

# Mark II logbook, Sep 9, 1947

9/9

0800  
1000

Antan started

" stopped - antan ✓

13<sup>00</sup> (032) MP - MC

(033) PRO 2

convd

Relays 6-2 in 033 failed special speed test  
in relay

Relays changed

1100  
1525

Started Cosine Tape (Sine check)

Started Mult + Adder Test.

1545



Relay #70 Panel F  
(moth) in relay.

~~1630~~  
1700

First actual case of bug being found.  
Antan started.  
closed down.

{ 1.2700 9.037 847 025  
9.037 846 995 convd  
~~1.982147000~~  
~~2.130476415~~ (23) 4.615925059(-2)

2.130476415

2.130676415

Relay  
2145  
Relay 3370

# Debugging and delta debugging

CSE 403 Software Engineering

# Today's outline

---

- Debugging basics
- Delta debugging technique
  - In-class exercise will complement this material

# A Bug's Life

---



**Defect** – mistake committed by a human

**Error** – incorrect computation

**Failure** – visible error: program violates its specification

Debugging starts when a failure is observed

- Unit testing
- Integration testing
- In the field

Goal of debugging: go *from failure back to defect*

# Ways to get your code right

---

- Design & verification
  - Prevent defects from appearing in the first place
- Defensive programming
  - Programming debugging in mind: fail fast
- Testing & validation
  - Uncover problems (even in spec), increase confidence
- Debugging
  - Find out why a program is not functioning as intended
- Testing  $\neq$  debugging
  - **test**: reveals existence of problem (failure)
  - **debug**: pinpoint location + cause of problem (defect)

# Defense in depth

---

1. Make errors **impossible**

Java prevents type errors, memory corruption

2. Don't **introduce** defects

Correctness: get things right the first time

3. Make errors immediately **visible**

Example: assertions; **checkRep()**

Converts an error to a failure; reduces distance from defect to failure

4. **Debugging** is the last resort

Work from effect (failure) to cause (defect)

