

19. Sorting a List

Topics:

Selection Sort

Merge Sort

Our examples will
highlight the interplay between
functions and lists

Sorting a List of Numbers

Before:

x -->

50	40	10	80	20	60
----	----	----	----	----	----

After:

x -->

10	20	40	50	60	80
----	----	----	----	----	----

We Will First Implement the Method of Selection Sort

At the Start:

x -->

50	40	10	80	20	60
----	----	----	----	----	----

High-Level:

```
for k in range(len(x)-1)
    Swap x[k] with the smallest
    value in x[k:]
```

Selection Sort: How It Works

Before:

x -->

50	40	10	80	20	60
----	----	----	----	----	----

Swap `x[0]` with the smallest value in `x[0:]`

Selection Sort: How It Works

Before:

x -->

50	40	10	80	20	60
----	----	----	----	----	----

Swap `x[0]` with the smallest value in `x[0:]`

After:

x -->

10	40	50	80	20	60
----	----	----	----	----	----

Selection Sort: How It Works

Before:

x -->

10	40	50	80	20	60
----	----	----	----	----	----

Swap `x[1]` with the smallest value in `x[1:]`

Selection Sort: How It Works

Before:

x -->

10	40	50	80	20	60
----	----	----	----	----	----

Swap `x[1]` with the smallest value in `x[1:]`

After:

x -->

10	20	50	80	40	60
----	----	----	----	----	----

Selection Sort: How It Works

Before:

x -->

10	20	50	80	40	60
----	----	----	----	----	----

Swap `x[2]` with the smallest value in `x[2:]`

Selection Sort: How It Works

Before:

x -->

10	20	50	80	40	60
----	----	----	----	----	----

Swap `x[2]` with the smallest value in `x[2:]`

After:

x -->

10	20	40	80	50	60
----	----	----	----	----	----

Selection Sort: How It Works

Before:

x -->

10	20	40	80	50	60
----	----	----	----	----	----

Swap `x[3]` with the smallest value in `x[3:]`

Selection Sort: How It Works

Before:

x -->

10	20	40	80	50	60
----	----	----	----	----	----

Swap `x[3]` with the smallest value in `x[3:]`

After:

x -->

10	20	40	50	80	60
----	----	----	----	----	----

Selection Sort: How It Works

Before:

x -->

10	20	40	50	80	60
----	----	----	----	----	----

Swap `x[4]` with the smallest value in `x[4:]`

Selection Sort: How It Works

Before:

x -->

10	20	40	50	80	60
----	----	----	----	----	----

Swap `x[4]` with the smallest value in `x[4:]`

After:

x -->

10	20	40	50	60	80
----	----	----	----	----	----

Selection Sort: Recap

50	40	10	80	20	60
----	----	----	----	----	----

10	40	50	80	20	60
----	----	----	----	----	----

10	20	50	80	40	60
----	----	----	----	----	----

10	20	40	80	50	60
----	----	----	----	----	----

10	20	40	50	80	60
----	----	----	----	----	----

10	20	40	50	60	80
----	----	----	----	----	----

10	20	40	50	60	80
----	----	----	----	----	----

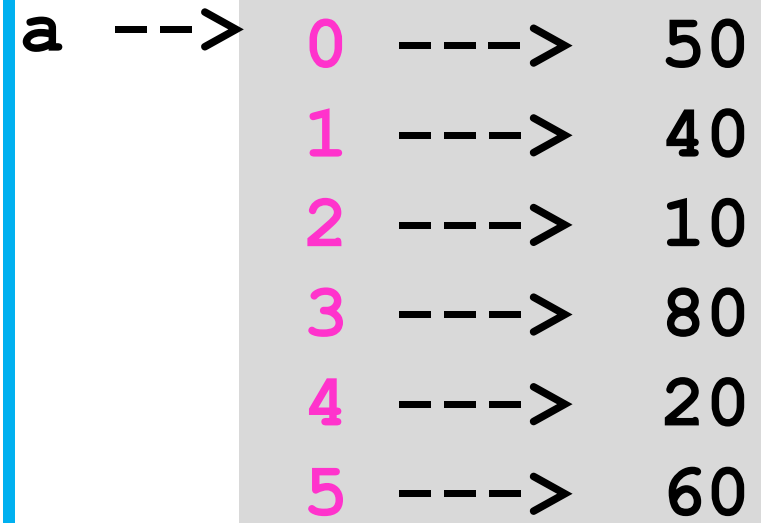
The Essential Helper Function: `Select(x,i)`

```
def Select(x,i):  
    """ Swaps the smallest value in  
    x[i:] with x[i]  
  
    PreC: x is a list of integers and  
    i is an in that satisfies  
    0<=i<len(x) """
```

Does not return anything and it has a list argument

How Does it Work?

The calling program has a list. E.g.,



```
a --> 0 ----> 50  
        1 ----> 40  
        2 ----> 10  
        3 ----> 80  
        4 ----> 20  
        5 ----> 60
```

Index	Value
0	50
1	40
2	10
3	80
4	20
5	60

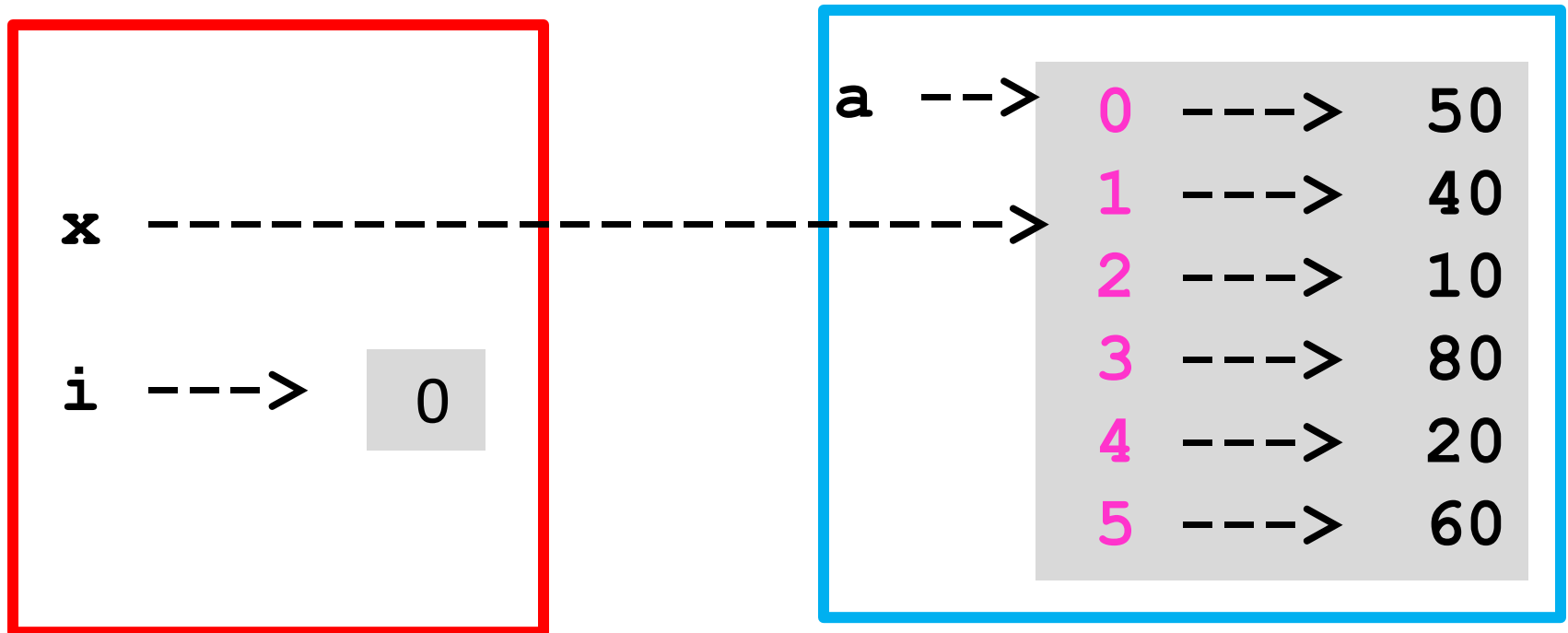
How Does it Work?

The calling program executes `Select(a, 0)`
and control passes to `Select`

a -->	0	---->	50
	1	---->	40
	2	---->	10
	3	---->	80
	4	---->	20
	5	---->	60

How Does Select Work?

- Nothing new about the assignment of 0 to i.
- But there is no assignment of the list a to x.
- Instead x now **refers** to the same list as a.



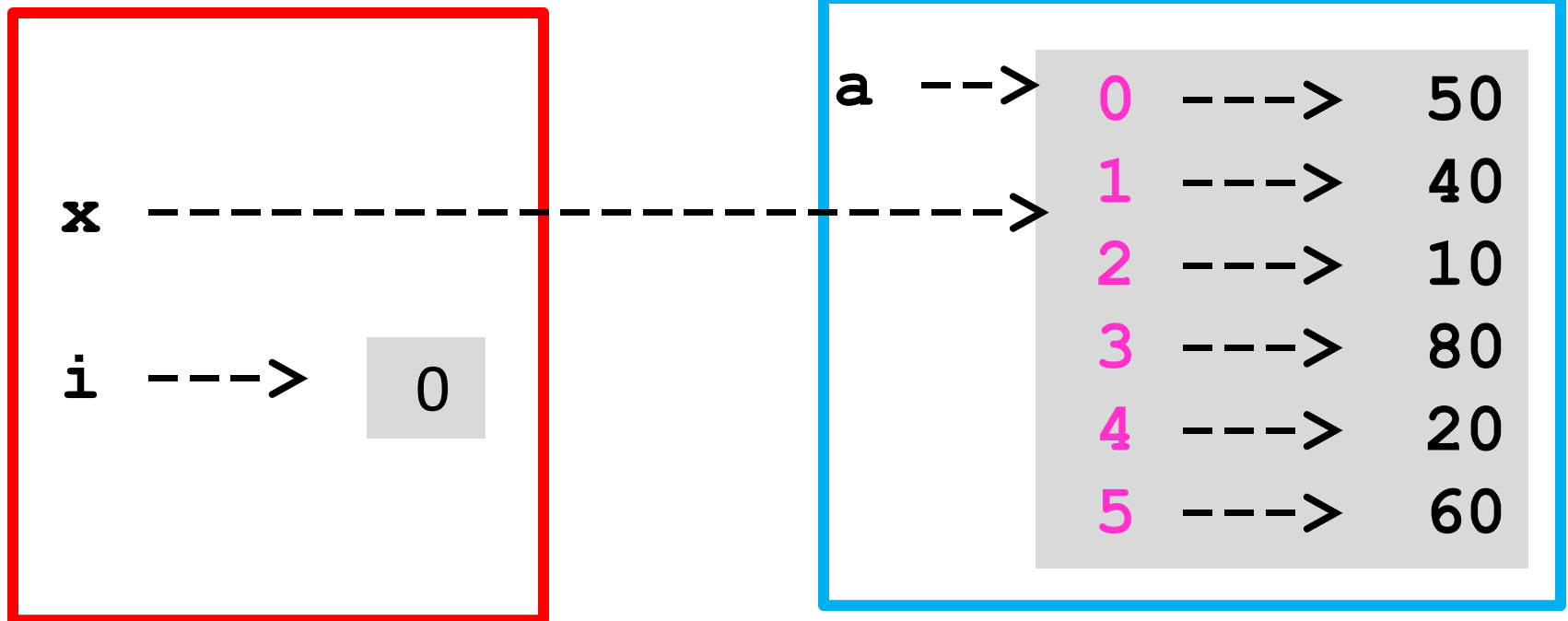
How Does Select Work?

