stacked first

Step-by-Step Schoolbus: Finding the Reverse Lookup Mapping

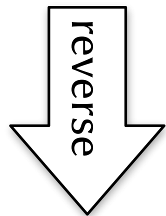


start state | target state

needle to loop

```
f2 -> n1, f3 -> n2,
f4 -> n3, f5 -> n4
```

```
f1 -> (n2,n1)
f3 -> (n4,n3)
```



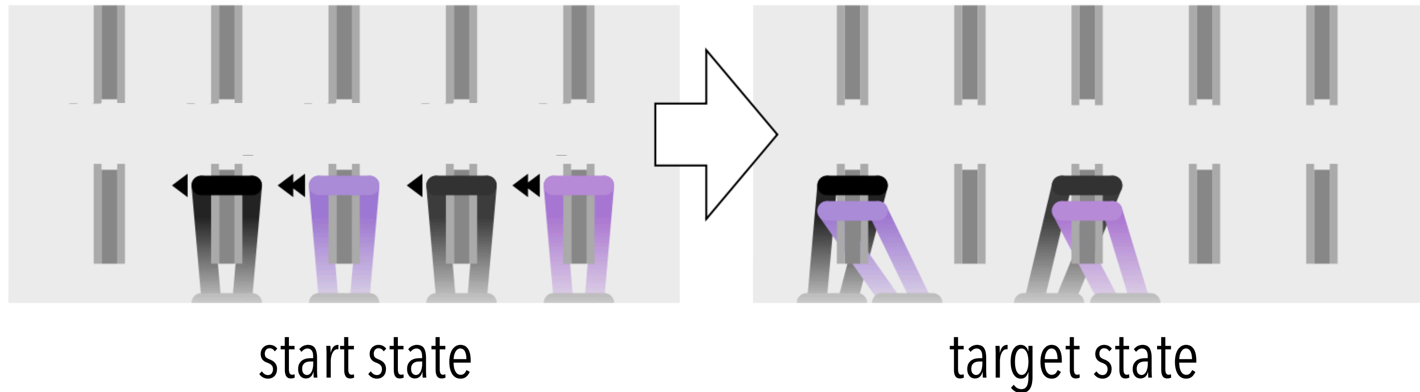
loop to needle

```
n1 -> f2, n2 -> f3,
n3 -> f4, n4 -> f5
```

```
n1 -> f1, n2 -> f1,
n3 -> f3, n4 -> f3
```

What's the offset O?

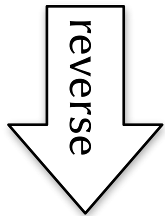
Step-by-Step Schoolbus: Compute the Offsets



needle to loop

```
f2 -> n1, f3 -> n2,
f4 -> n3, f5 -> n4
```

```
f1 -> (n2,n1)
f3 -> (n4,n3)
```



loop to needle

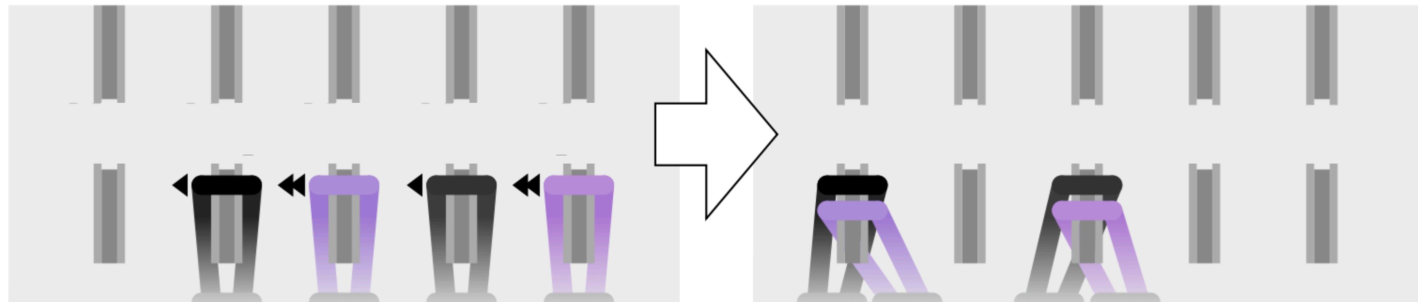
n1 -> **f2**, n2 -> **f3**,
n3 -> **f4**, n4 -> **f5**

n1 -> **f1**, n2 -> **f1**,
n3 -> **f3**, n4 -> **f3**

f4 to f3 is -1 offset

O = {n1: ?, n2: ?, n3: ?, n4: ?}

Step-by-Step Schoolbus: Start by Transferring Everything to the Back



start state

```
n1 -> f2, n2 -> f3,
n3 -> f4, n4 -> f5
```

target state

```
n1 -> f1, n2 -> f1,
n3 -> f3, n4 -> f3
```

```
O = {n1:-1, n2:-2, n3:-1, n4:-2}
```

- Everything to the back!

```
n1 -> f2, n2 -> f3,
n3 -> f4, n4 -> f5
```

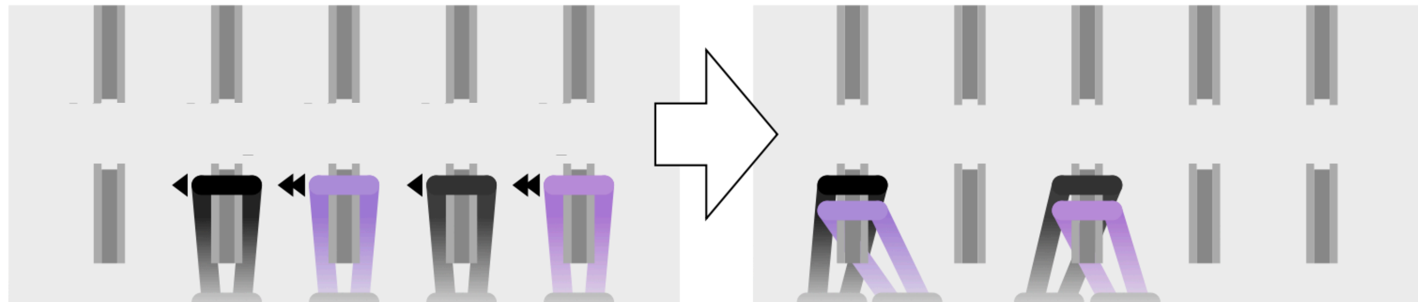
→

```
xfer f2 b2
xfer f3 b3
xfer f4 b4
xfer f5 b5
```