**Scientific method**: Design experiments to gain information about the defect

Easiest in a modular program with good specs and test suites

# Debugging and the scientific method

Debugging must

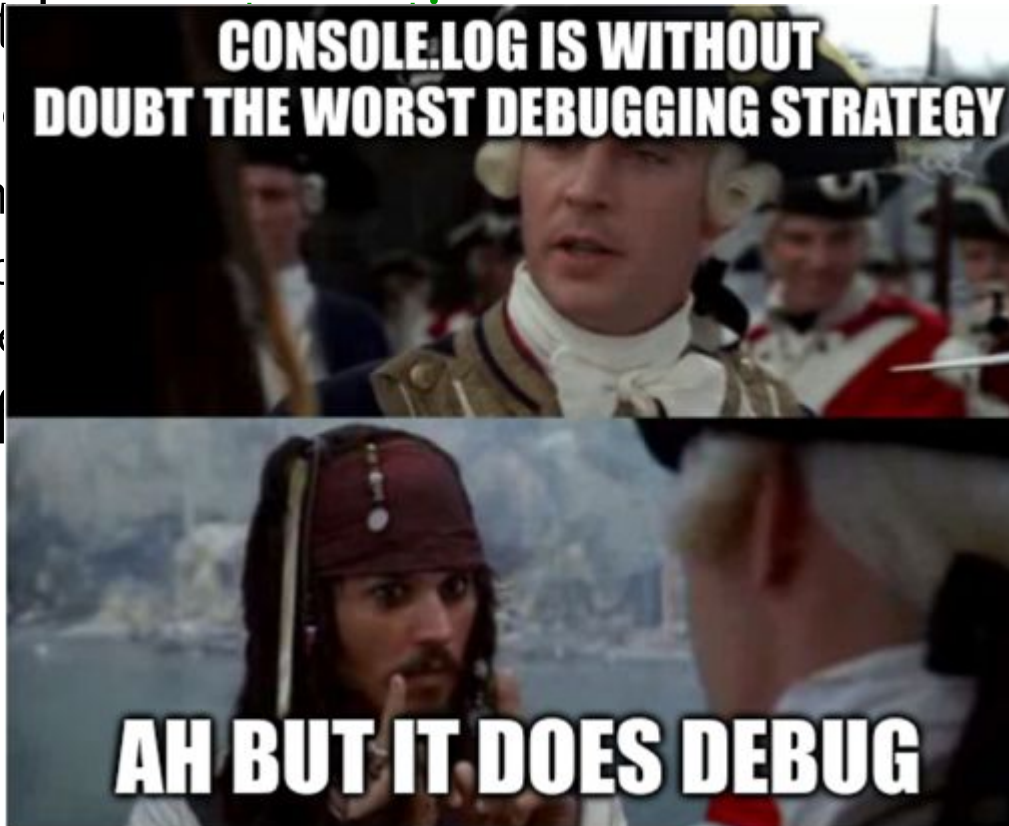
Carefully decide

**Record** everything

Can replicate p

Or avoid the ne

Iterative scientific



hatory power  
ne 3 is wrong"

Interpret **results**

**experiment**

Perform the **experiment**

Should investigate cause

# The typical debugging process

---

- **Identify** – it's a bug, not a feature
- **Understand** – what are the inputs and conditions causing the error
- **Reproduce** – create a (minimal) test to illustrate the issue
- **Investigate** – locate the problematic code
- Capture in a **regression test**
- **Fix** the code
- **Validate**

# Writing a good bug (issue) report

---

**A bug report** should be as specific as possible so that the engineer knows how to recreate the failure

- Provide information to reproduce the bug, including context
- What might be “context”?

**A test case** should be as simple as possible

- Why?

# Binary search (e.g., git bisect)

Continuous integration runs:



Add a **new test case**:



Binary search is not guaranteed to reproduce the original failure.

=> You might not fix the defect that caused that failure.

v13 might have failed in a different way,  
for a different reason.

After fix,  
passes

Still fails!

Did this use the scientific method?

# Delta Debugging

---

A debugging technique to create a **minimal** test case that fails *in the same way*.

Input:

- Program
- Failing test case

Output:

- Failing test case that is as small as possible

# This is a crashing test case



```
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All
<OPTION VALUE="Windows 3.1">Windows 3.1
<OPTION VALUE="Windows 95">Windows 95
<OPTION VALUE="Windows 98">Windows 98
<OPTION VALUE="Windows ME">Windows ME
<OPTION VALUE="Windows 2000">Windows 2000
<OPTION VALUE="Windows NT">Windows NT
<OPTION VALUE="Mac System 7">Mac System 7
<OPTION VALUE="Mac System 7.5">Mac System 7.5
<OPTION VALUE="Mac System 7.6.1">Mac System 7.6.1
<OPTION VALUE="Mac System 8.0">Mac System 8.0
<OPTION VALUE="Mac System 8.5">Mac System 8.5
<OPTION VALUE="Mac System 8.6">Mac System 8.6
<OPTION VALUE="Mac System 9.x">Mac System 9.x
<OPTION VALUE="MacOS X">MacOS X
<OPTION VALUE="Linux">Linux
<OPTION VALUE="BSDI">BSDI
<OPTION VALUE="FreeBSD">FreeBSD
<OPTION VALUE="NetBSD">NetBSD
<OPTION VALUE="OpenBSD">OpenBSD
<OPTION VALUE="AIX">AIX
<OPTION VALUE="BeOS">BeOS
<OPTION VALUE="HP-UX">HP-UX
<OPTION VALUE="IRIX">IRIX
<OPTION VALUE="Neutrino">Neutrino
<OPTION VALUE="OpenVMS">OpenVMS
<OPTION VALUE="OS/2">OS/2
<OPTION VALUE="OSF/1">OSF/1
<OPTION VALUE="Solaris">Solaris
<OPTION VALUE="SunOS">SunOS
<OPTION VALUE="other">other</SELECT></td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION
VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION
VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION
VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</td>
</tr>
</table>
```

- Crashed Mozilla
- Consider 370 of these being filed!
- What content is sufficient to reproduce the failure?