If inside Select we have

$t = x[0]; x[0] = x[2]; x[2] = t$

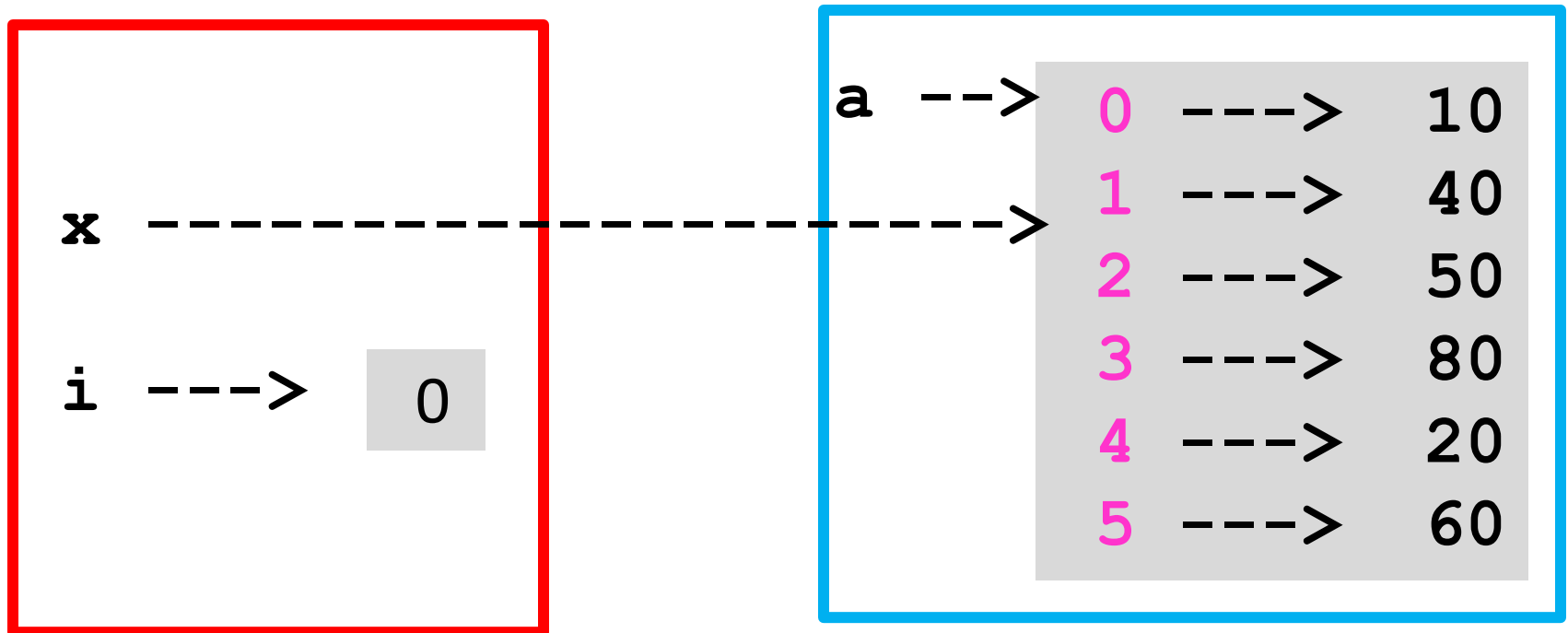
it is as if we said

$t = a[0]; a[0] = a[2]; a[2] = t$



How Does Select Work?

It changes the list `a` in the calling program.
We say `x` and `a` are aliased. They refer to the same list



Let's Assume This Is Implemented

```
def Select(x,i):  
    """ Swaps the smallest value in  
    x[i:] with x[i]  
  
    PreC: x is a list of integers and  
    i is an in that satisfies  
    0<=i<len(x) """
```

After this:

The list a looks like this

Initialization

50	40	10	80	20	60
----	----	----	----	----	----

Select (a, 0)

10	40	50	80	20	60
----	----	----	----	----	----

Select (a, 1)

10	20	50	80	40	60
----	----	----	----	----	----

Select (a, 2)

10	20	40	80	50	60
----	----	----	----	----	----

Select (a, 3)

10	20	40	50	80	60
----	----	----	----	----	----

Select (a, 4)

10	20	40	50	60	80
----	----	----	----	----	----

Select (a, 5)

10	20	40	50	60	80
----	----	----	----	----	----

In General We Have This

```
def SelectionSort(a):  
    n = len(a)  
    for k in range(n):  
        Select(a,k)
```


Next Problem

Merging Two Sorted Lists
into a
Single Sorted List

Example

x ->

12	33	35	45
----	----	----	----

y ->

15	42	55	65	75
----	----	----	----	----

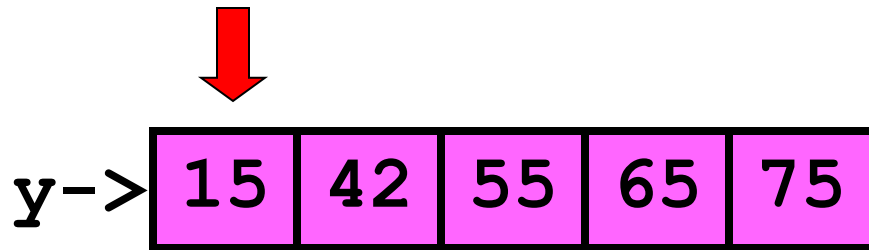
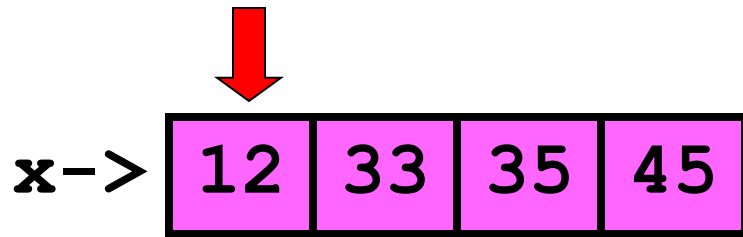
x and y are input
They are sorted

z is the output

z ->

12	15	33	35	42	45	55	65	75
----	----	----	----	----	----	----	----	----

Merging Two Sorted Lists



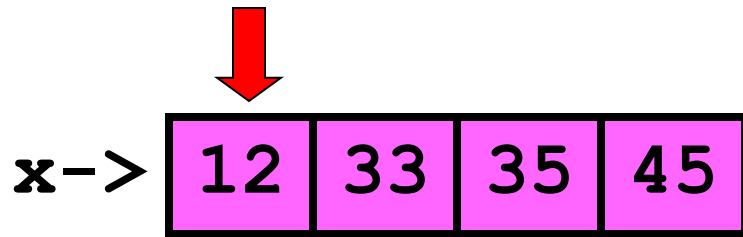
z -> []

ix and iy
keep track
of where
we are in x
and y

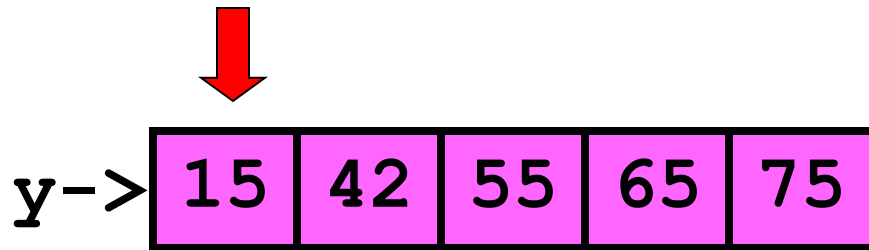
ix: [0]

iy: [0]

Merging Two Sorted Lists



ix: [0]



iy: [0]

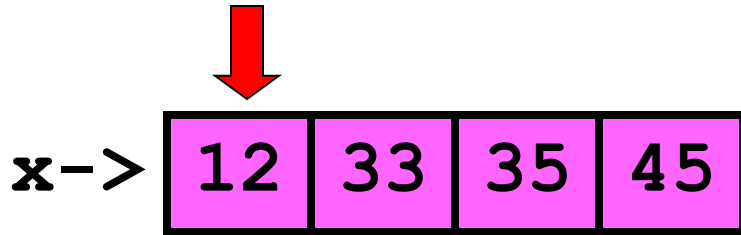
z -> []


Do we pick from x ?

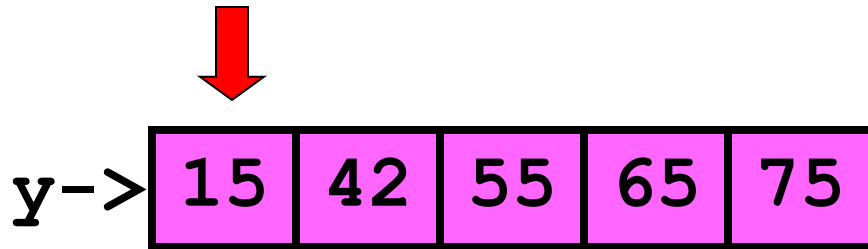
x[**ix**] <= **y**[**iy**]


???

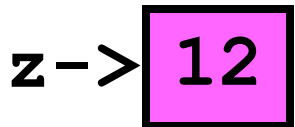
Merge



ix: 

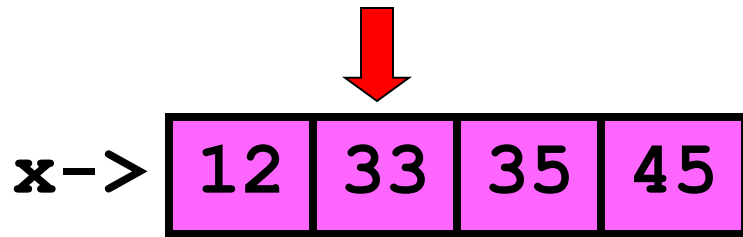


iy: 



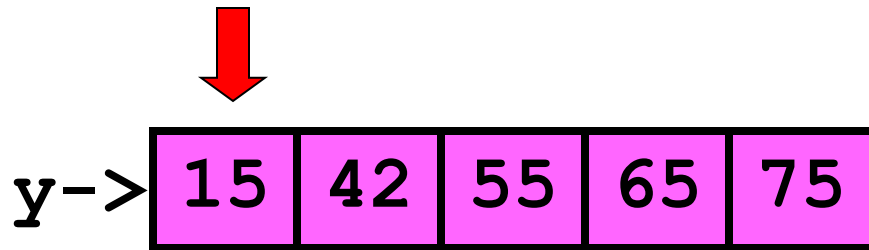
Yes. So update ix

Merge



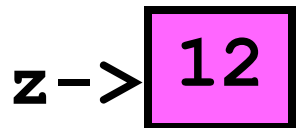
ix:

1



iy:

0

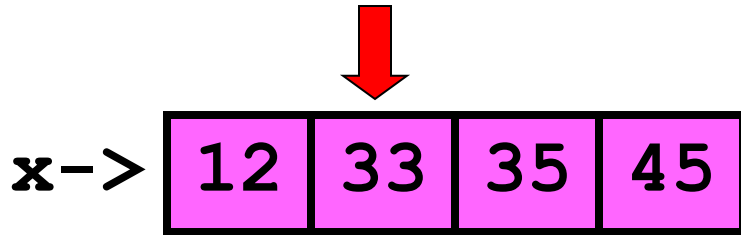


Do we pick from x ?

$x[ix] \leq y[iy]$

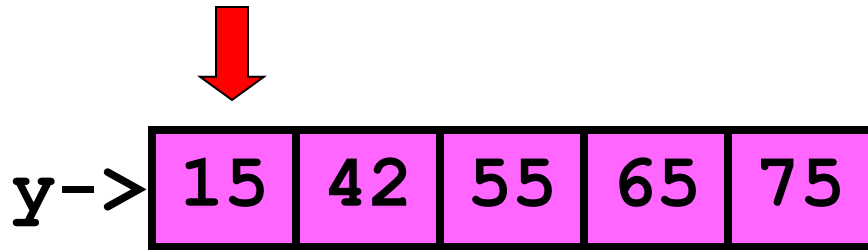
???

Merge



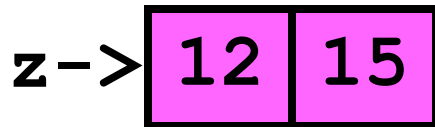
ix:

1



iy:

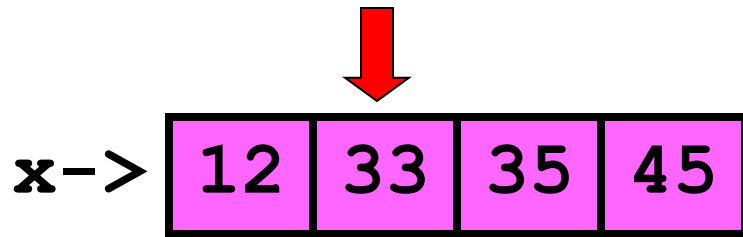
0



iz:

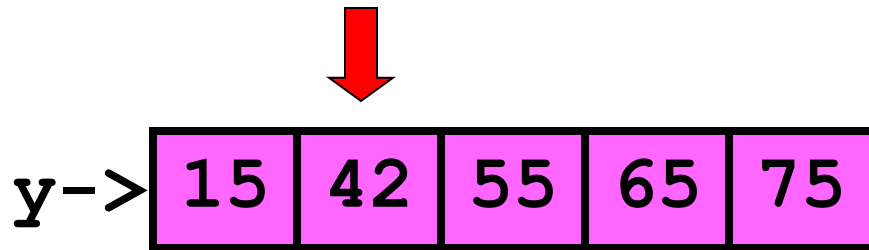
No. So update iy

Merge



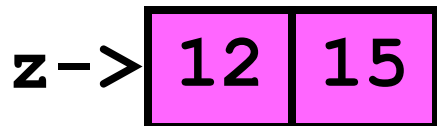
ix:

1



iy:

1

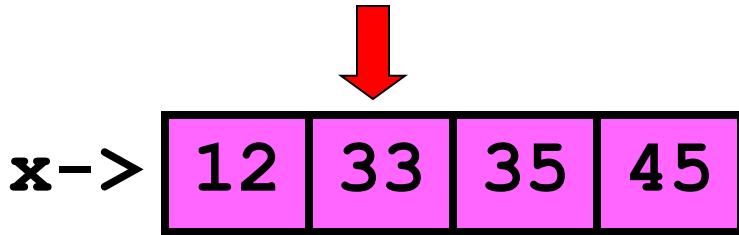


Do we pick from x ?