→

```
n1 -> b2, n2 -> b3,
n3 -> b4, n4 -> b5
```

Step-by-Step Schoolbus: Grouped Transfers Based on Offsets



start state

```
n1 -> f2, n2 -> f3,
n3 -> f4, n4 -> f5
```

target state

```
n1 -> f1, n2 -> f1,
n3 -> f3, n4 -> f3
```

```
O = {n1:-1, n2:-2, n3:-1, n4:-2}
```

- Offset at -2 loops get transferred together

```
n1 -> b2, n2 -> b3,
n3 -> b4, n4 -> b5
```

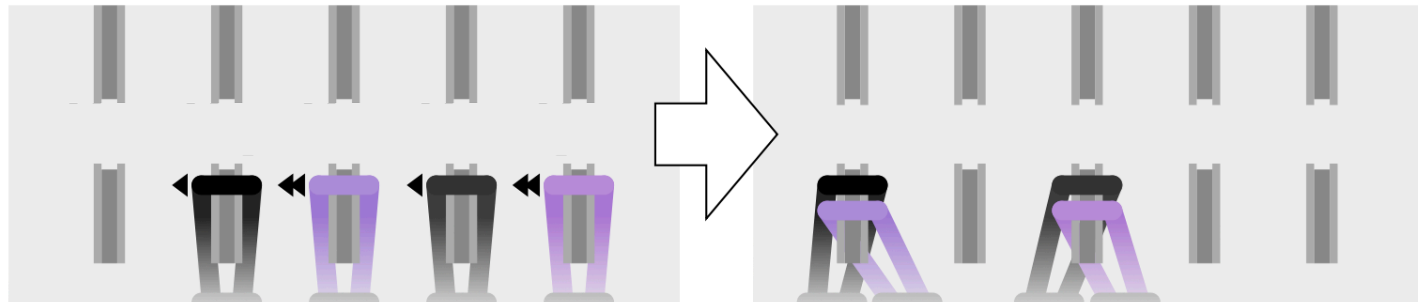
→

```
rack -2
xfer b3 f1
xfer b5 f3
```

→

```
n1 -> b2, n2 -> f1,
n3 -> b4, n4 -> f3
```

Step-by-Step Schoolbus: Grouped Transfers Based on Offsets



start state

```
n1 -> f2, n2 -> f3,
n3 -> f4, n4 -> f5
```

target state

```
n1 -> f1, n2 -> f1,
n3 -> f3, n4 -> f3
```

```
O = {n1:-1, n2:-2, n3:-1, n4:-2}
```

- Offset at -1 loops get transferred together

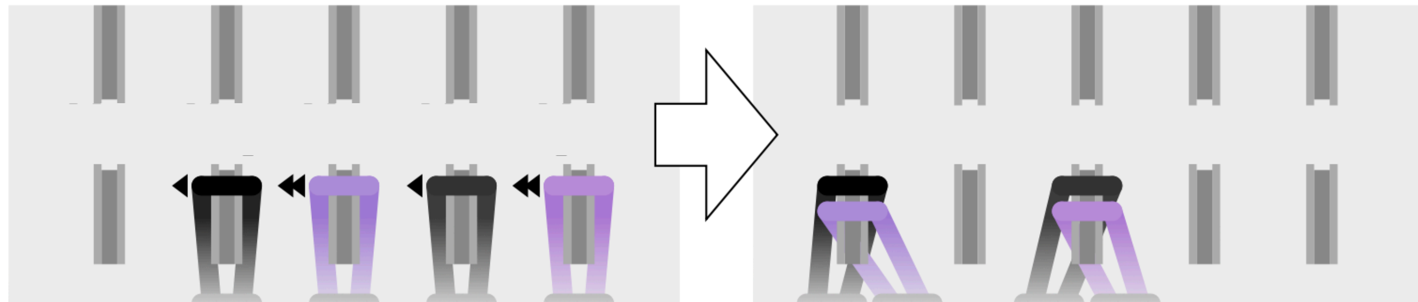
```
n1 -> b2, n2 -> f1,
n3 -> b4, n4 -> f3
```

```
rack -1
xfer b2 f1
xfer b4 f3
```

```
n1 -> f1, n2 -> f1,
n3 -> f3, n4 -> f3
```

Success!

Step-by-Step Schoolbus: Reset Back Bed



start state

```
n1 -> f2, n2 -> f3,
n3 -> f4, n4 -> f5
```

target state

```
n1 -> f1, n2 -> f1,
n3 -> f3, n4 -> f3
```

```
O = {n1:-1, n2:-2, n3:-1, n4:-2}
```