# This is a crashing test case



```
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="A11">A11
<OPTION VALUE="Windows 3.1">Windows 3.1
<OPTION VALUE="Windows 95">Windows 95
<OPTION VALUE="Windows 98">Windows 98
<OPTION VALUE="Windows ME">Windows ME
<OPTION VALUE="Windows 2000">Windows 2000
<OPTION VALUE="Windows NT">Windows NT
<OPTION VALUE="Mac System 7">Mac System 7
<OPTION VALUE="Mac System 7.5">Mac System 7.5
<OPTION VALUE="Mac System 7.6.1">Mac System 7.6.1
<OPTION VALUE="Mac System 8.0">Mac System 8.0
<OPTION VALUE="Mac System 8.5">Mac System 8.5
<OPTION VALUE="Mac System 8.6">Mac System 8.6
<OPTION VALUE="Mac System 9.x">Mac System 9.x
<OPTION VALUE="MacOS X">MacOS X
<OPTION VALUE="Linux">Linux
<OPTION VALUE="BSDI">BSDI
<OPTION VALUE="FreeBSD">FreeBSD
<OPTION VALUE="NetBSD">NetBSD
<OPTION VALUE="OpenBSD">OpenBSD
<OPTION VALUE="AIX">AIX
<OPTION VALUE="BeOS">BeOS
<OPTION VALUE="HP-UX">HP-UX
<OPTION VALUE="IRIX">IRIX
<OPTION VALUE="Neutrino">Neutrino
<OPTION VALUE="OpenVMS">OpenVMS
<OPTION VALUE="OS/2">OS/2
<OPTION VALUE="OSF/1">OSF/1
<OPTION VALUE="Solaris">Solaris
<OPTION VALUE="SunOS">SunOS
<OPTION VALUE="other">other</SELECT></td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">---<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION
VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION
VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION
VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```

- Crashed Mozilla
- What content is sufficient to reproduce the failure?
- A minimal test case is:  
**<SELECT>**
- Can we automate the process of minimizing test cases?
- Idea: use binary search

# Try the first half of the input