$x[ix] \leq y[iy]$

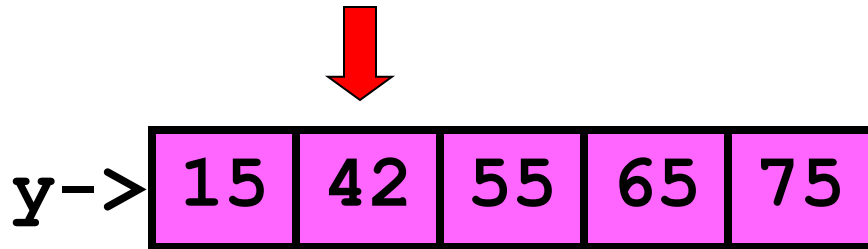
???

Merge



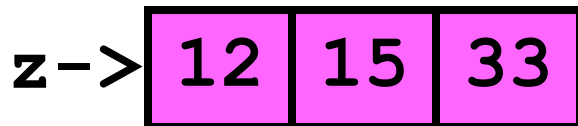
ix:

1



iy:

1



Yes. So update ix

Merge



x->

12	33	35	45
----	----	----	----

ix:

2



y->

15	42	55	65	75
----	----	----	----	----

iy:

1

z->

12	15	33
----	----	----

Do we pick from x ?

$x[ix] \leq y[iy]$

???

Merge



x->

12	33	35	45
----	----	----	----

ix:

2



y->

15	42	55	65	75
----	----	----	----	----

iy:

1

z->

12	15	33	35
----	----	----	----

Yes. So update ix

Merge



x->

12	33	35	45
----	----	----	----

ix:

3



y->

15	42	55	65	75
----	----	----	----	----

iy:

1

z->

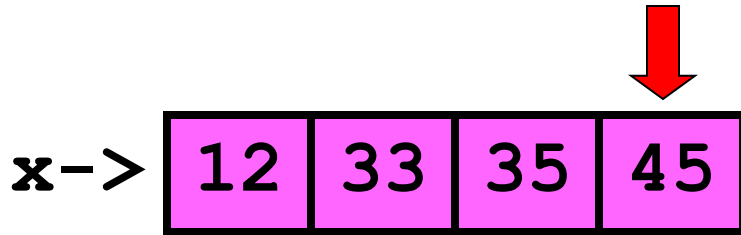
12	15	33	35
----	----	----	----

Do we pick from x ?

$x[ix] \leq y[iy]$

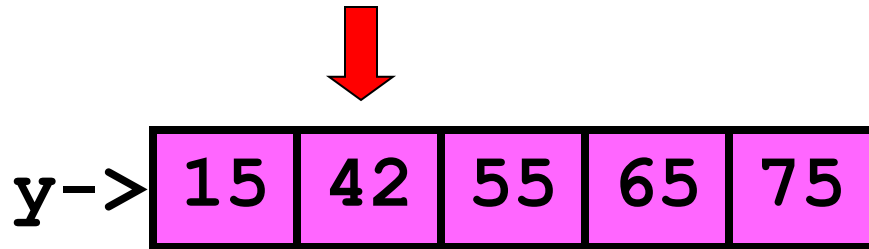
???

Merge



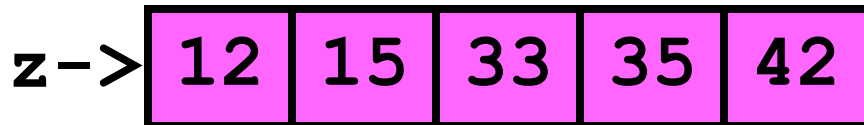
ix:

3



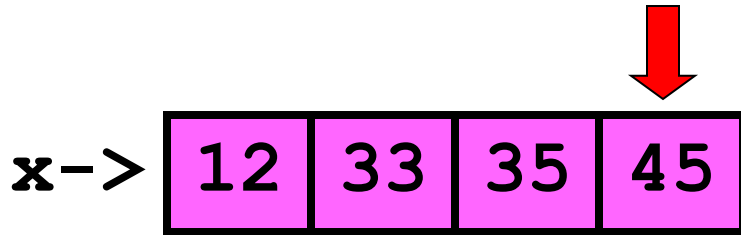
iy:

1



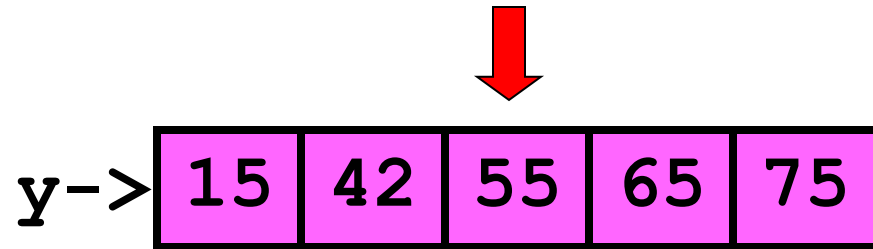
No. So update iy...

Merge



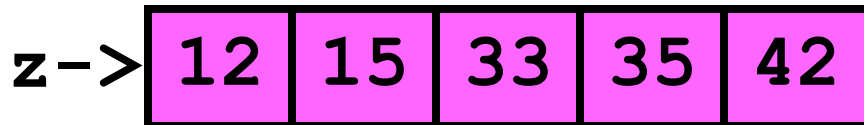
ix:

3



iy:

2

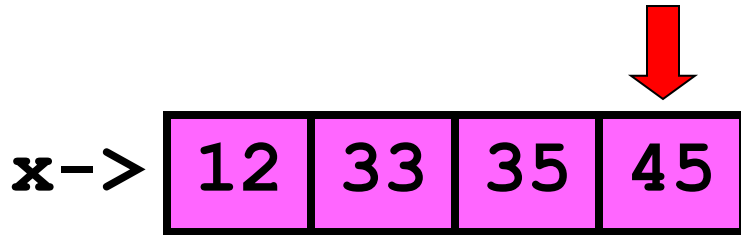


Do we pick from x ?

$x[ix] \leq y[iy]$

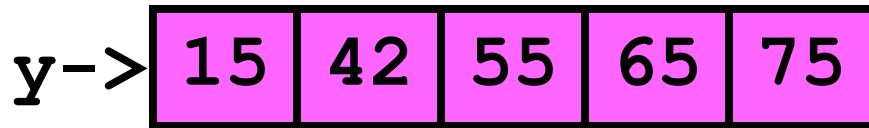
???

Merge



ix:

3



iy:

2



Yes. So update ix.

Merge



x->

12	33	35	45
----	----	----	----

ix:

4



y->

15	42	55	65	75
----	----	----	----	----

iy:

2

z->

12	15	33	35	42	45
----	----	----	----	----	----

Done with x. Pick from y

Merge



x->

12	33	35	45
----	----	----	----

ix:

4



y->

15	42	55	65	75
----	----	----	----	----

iy:

2

z->

12	15	33	35	42	45	55
----	----	----	----	----	----	----

So update iy

Merge



x->

12	33	35	45
----	----	----	----

ix:

4



y->

15	42	55	65	75
----	----	----	----	----

iy:

3

z->

12	15	33	35	42	45	55
----	----	----	----	----	----	----

Done with x. Pick from y

Merge



x->

12	33	35	45
----	----	----	----

ix:

4



y->

15	42	55	65	75
----	----	----	----	----

iy:

3

z->

12	15	33	35	42	45	55	65
----	----	----	----	----	----	----	----

So update iy.

Merge



x->

12	33	35	45
----	----	----	----

ix:

4



y->

15	42	55	65	75
----	----	----	----	----

iy:

4

z->

12	15	33	35	42	45	55	65
----	----	----	----	----	----	----	----

Done with x. Pick from y

Merge



x->

12	33	35	45
----	----	----	----

ix:

4



y->

15	42	55	65	75
----	----	----	----	----

iy:

4

z->

12	15	33	35	42	45	55	65	75
----	----	----	----	----	----	----	----	----

Update iy

Merge



x->

12	33	35	45
----	----	----	----

ix:

4



y->

15	42	55	65	75
----	----	----	----	----

iy:

5

z->

12	15	33	35	42	45	55	65	75
----	----	----	----	----	----	----	----	----

All Done

The Python Implementation...

```

def Merge(x,y):
    n = len(x); m = len(y);
    ix = 0; iy = 0; z = []
    for iz in range(n+m):
        if ix >= n:
            z.append(y[iy]); iy+=1
        elif iy >= m:
            z.append(x[ix]); ix+=1
        elif x[ix] <= y[iy]:
            z.append(x[ix]); ix+=1
        elif x[ix] > y[iy]:
            z.append(y[iy]); iy+=1
    return z

```

Build z up
via repeated
appending

x-list exhausted y-list exhausted x-value smaller y-value smaller


```

def Merge(x,y):
    n = len(x); m = len(y);
    ix = 0; iy = 0; z = []
    for iz in range(n+m):
        if ix>=n:
            z.append(y[iy]); iy+=1
        elif iy>=m:
            z.append(x[ix]); ix+=1
        elif x[ix] <= y[iy]:
            z.append(x[ix]); ix+=1
        elif x[ix] > y[iy]:
            z.append(y[iy]); iy+=1
    return z

```

len(x)+len(y)
is the total length
of the merged list

x-list exhausted y-list exhausted x-value smaller y-value smaller

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Make copies of the
Incoming lists

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Build z up via
repeated appending

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Every “pop” reduces the length by 1. The loop shuts down when one of u or v is exhausted

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

g gets the popped value
and it is appended to z

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Add what is left in u.
OK if u is the empty list

Implementation Using Pop

```
def Merge(x,y):  
    u = list(x)  
    v = list(y)  
    z = []  
    while len(u)>0 and len(v)>0 :  
        if u[0]<= v[0]:  
            g = u.pop(0)  
        else:  
            g = v.pop(0)  
        z.append(g)  
    z.extend(u)  
    z.extend(v)  
    return z
```

Add what is left in v.
OK if v is the empty list

MergeSort

Binary Search is an example of a “divide and conquer” approach to problem solving.

A method for sorting a list that features this strategy is MergeSort

Motivation

You are asked to sort a list but you have two "helpers": H1 and H2.

Idea:

1. Split the list in half and have each helper sort one of the halves.
2. Then merge the two sorted lists into a single larger list.

This idea can be repeated if H1 has two helpers and H2 has two helpers.

Subdivide the Sorting Task

H	E	M	G	B	K	A	Q	F	L	P	D	R	C	J	N
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

H	E	M	G	B	K	A	Q
---	---	---	---	---	---	---	---

F	L	P	D	R	C	J	N
---	---	---	---	---	---	---	---

Subdivide Again



H	E	M	G	B	K	A	Q
---	---	---	---	---	---	---	---

F	L	P	D	R	C	J	N
---	---	---	---	---	---	---	---

H	E	M	G
---	---	---	---

B	K	A	Q
---	---	---	---

F	L	P	D
---	---	---	---

R	C	J	N
---	---	---	---

And Again

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

H	E	M	G
---	---	---	---

B	K	A	Q
---	---	---	---

F	L	P	D
---	---	---	---

R	C	J	N
---	---	---	---

H	E
---	---

M	G
---	---

B	K
---	---

A	Q
---	---

F	L
---	---

P	D
---	---

R	C
---	---

J	N
---	---

And One Last Time

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

--	--	--	--	--	--	--	--

--	--	--	--

--	--	--	--

--	--	--	--

--	--	--	--

--	--

--	--

--	--

--	--

--	--

--	--

--	--

--	--

H	E	M	G	B	K	A	Q	F	L	P	D	R	C	J	N
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Now Merge



E H

G M

B K

A Q

F L

D P

C R

J N

H E

M G

B K

A Q

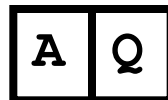
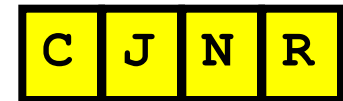
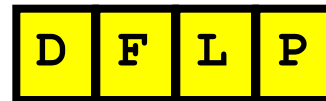
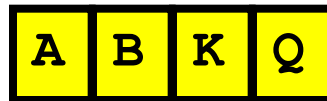
F L

P D

R C

J N

And Merge Again



And Again

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

A	B	E	G	H	K	M	Q
---	---	---	---	---	---	---	---

C	D	F	J	L	N	P	R
---	---	---	---	---	---	---	---

E	G	H	M
---	---	---	---

A	B	K	Q
---	---	---	---

D	F	L	P
---	---	---	---

C	J	N	R
---	---	---	---

And One Last Time

A	B	C	D	E	F	G	H	J	K	L	M	N	P	Q	R
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	E	G	H	K	M	Q
---	---	---	---	---	---	---	---

C	D	F	J	L	N	P	R
---	---	---	---	---	---	---	---

Done!

A	B	C	D	E	F	G	H	J	K	L	M	N	P	Q	R
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

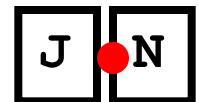
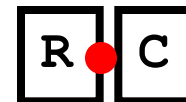
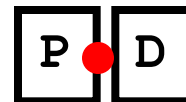
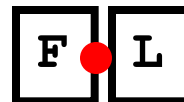
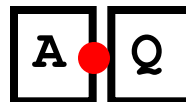
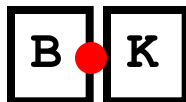
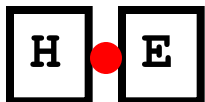
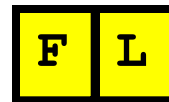
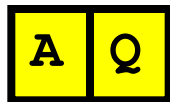
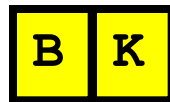
Done!