- Don’t forget to rack the back bed back to 0 position

```
rack 0
```

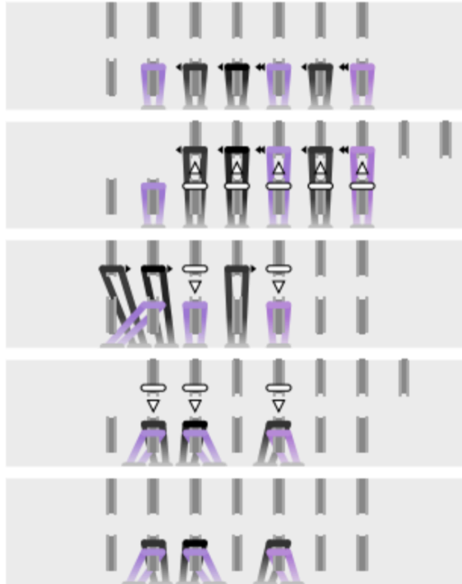
Step-by-Step Schoolbus

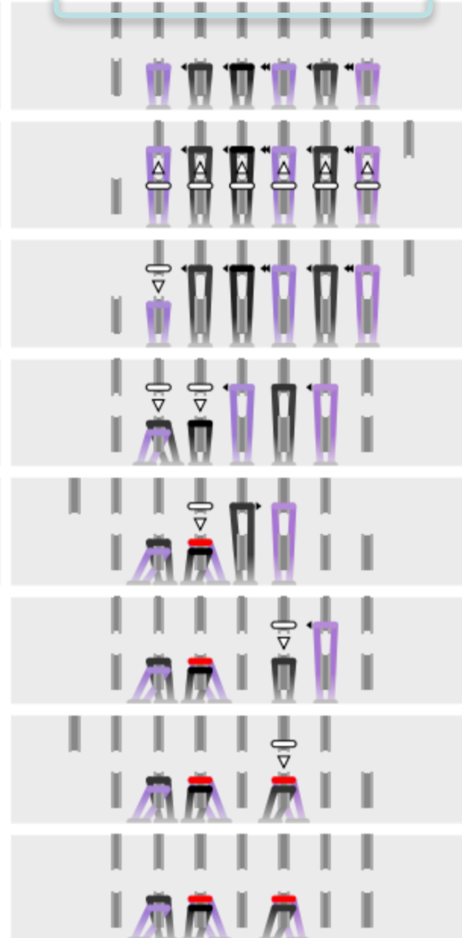
- These slides correspond to what you need to implement for HW4!
  - creating the initial machine state when all loops are on the front bed (`initial_state(n)`)
  - creating the reverse lookup mapping from loop to needle instead of from needle to loop (`build_reverse_lookup_dict(machine_state)`)
  - computing the unique racking values needed and the offsets (`unique_rackings(initial, target)`, `calculate_offsets(initial, target)`)

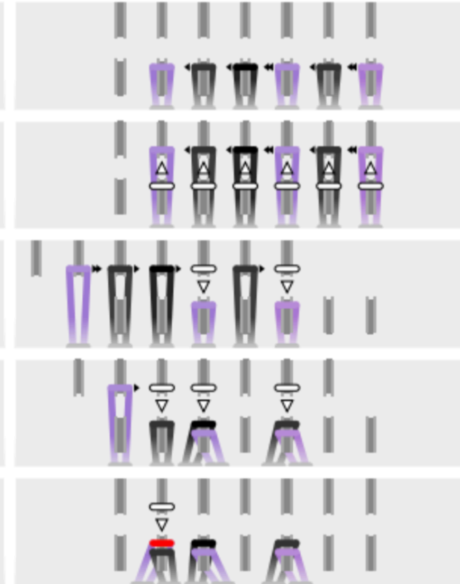
Is this the only possible way to do the transfers?

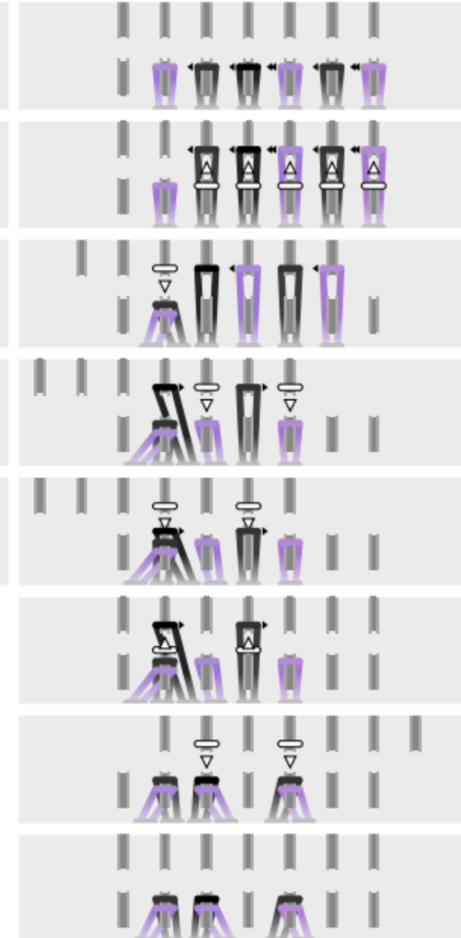
- Of course not!
- We call "schoolbus" an algorithm to solve the transfer planning problem, and by algorithm, we mean:
  - a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer (from dictionary definition)
  - basically steps you take to solve a problem
    - it's, after all, hard to imagine that there's only one way to solve a specific problem

Algorithms for the flat lace transfer planning problem









Collapse-Shift-Expand

- This is an algorithm for a more general cycle transfer problem but it can be applied to lace transfers too

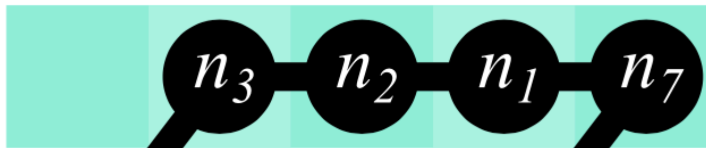
A compiler for 3D machine knitting. McCann, Albaugh, Narayanan, Grow, Matusik, Mankoff, Hodgins. ACM TOG 2016.