```
<td align=left valign=top>  
<SELECT NAME="op sys" MULTIPLE SIZE=7>  
<OPTION VALUE="All">All  
<OPTION VALUE="Windows 3.1">Windows 3.1  
<OPTION VALUE="Windows 95">Windows 95  
<OPTION VALUE="Windows 98">Windows 98  
<OPTION VALUE="Windows ME">Windows ME  
<OPTION VALUE="Windows 2000">Windows 2000  
<OPTION VALUE="Windows NT">Windows NT  
<OPTION VALUE="Mac System 7">Mac System 7  
<OPTION VALUE="Mac System 7.5">Mac System 7.5  
<OPTION VALUE="Mac System 7.6.1">Mac System 7.6.1  
<OPTION VALUE="Mac System 8.0">Mac System 8.0  
<OPTION VALUE="Mac System 8.5">Mac System 8.5  
<OPTION VALUE="Mac System 8.6">Mac System 8.6  
<OPTION VALUE="Mac System 9.x">Mac System 9.x  
<OPTION VALUE="MacOS X">MacOS X  
<OPTION VALUE="Linux">Linux  
<OPTION VALUE="BSDI">BSDI  
<OPTION VALUE="FreeBSD">FreeBSD  
<OPTION VALUE="NetB
```

- Crashed Mozilla
- What content is sufficient to reproduce the failure?
- A minimal test case is:  
**<SELECT>**
- Can we automate the process of minimizing test cases?
- Idea: use binary search
- What is the result of the test?

# Minimizing test cases

---

Test case

Test case

Test case

Think of each test case as an input file with  $n$  lines

# Delta Debugging

---