A	B	C	D	E	F	G	H	J	K	L	M	N	P	Q	R
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Let's write a function to do this making use of

```
def Merge(x,y):  
    """ Returns a float list that is the  
    merge of sorted lists x and y.  
  
    PreC: x and y are lists of floats  
    that are sorted from small to big.  
    """
```

8 Merges Producing length-2 lists



Handcoding the $n = 16$ case

A0 = Merge(a[0], a[1])

A1 = Merge(a[2], a[3])

A2 = Merge(a[4], a[5])

A3 = Merge(a[6], a[7])

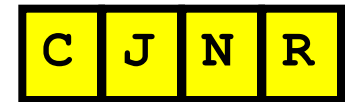
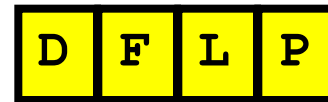
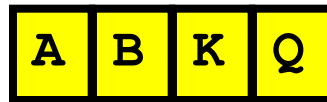
A4 = Merge(a[8], a[9])

A5 = Merge(a[10], a[11])

A6 = Merge(a[12], a[13])

A7 = Merge(a[14], a[15])

4 Merges Producing Length-4 lists



Handcoding the $n = 16$ case

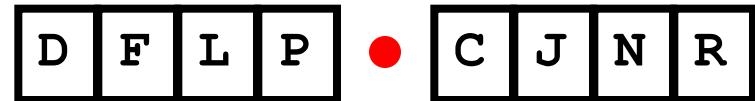
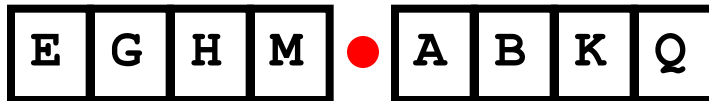
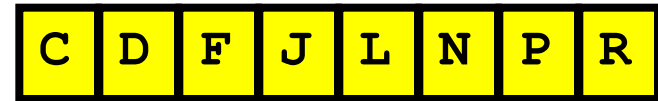
B0 = Merge (A0 ,A1)

B1 = Merge (A2 ,A3)

B2 = Merge (A4 ,A5)

B3 = Merge (A6 ,A7)

2 Merges Producing Length-8 Lists



Handcoding the $n = 16$ case

C0 = Merge (B0 , B1)

C1 = Merge (B2 , B3)

1 Merge Producing a Length-16 List

A	B	C	D	E	F	G	H	J	K	L	M	N	P	Q	R
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	E	G	H	K	M	Q
---	---	---	---	---	---	---	---



C	D	F	J	L	N	P	R
---	---	---	---	---	---	---	---

All Done!

`D0 = Merge (C0 , C1)`

For general n , it can be handled using recursion.

Recursive Merge Sort

```
def MergeSort(a):  
    n = length(a)  
    if n==1:  
        return a  
    else:  
        m = n/2  
        u0 = list(a[:m])  
        u1 = list(a[m:])  
        y0 = MergeSort(u0)  
        y1 = MergeSort(u1)  
        return Merge(y0,y1)
```

A function
can call
itself!

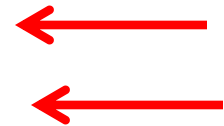


Back To Merge Sort

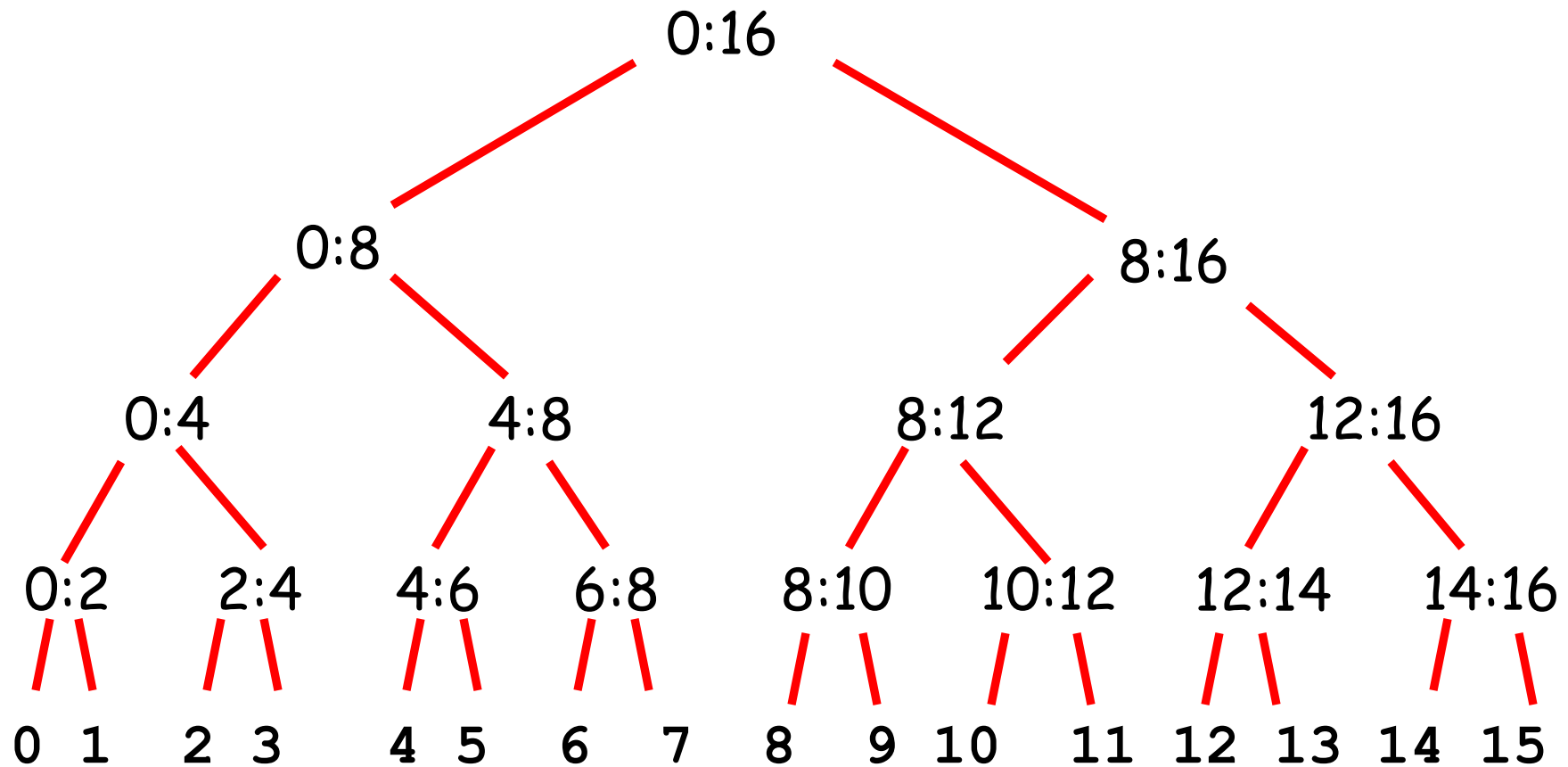
Recursive Merge Sort

```
def MergeSort(a):  
    n = length(a)  
    if n==1:  
        return a  
    else:  
        m = n/2  
        u0 = list(a[:m])  
        u1 = list(a[m:])  
        y0 = MergeSort(u0)  
        y1 = MergeSort(u1)  
        return Merge(y0,y1)
```

A function
can call
itself!