The cycle transfer algorithm (collapse-shift-expand)

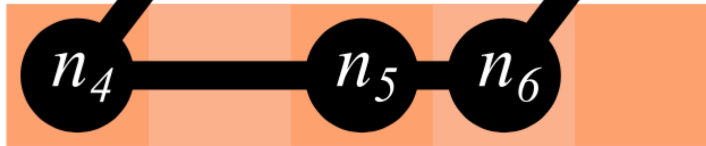
Starting configuration

Target configuration

Back bed



Front bed

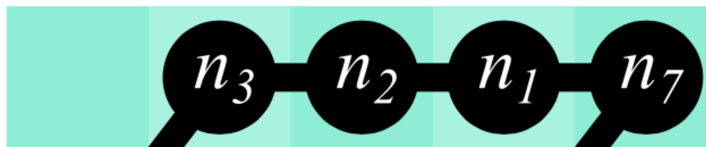


The cycle transfer algorithm

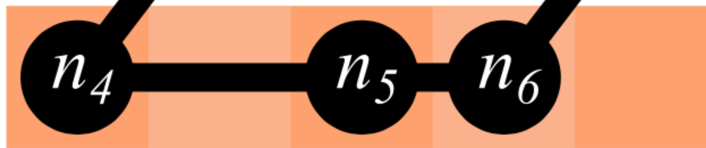
Starting configuration

Target configuration

Back bed



Front bed



b		n3	n2	n1	n7
f	n4		n5	n6	

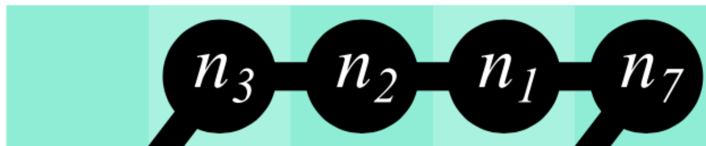
b	n7’	n6’		n5’	
f		n1’	n2’	n3’	n4’

The cycle transfer algorithm uses sliders

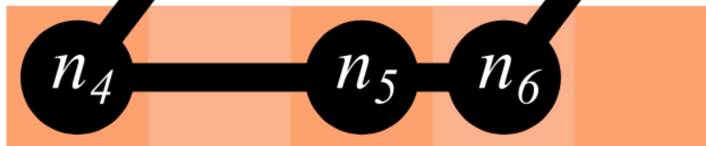
Starting configuration

Target configuration

Back bed



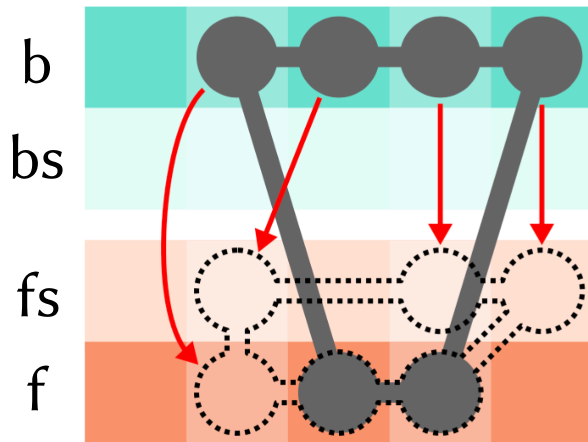
Front bed



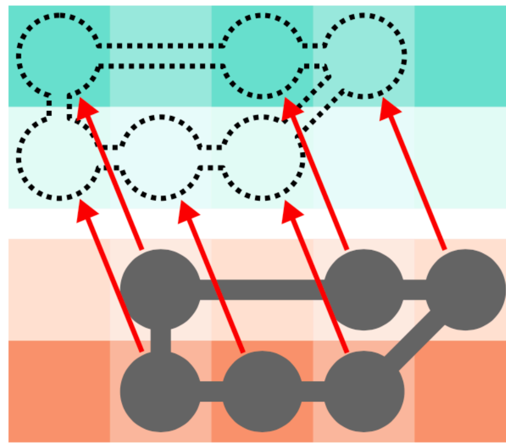
b		n3	n2	n1	n7
bs					
fs					
f	n4		n5	n6	

b	n7’	n6’		n5’	
bs					
fs					
f		n1’	n2’	n3’	n4’

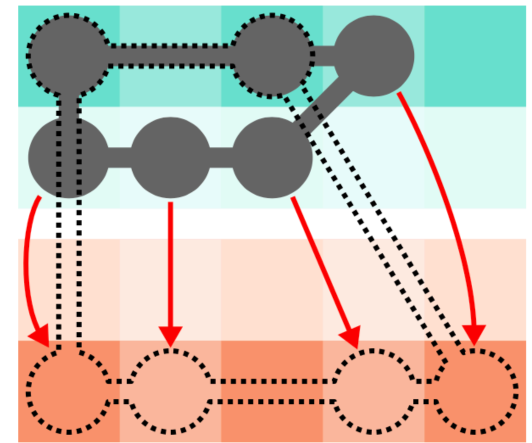
Collapse-Expand Transform



collapse to front bed



entire cycle shift to back bed



expand to front bed

One example transfer plan

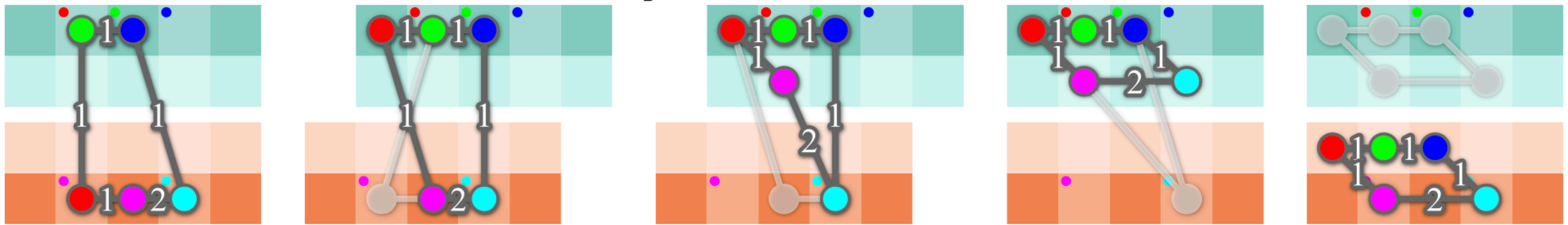
```
rack 1
xfer f2 b1
```

```
xfer f3 bs2
```

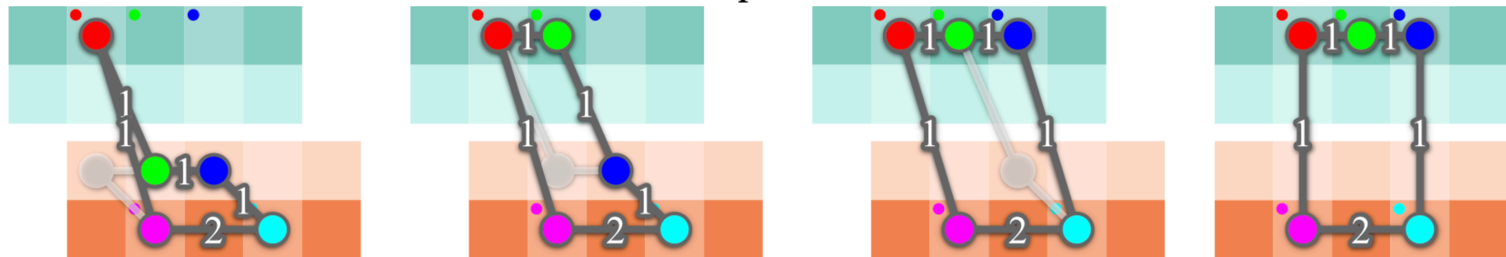
```
rack 0
xfer f4 bs4
```

```
xfer b1 fs1
xfer bs2 f2
xfer b2 fs2
xfer b3 fs3
xfer bs4 f4
```