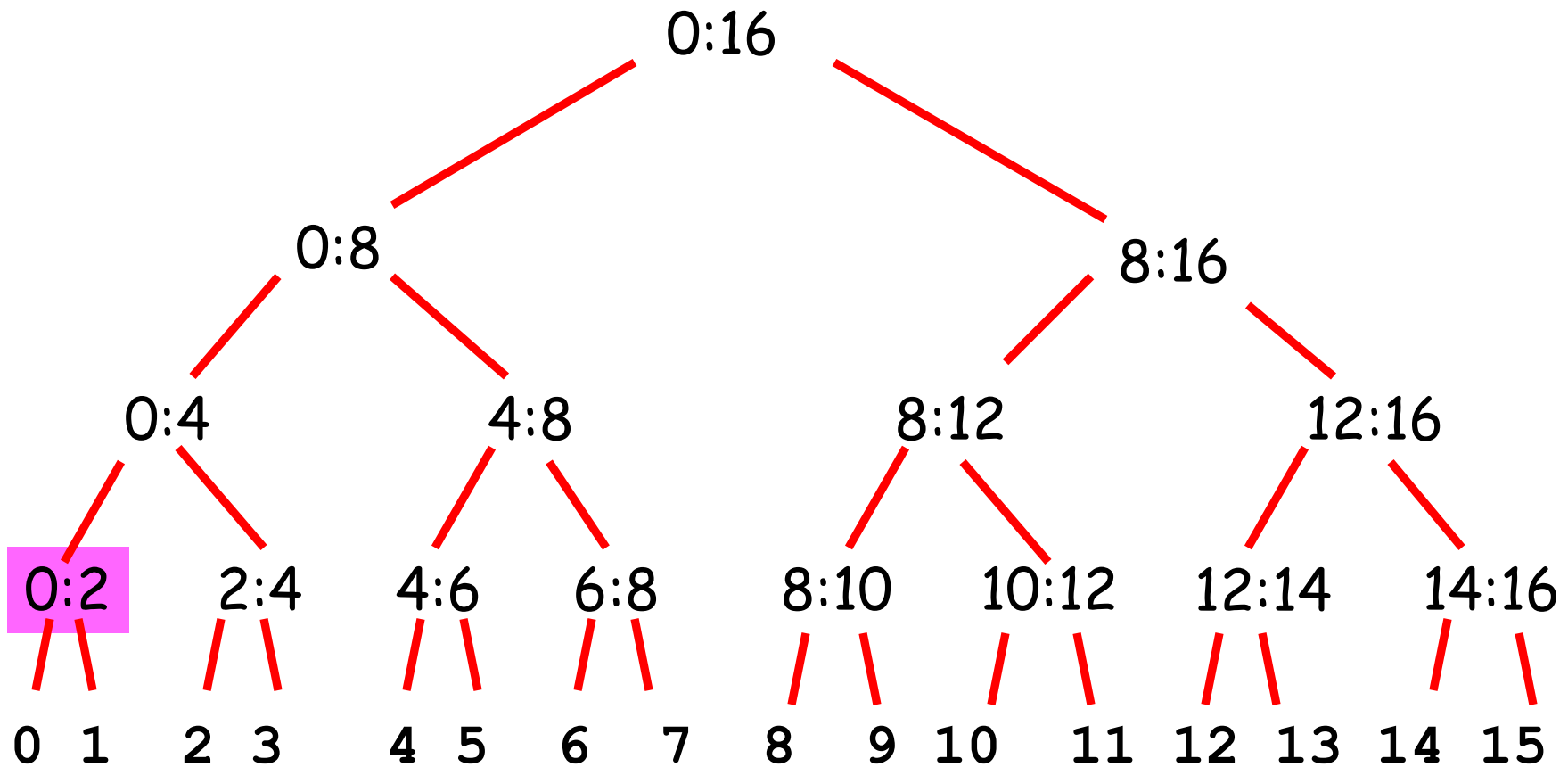


A Schematic



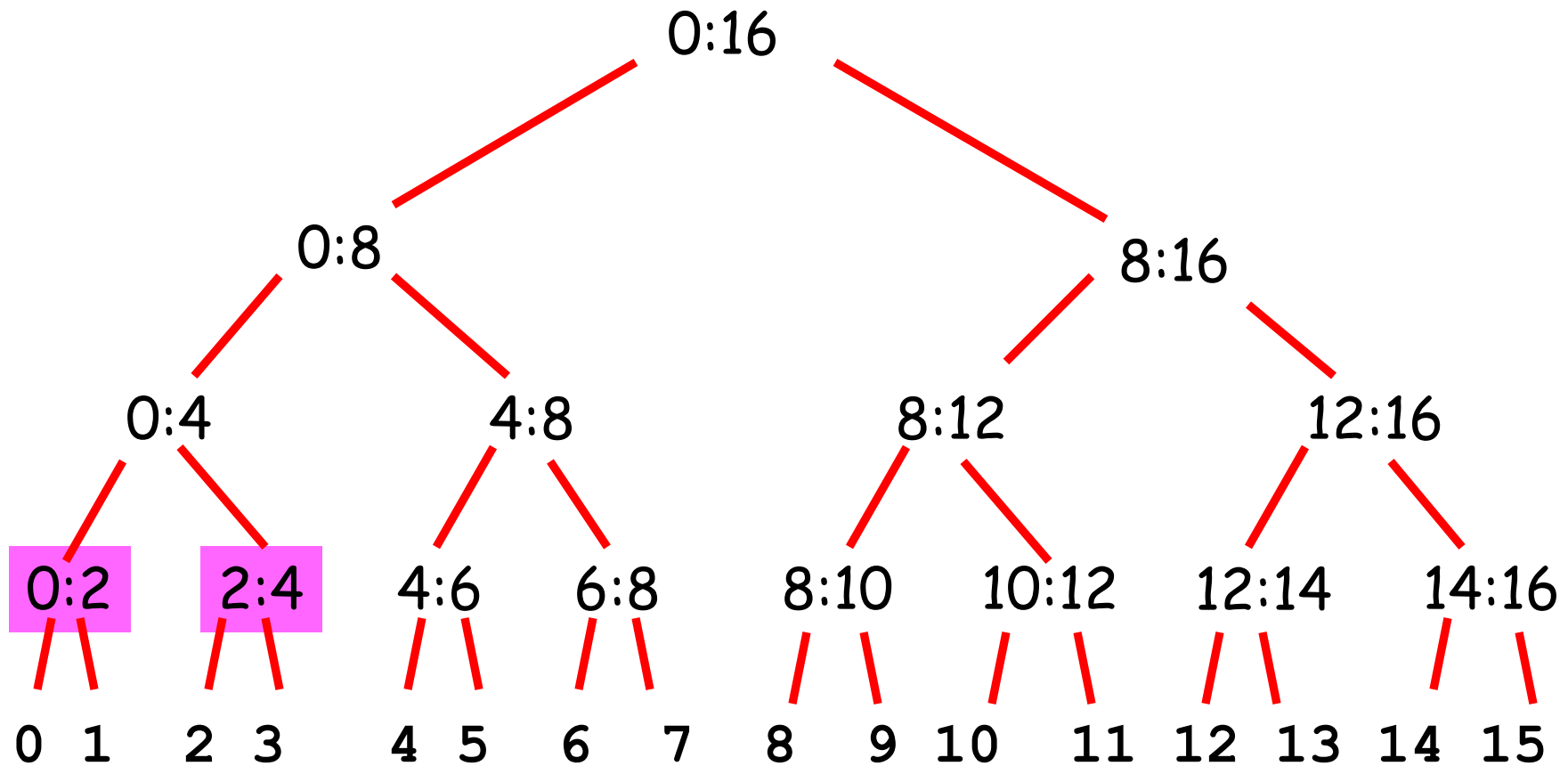
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted

A Schematic



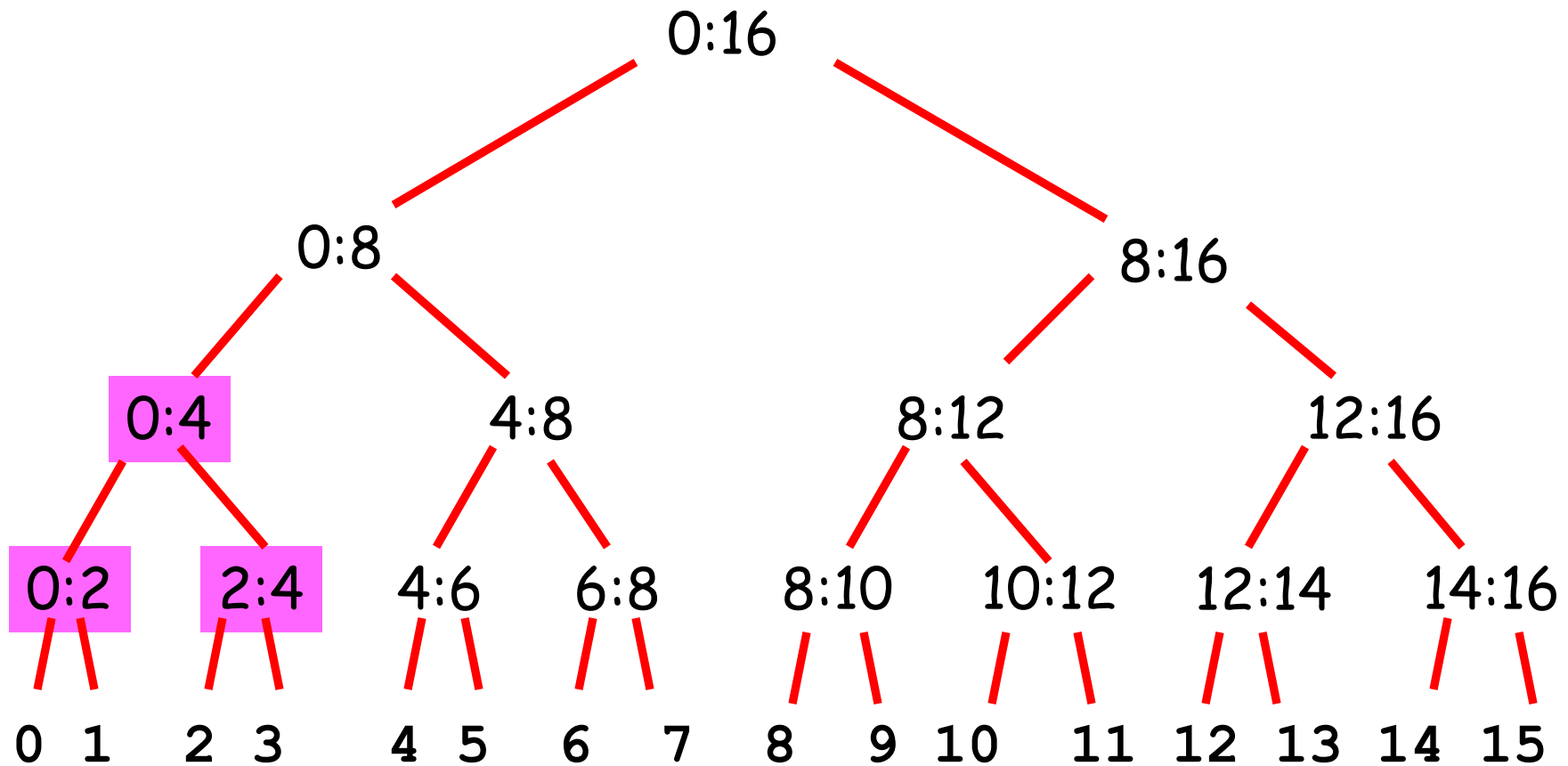
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



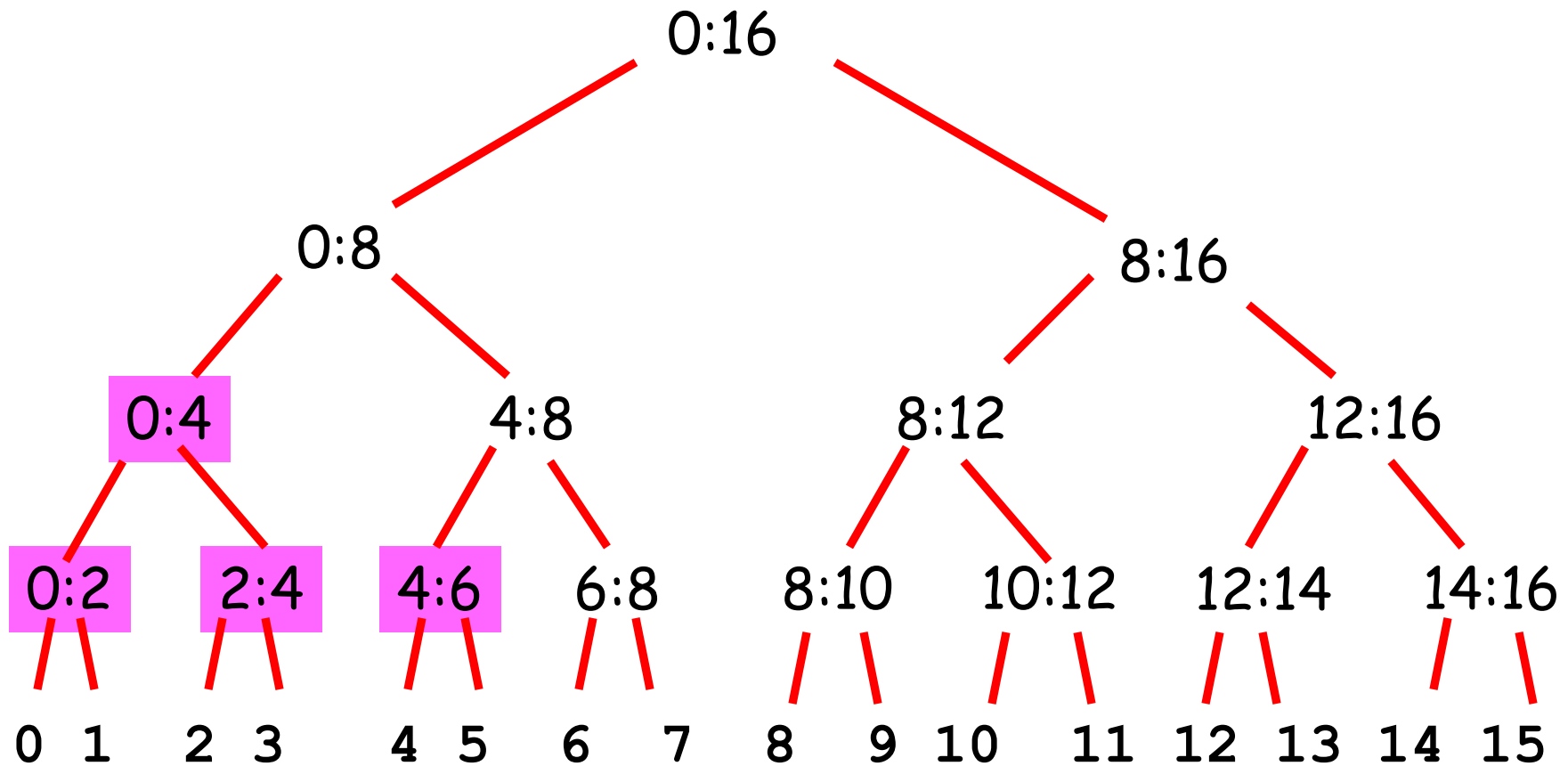
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



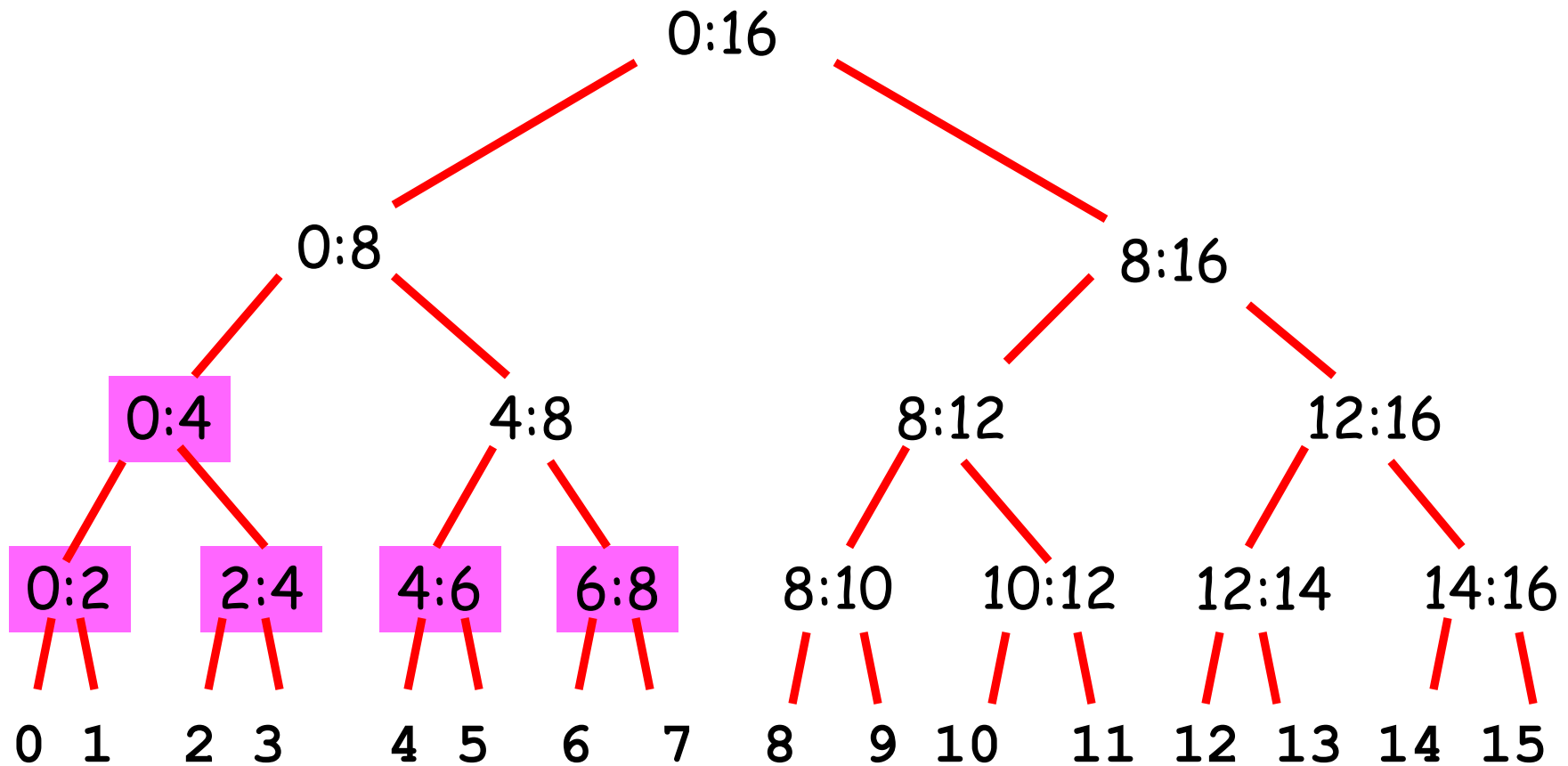
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