collapse — shift



expand



```
rack -1
xfer fs1 b2
```

```
xfer fs2 b3
```

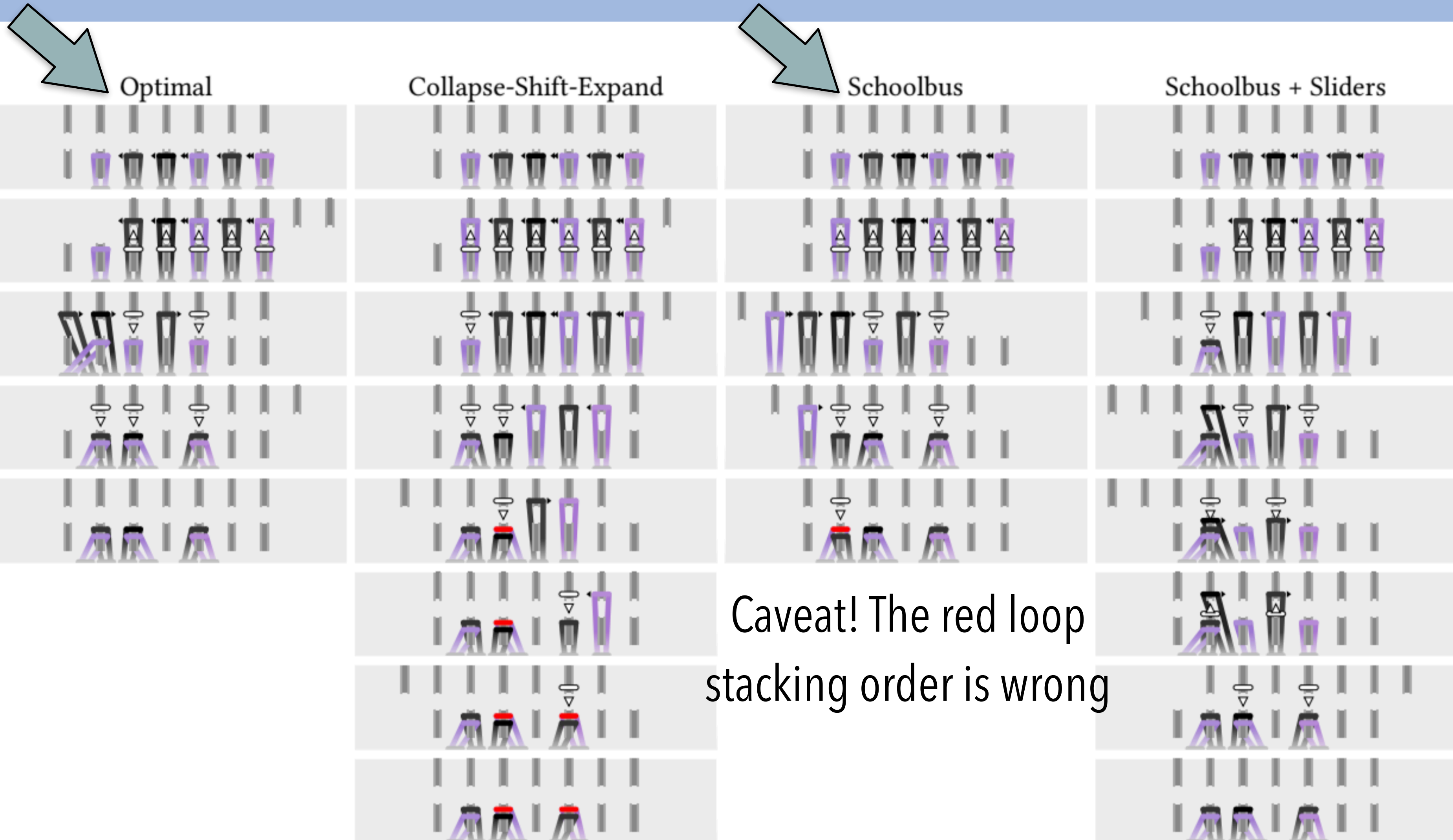
```
xfer fs3 b4
```

```
rack 0
```

Recall that Knit Program Efficiency ≈ # Passes

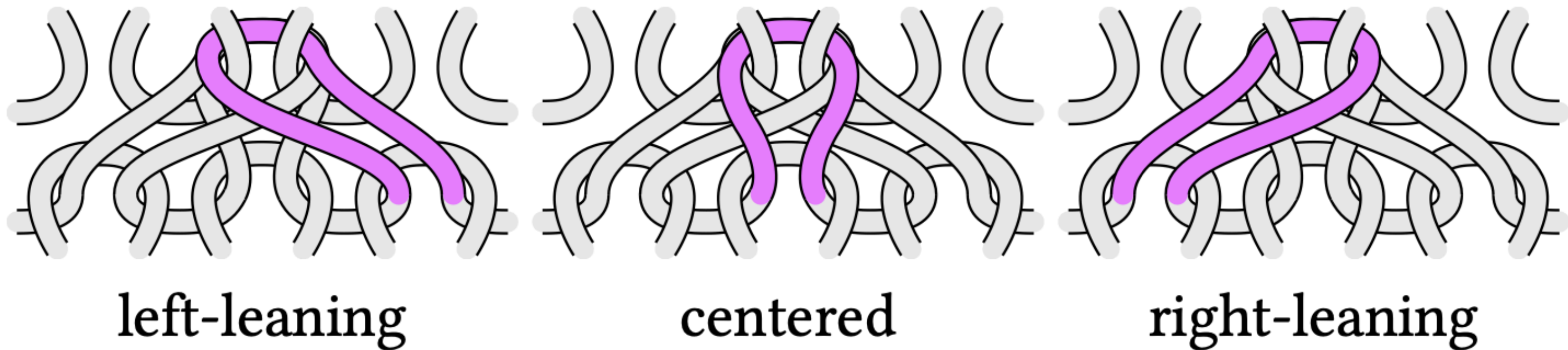
- Efficient transfer planning is important because machine knitting's efficiency is O(#passes) and badly-planned transfers will cause a lot more **passes** than necessary and thus poor efficiency

Which takes fewer #passes? (visually)

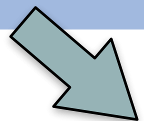


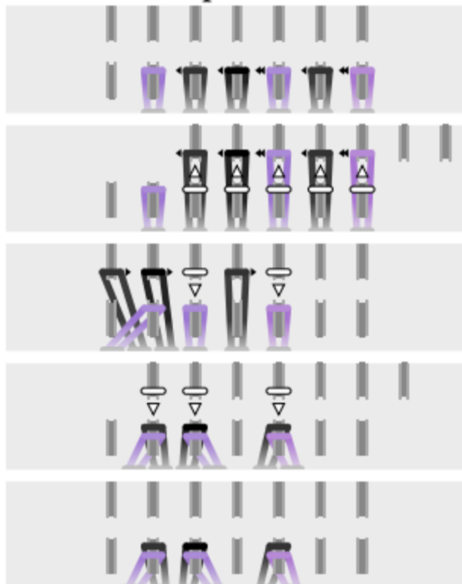
Loop stack order (might) matter for flat lace transfer planning

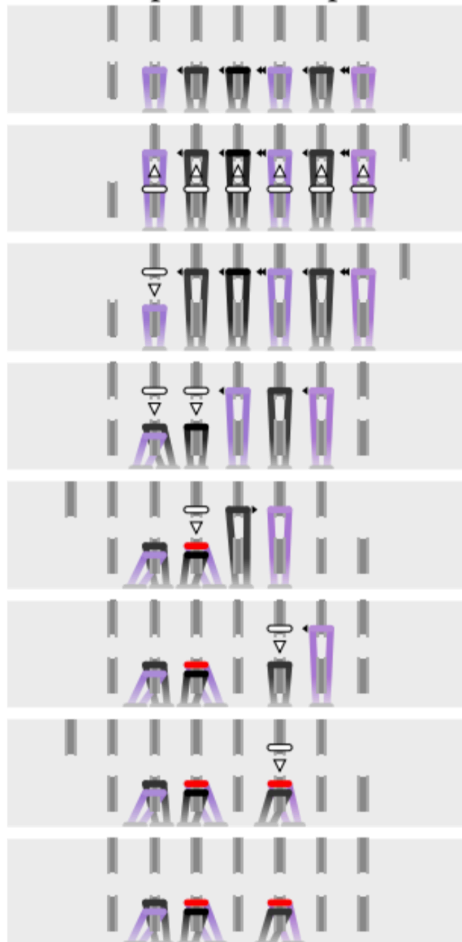
- Loop stack order changes the look! (this image is a slightly exaggerated illustration of three loops leaning stacked on one loop)

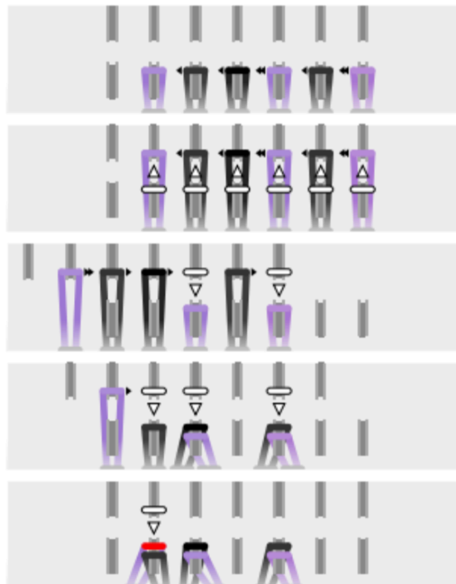


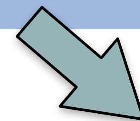
Which respects loop stacking order?

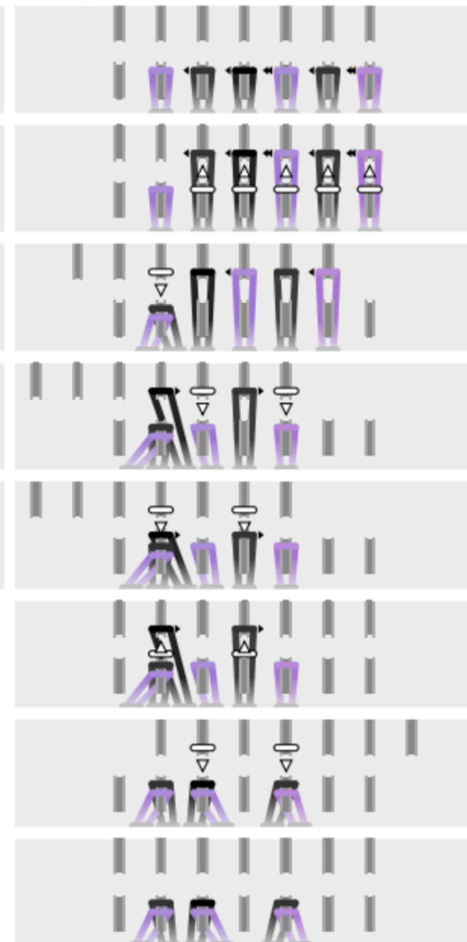




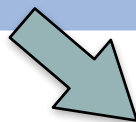
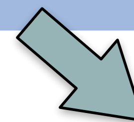
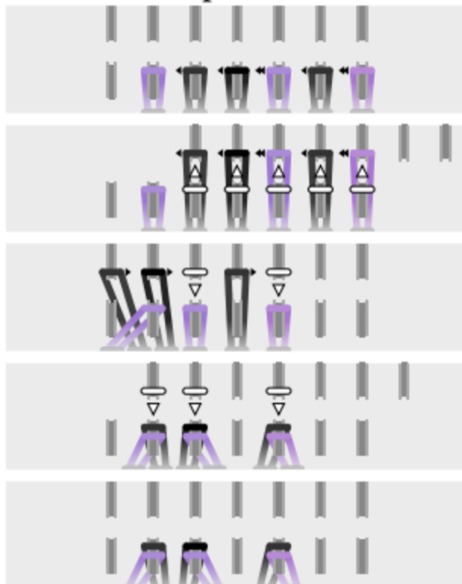


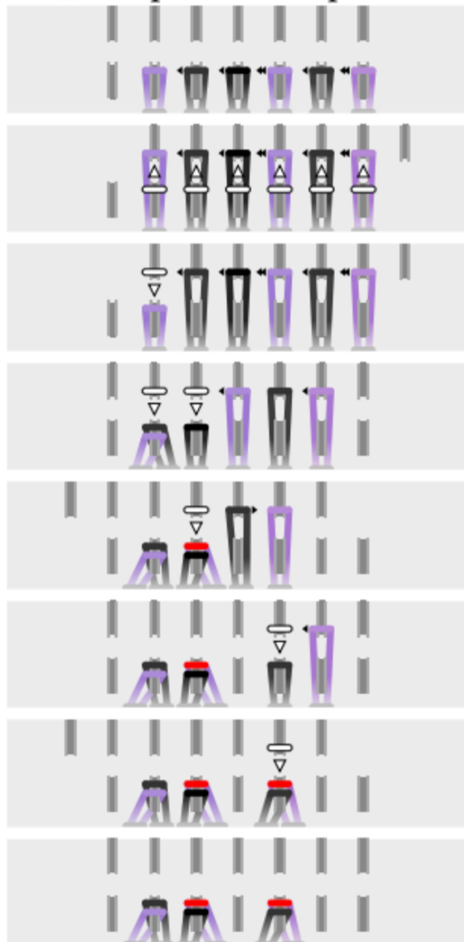


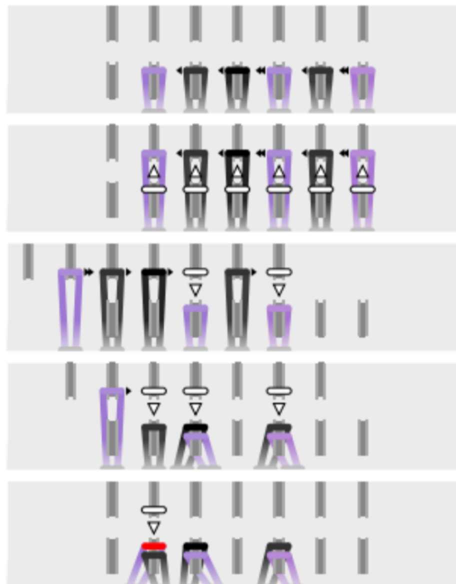


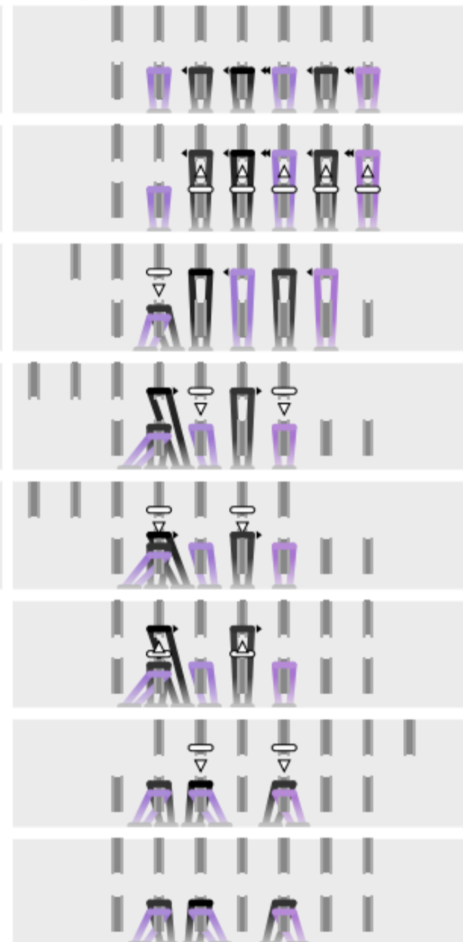


Which uses sliders?