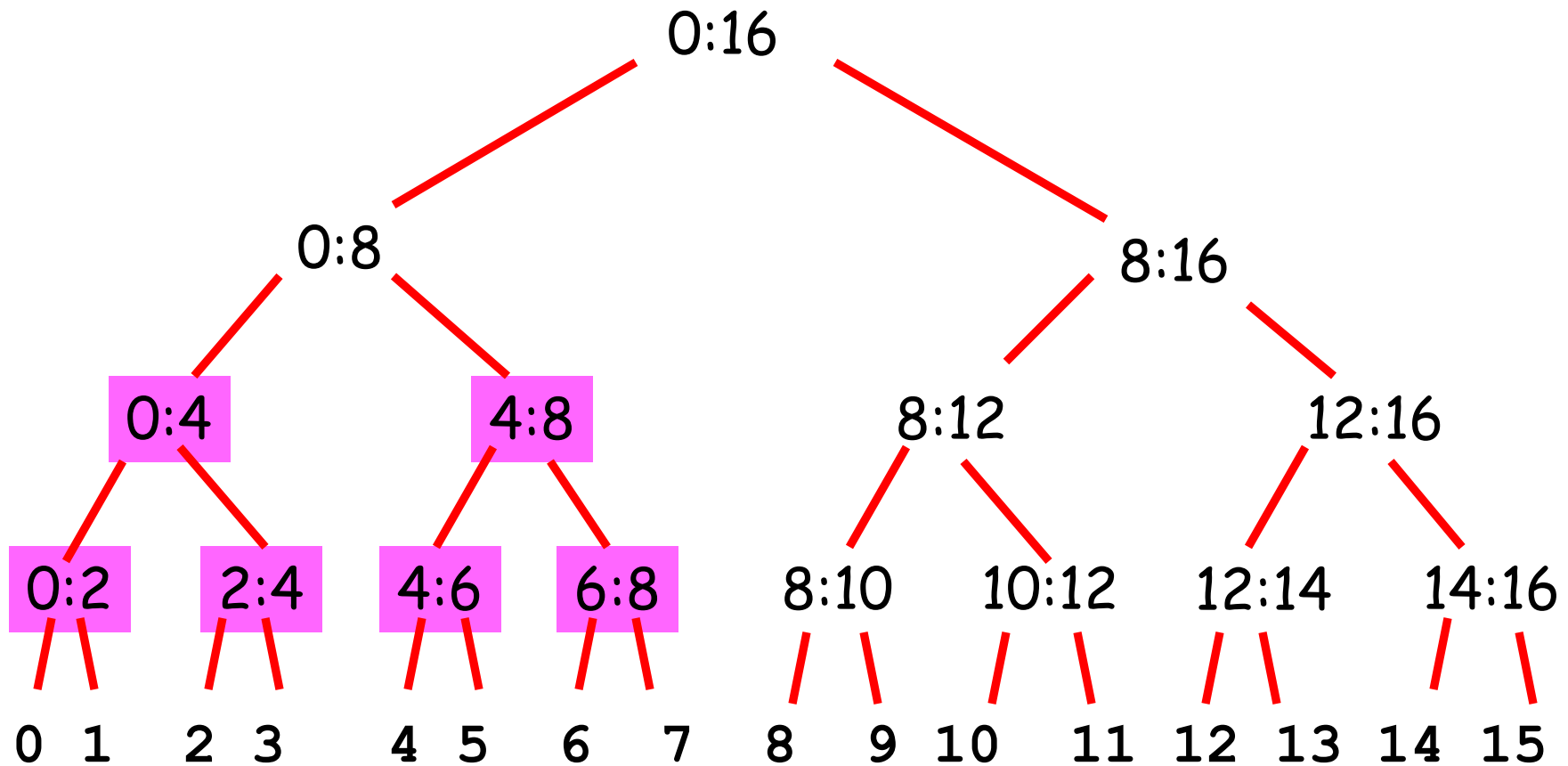
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



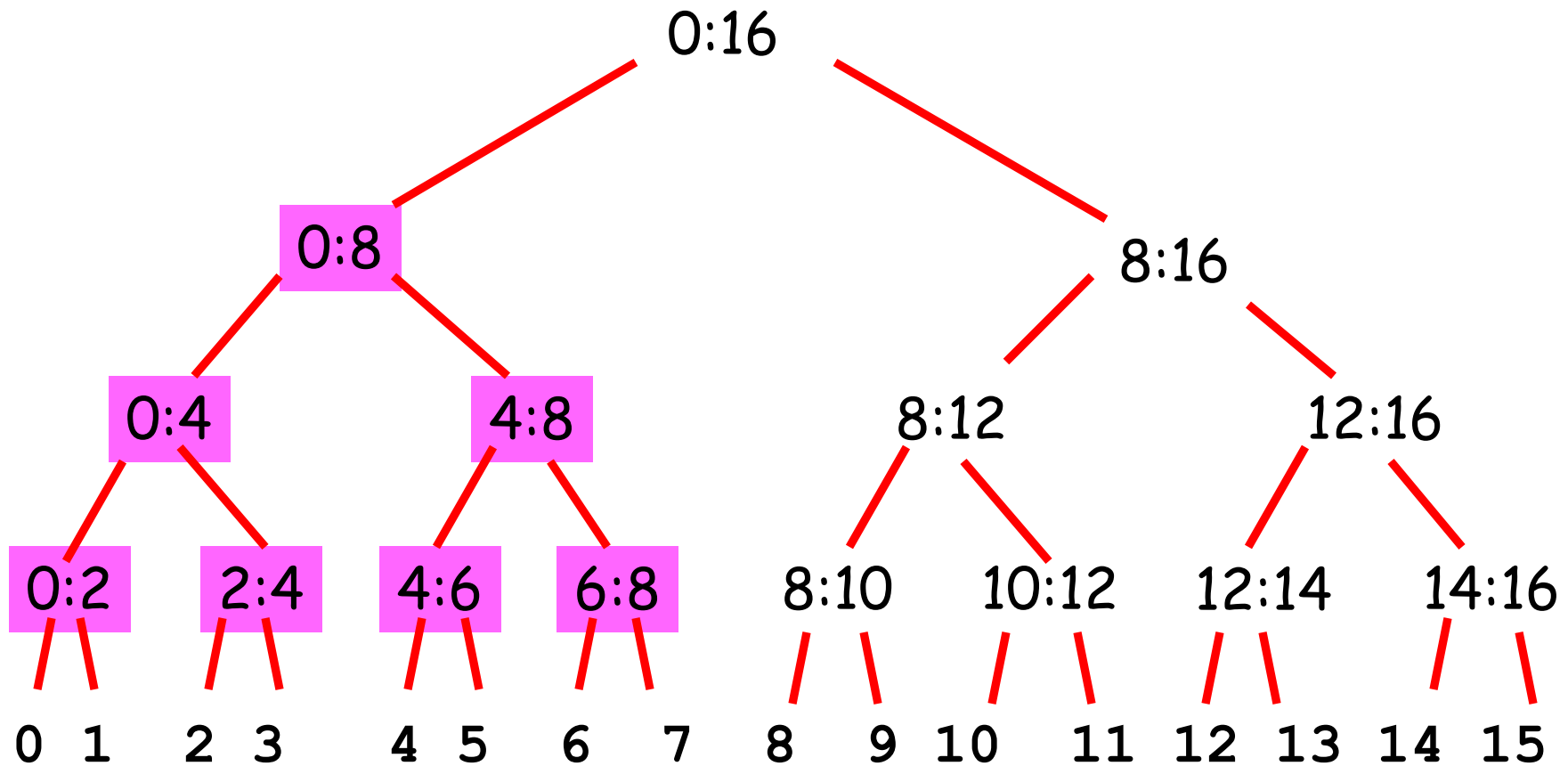
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



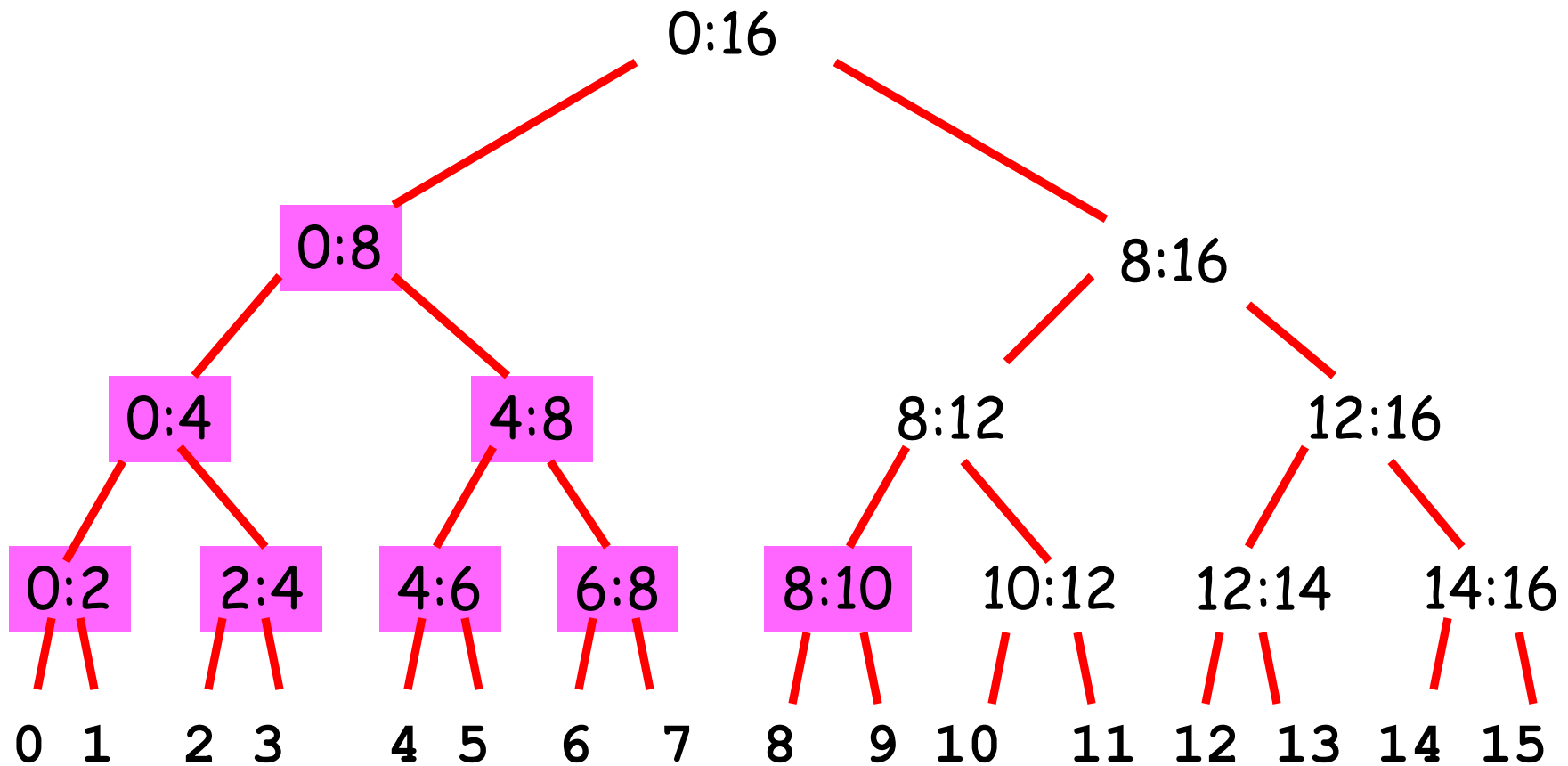
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



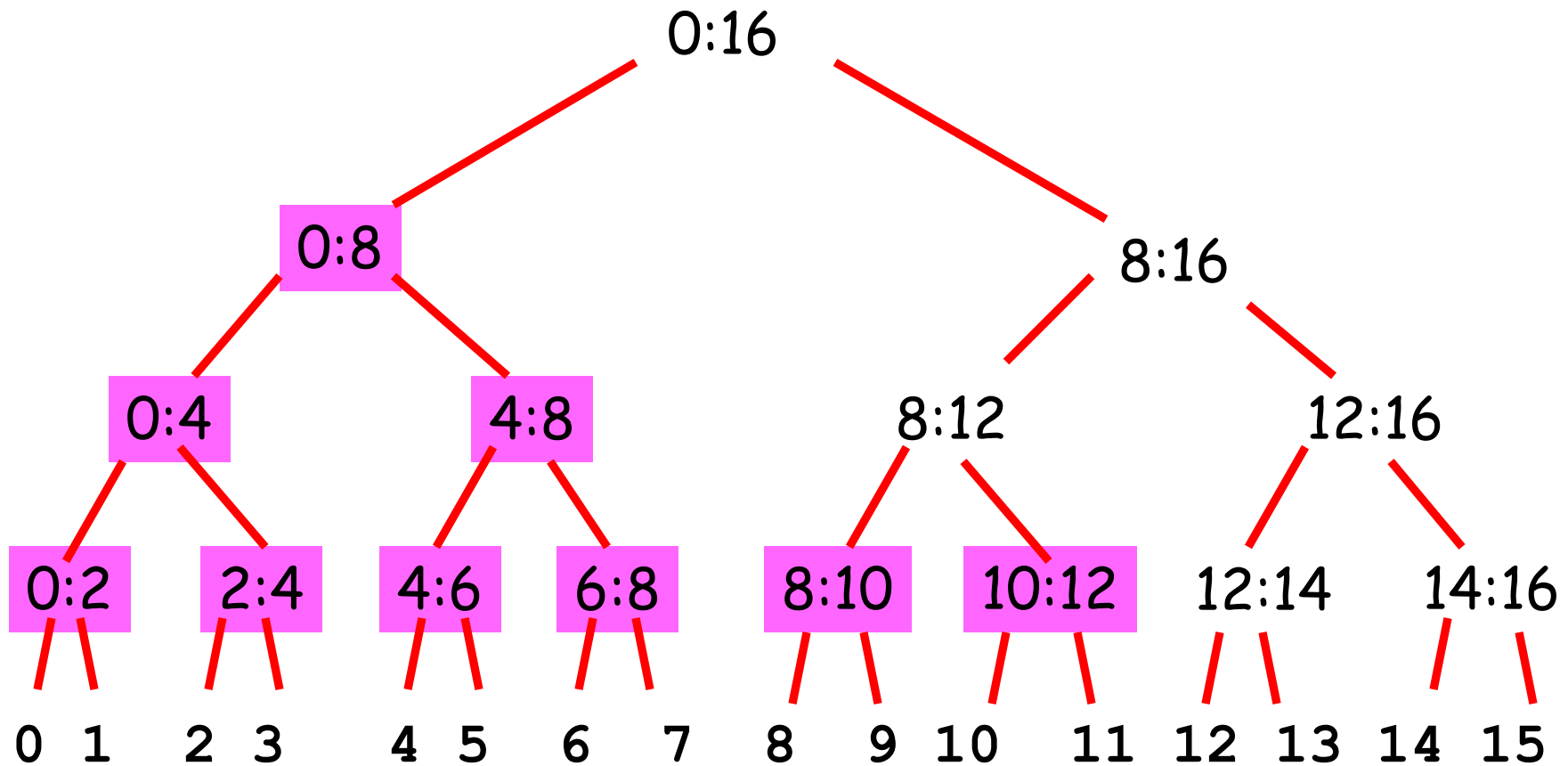
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



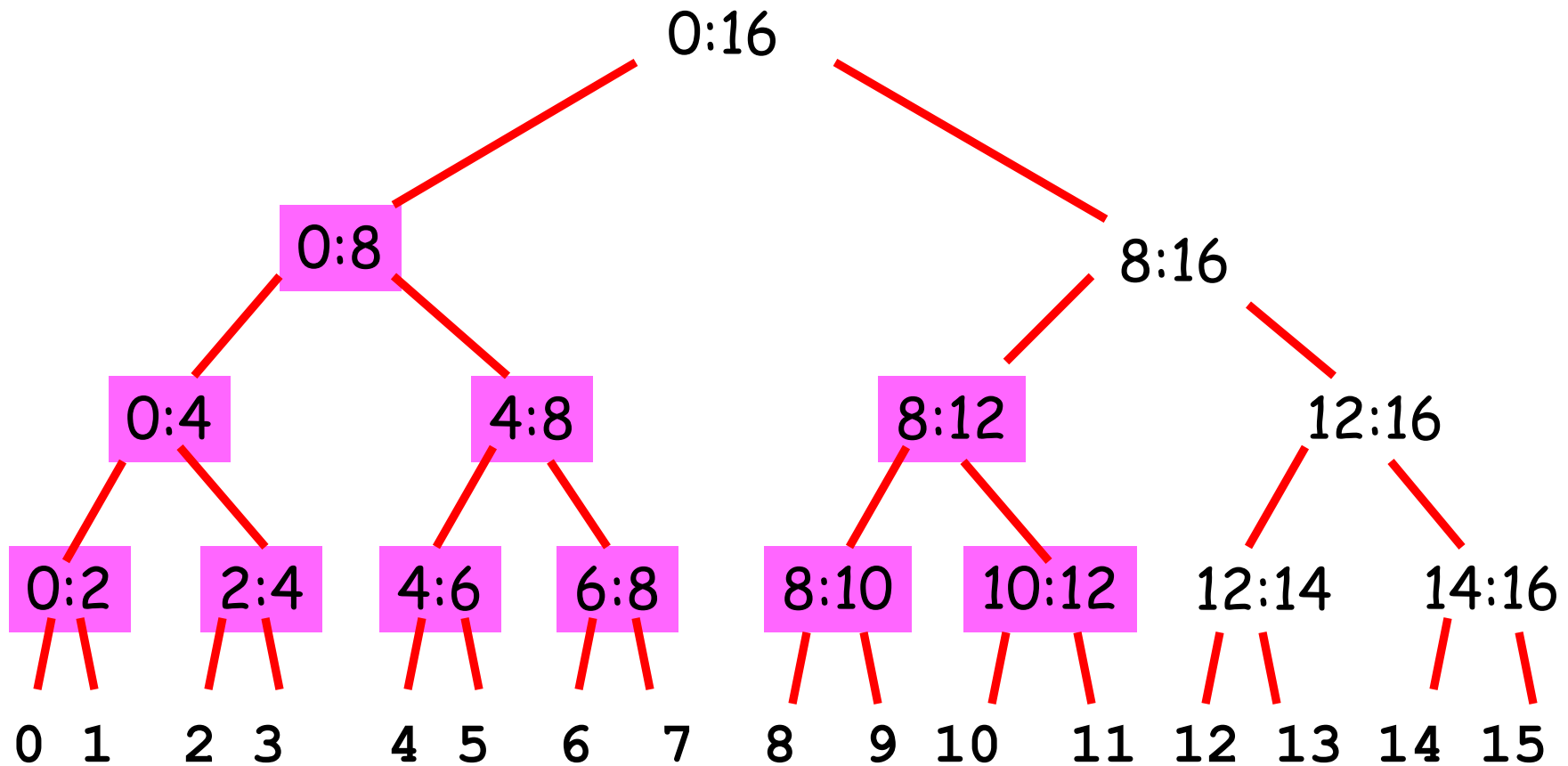
A Sorted List is produced at each ":" Let's look at the order in which

A Schematic



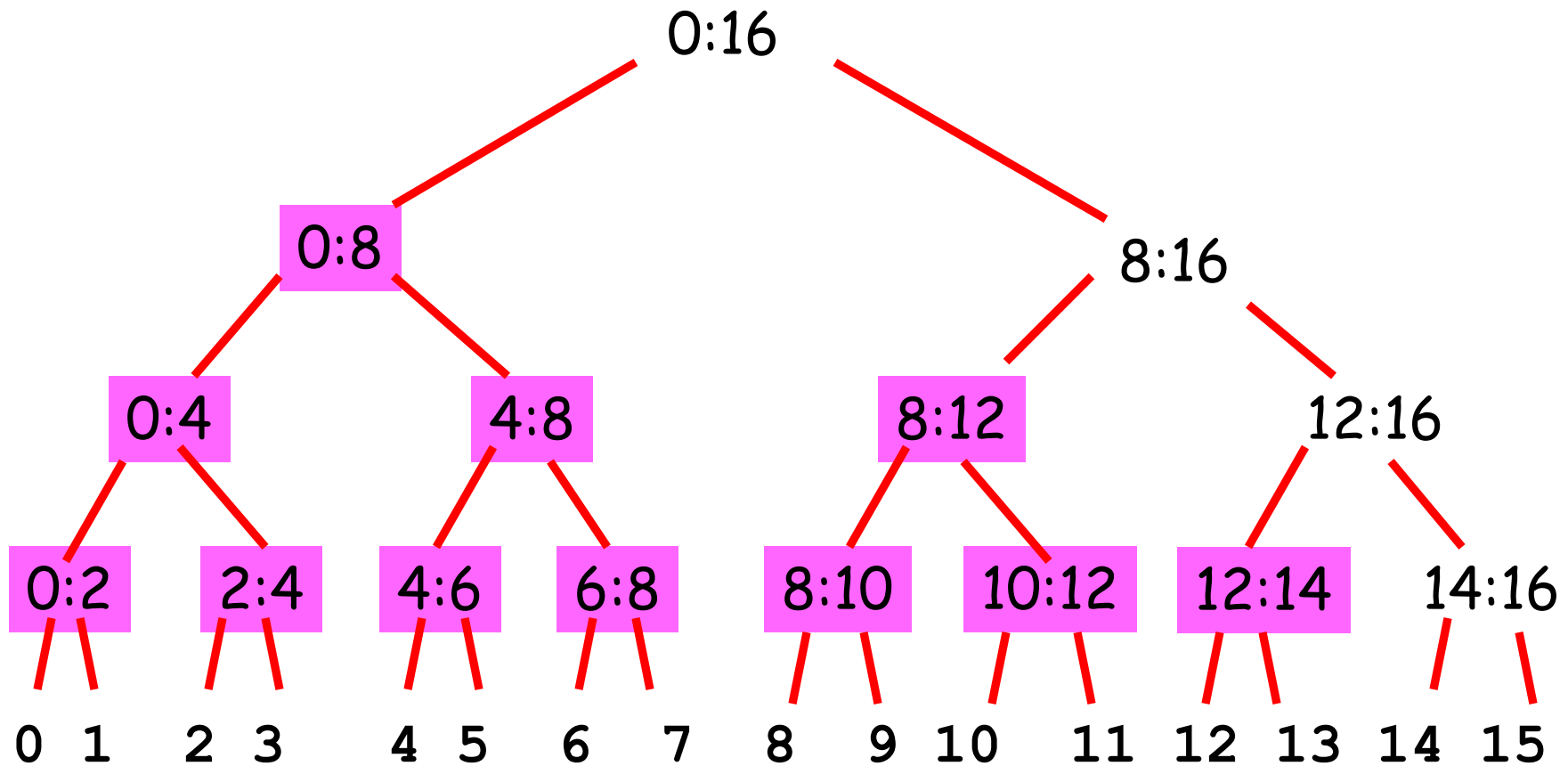
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



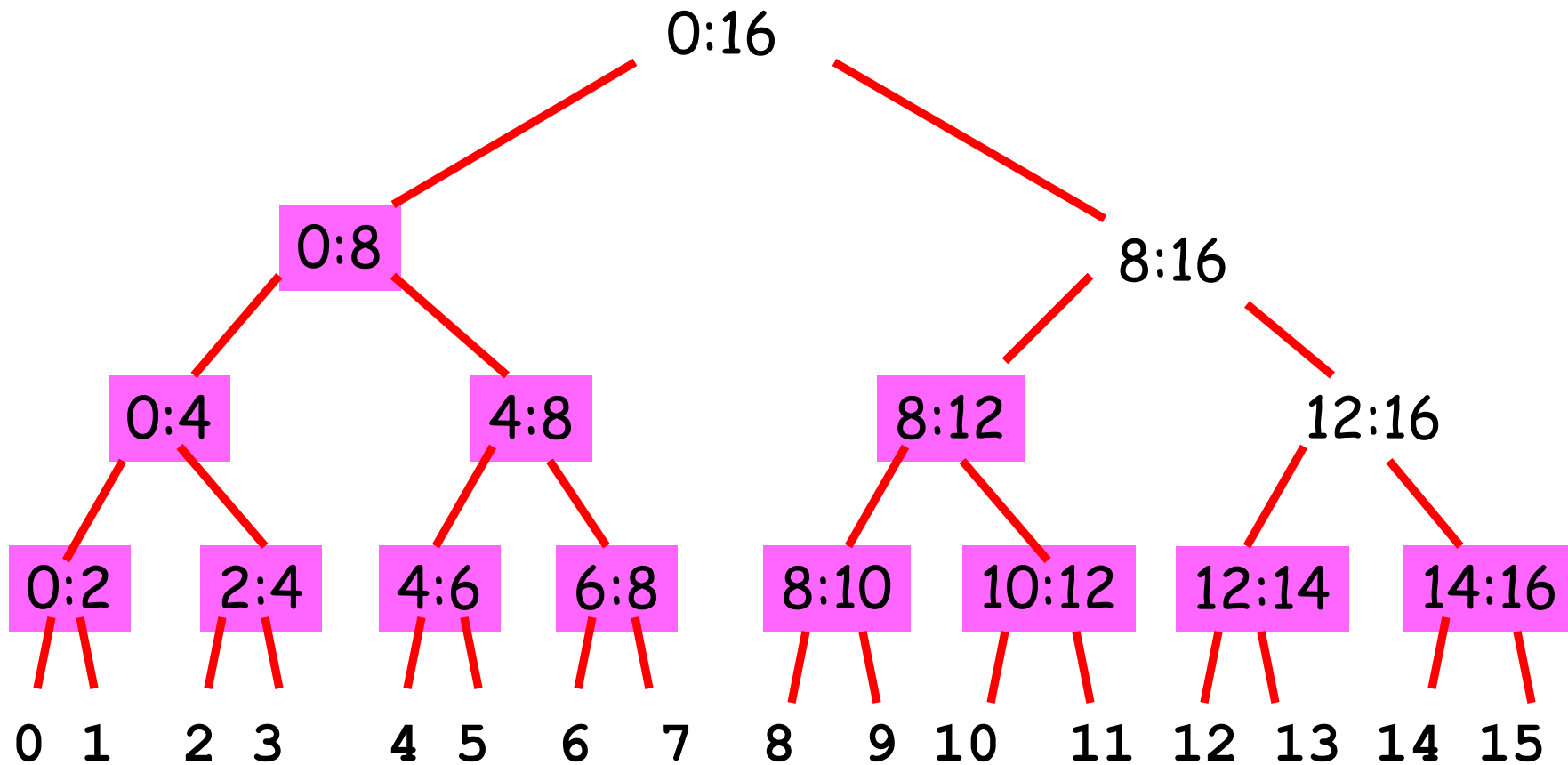
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