So what happened just now was that...

- We are evaluating algorithms, or the ways to solve a specific problem
  - "correctness"
  - efficiency
  - complexity
- How do we evaluate a computer algorithm/program's efficiency? Is there an equivalent of "pass" in computer programs?

Unit Operations

- A highly unrealistic but still helpful notion is that each line of code that doesn’t not use a complex function usually does some unit operation
- Examples include
  - a + b
  - print(a)
  - grid = []

Recall: the four helper functions in HW2 knitout_helpers.py

- `write_headers`
- `cast_on`
- `knit_waste`
- `drop_all`

write_headers

```
def write_headers(knitout_lines):
    """Write standard knitout file headers."""
    knitout_lines.append(";!knitout-2")
    knitout_lines.append(";;Carriers: 1 2 3 4 5 6 7 8 9 10")
    knitout_lines.append(";;Position: Center")
    knitout_lines.append(";;Width: 450")
```

The len(knitout_lines) increase by

4 after write_headers.

cast_on

```
def cast_on(knitout_lines, carrier):
    """Perform alternating tuck cast-on on front bed."""
    for s in range(width, 0, -1):
        if (width - s) % 2 == 0:
            knitout_lines.append(f"tuck - f{s} {carrier}")
    for s in range(1, width + 1):
        if (width - s) % 2 == 1:
            knitout_lines.append(f"tuck + f{s} {carrier}")
```

The len(knitout_lines) increase by width after cast_on.

knit_waste

```
def knit_waste(knitout_lines, carrier):
    """Knit the waste rows."""
    for r in range(waste):
        for s in range(width, 0, -1):
            knitout_lines.append(f"knit - f{s} {carrier}")
        for s in range(1, width + 1):
            knitout_lines.append(f"knit + f{s} {carrier}")
```

The len(knitout_lines) increase by

waste * width * 2 after cast_on.

drop_all

```
def drop_all(knitout_lines):
    """Drop all stitches on front bed."""
    for s in range(1, width + 1):
        knitout_lines.append(f"drop f{s}")
```

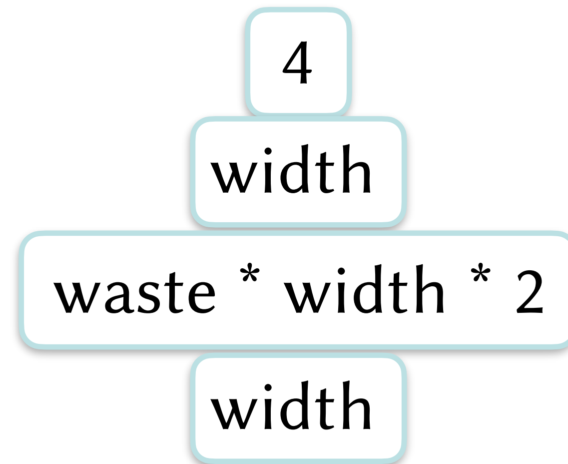
The len(knitout_lines) increase by

width after cast_on.

Recall: the four helper functions in HW2 knitout_helpers.py

After calling this function, the `len(knitout_lines)` increase by:

- `write_headers`
- `cast_on`
- `knit_waste`
- `drop_all`



We were essentially counting the number of unit operations!

Big-O Notation

- In computer science, we use a notation called the Big O to represent the complexity of some piece of code.

Function	len(knitout_lines) increase by	Big O notation
`write_headers`	4	O(1)
`cast_on`	width	O(width)
`knit_waste`	waste * width * 2	O(waste * width)
`drop_all`	width	O(width)

constants are gone!

Exercises

- Let's read some snippets and count the number of basic operations and write out the Big-O notation for these snippets' complexity.
- We'll use the snippets we saw during the review lecture.

Snippet A

```
def double(n):
    """Given n, create a list with numbers 0~(n-1) doubled."""
    result = []
    for item in range(n):
        result.append(item * 2)
    return result
```

- Number of operations
  - 2*n + 2 (or n + 2 if we treat the `result.append(item * 2)` as 1 unit operation)
- Big O notation
  - O(n)

Snippet B

```
if x > 0:
    return "positive"
elif x == 0:
    return "zero"
else:
    return "negative"
```

- Number of operations
  - 2~5 depending on the value of x
- Big O notation
  - O(1)

Snippet C

```
# grid is a 2D array defined before
for i in range(len(grid)):
    for j in range(len(grid[i])):
        grid[i][j] = 0
```

- Number of operations
  - the size of grid (len(grid) * len(grid[i]))
- Big O notation
  - O(len(grid) * len(grid[i]))

Snippet D

```
def sum_even(numbers):
    """Given a list numbers, return the sum of all the items that are
even numbers."""
    total = 0
    for num in numbers:
        if num % 2 == 0:
            total += num
    return total
```

- Number of operations
    - 2 * len(numbers) + 2
- Big O notation
    - O(len(numbers))

Snippet E

```
num = 0
grid = []
for i in range(rows):
    row = []
    for j in range(cols):
        row.append(num)
        num += 1
    grid.append(row)
```

- Number of operations
  - (2 * cols + 2) * rows + 2
- Big O notation
  - O(cols * rows)

Summary on Complexity

- You can count the number of unit operations and then figure out the Big O for that expression to get a sense of the time complexity of the program.
- For machine knitting, the concept is similar except that we are counting the number of passes.