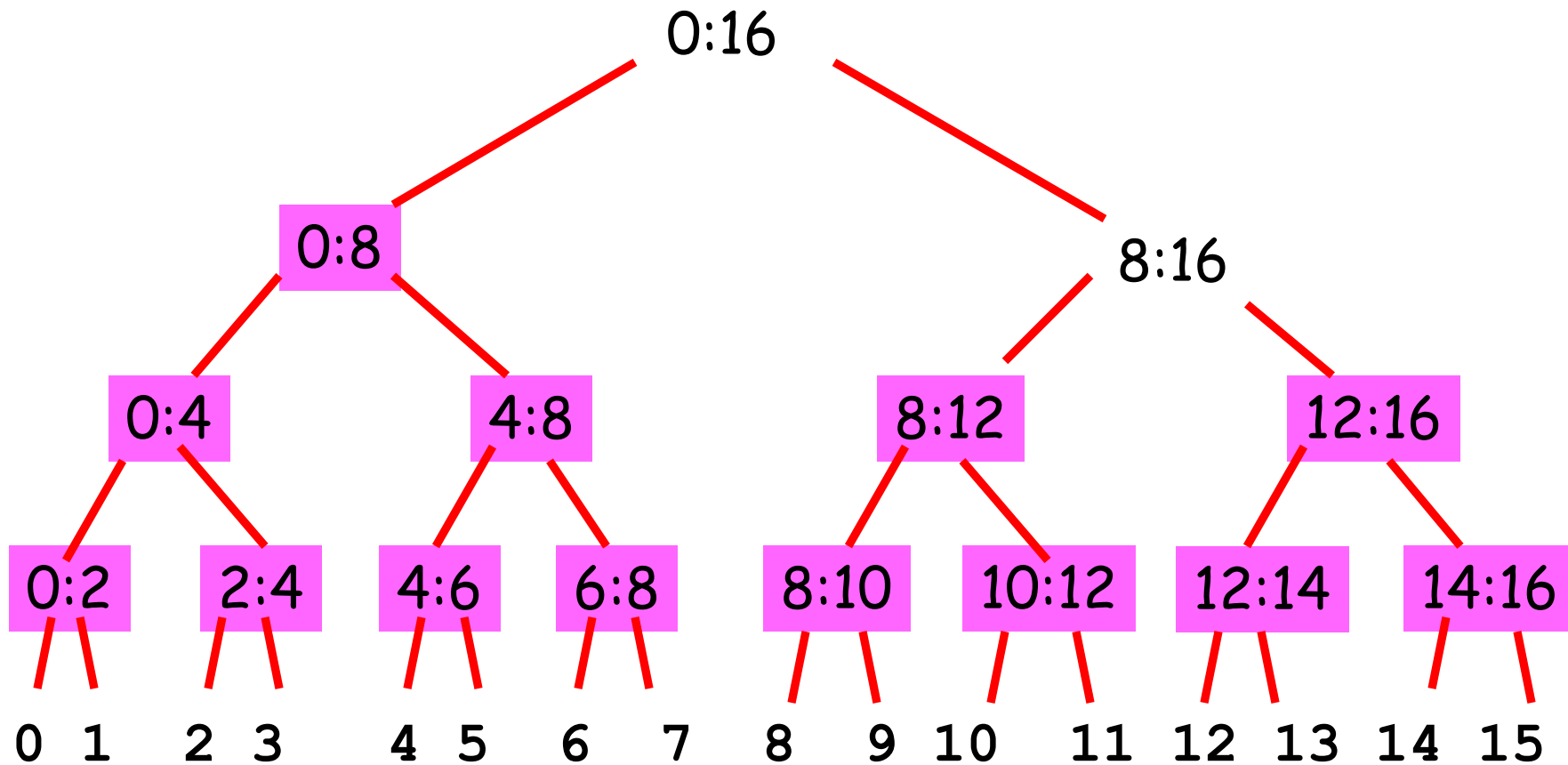
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



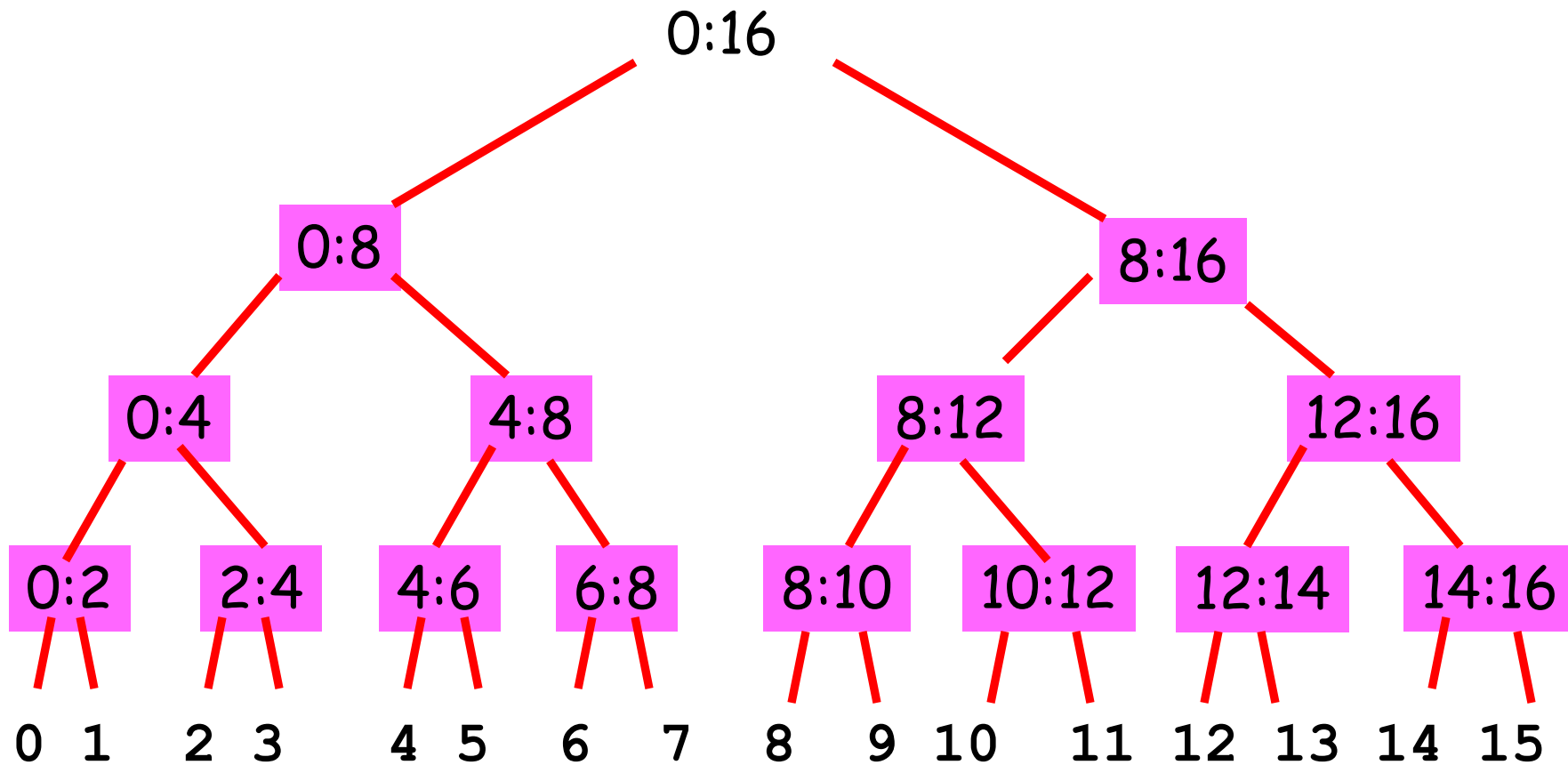
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



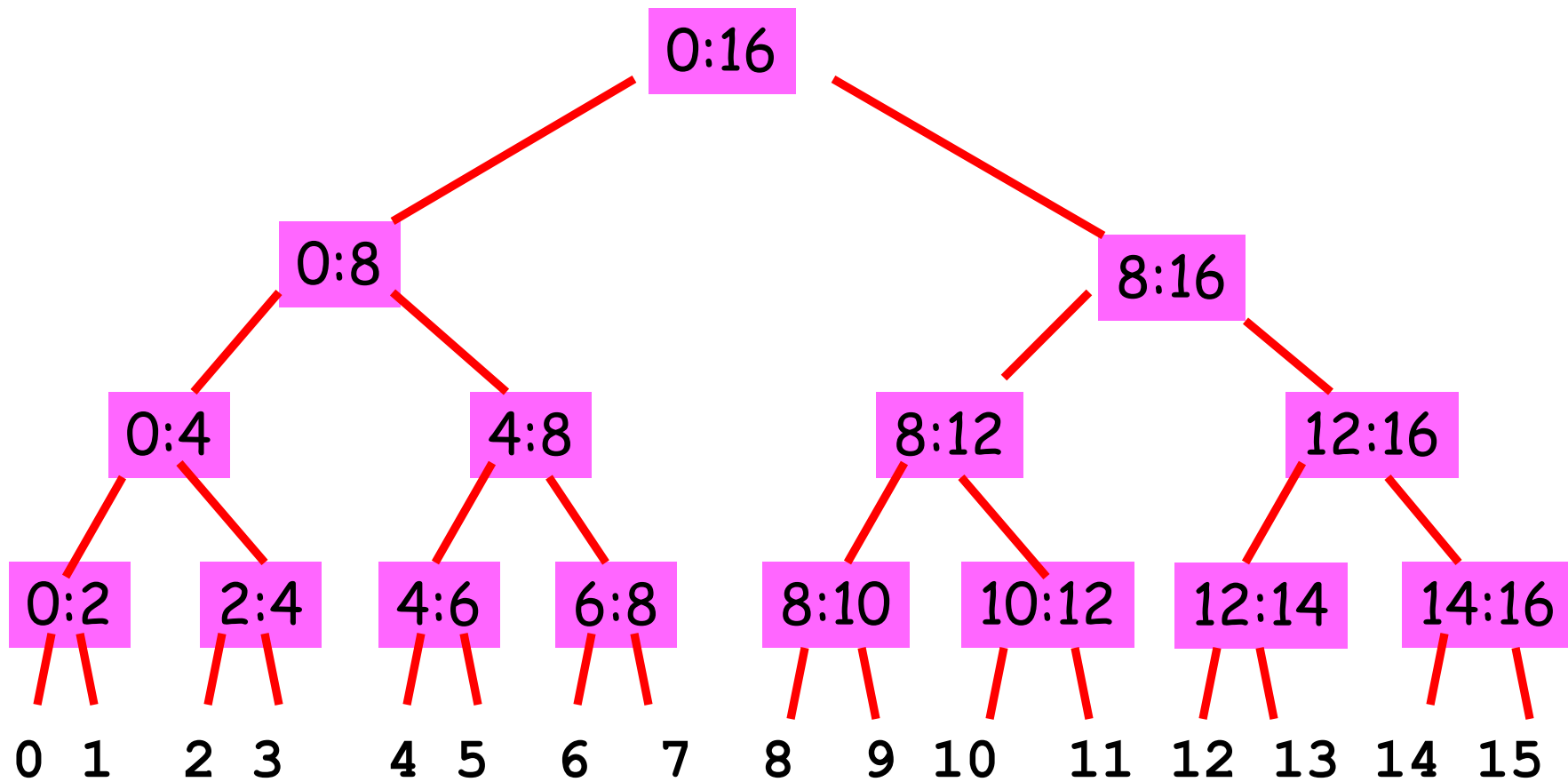
A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



A Sorted List is produced at each ":" Let's look at the order in which lists are sorted.

A Schematic



All Done!

Some Conclusions

Infinite recursion (like infinite loops) can happen so careful reasoning is required.

Will we reach the "base case"?

In **MergeSort**, a recursive call always involves a list that is shorter than the input list. So eventually we reach the $\text{len}(a)=1$ base case.