A debugging technique to create a **minimal** test case that fails *in the same way*.

Input:

- Program
- Failing test case
- 

Output:

- Failing test case that is as small as possible

# Delta Debugging

---

A debugging technique to create a **minimal** test case that fails *in the same way*.

Input:

- Program
- Failing test case
- Predicate on executions:  
did the execution fail in the same way?

Output:

- Failing test case that is as small as possible

Test passes => **false**  
Test fails in the same way => **true**  
Test fails in some other way => **false**

# Minimizing test cases

---

Test case

**Failing**

Test case

**Passing**

Test case

**Passing**

# Minimizing test cases

---

Test case

**Failing**

Test case

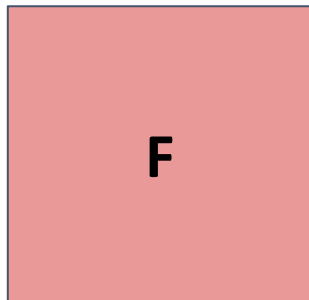
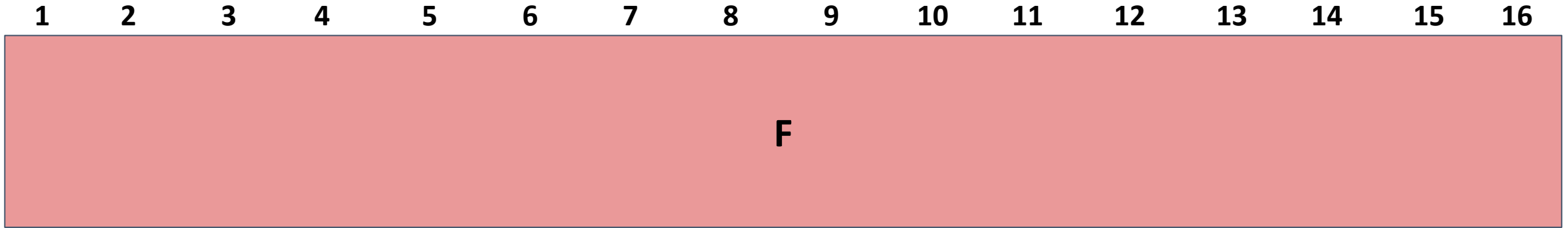
**Passing**

Test case

**Passing**

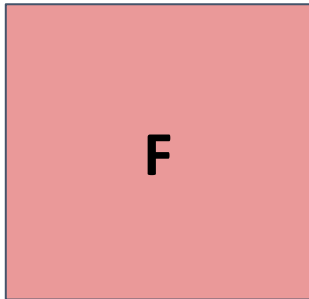
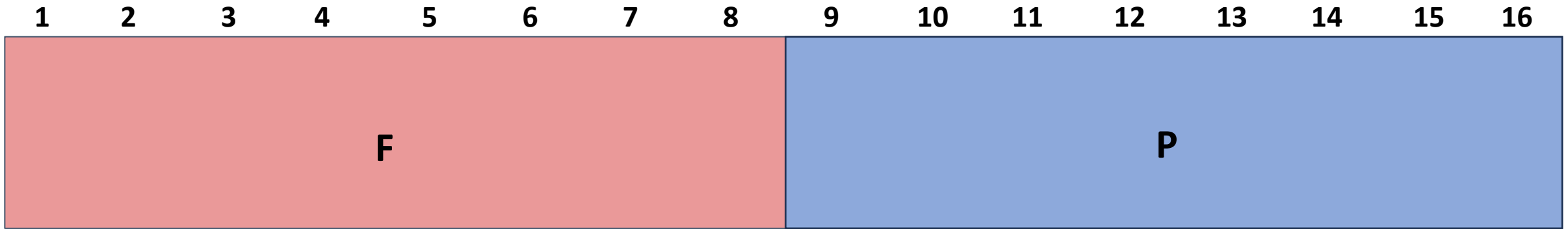
**Goal: minimize the failing test case**

# The happy path: binary search

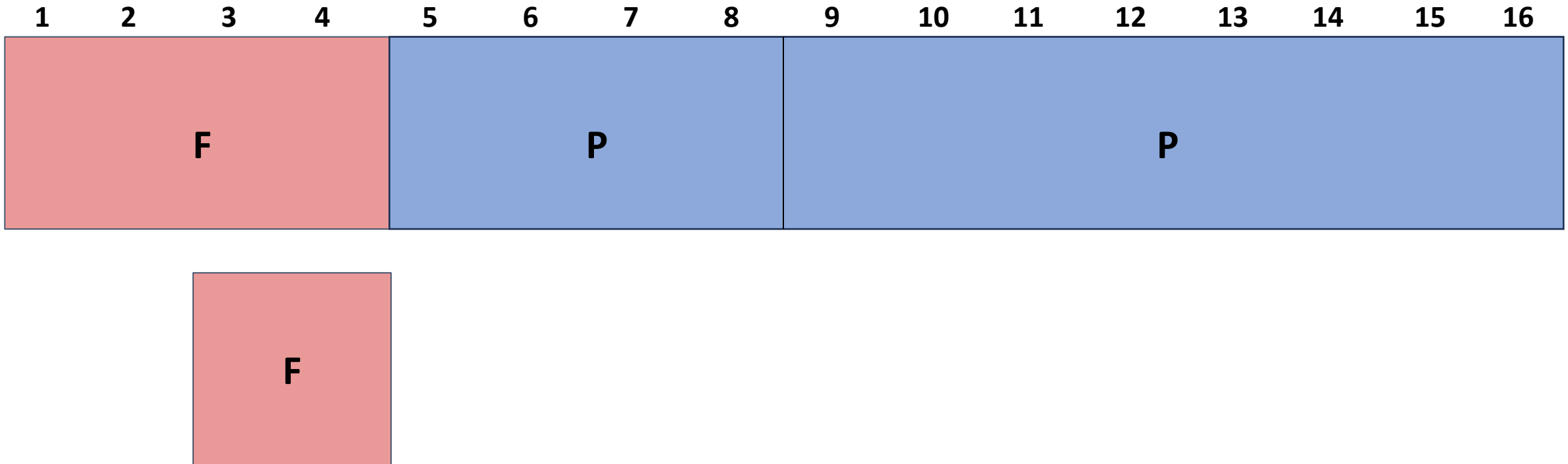


Failing test with 16 lines  
The minimal failing test has 2 lines: 3 and 4

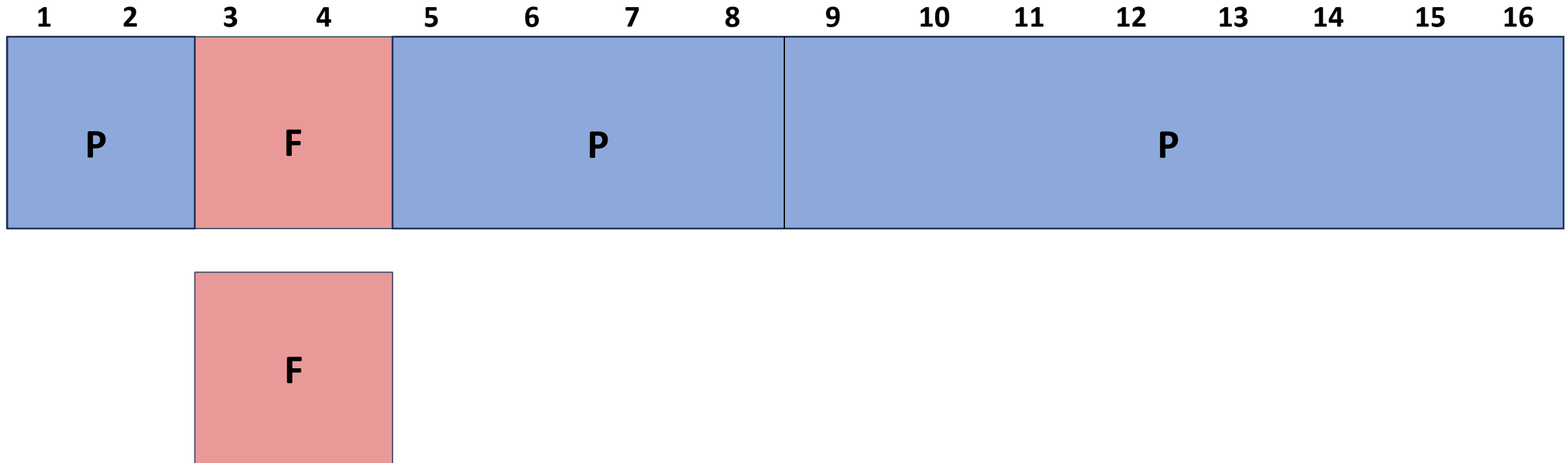
# The happy path: binary search



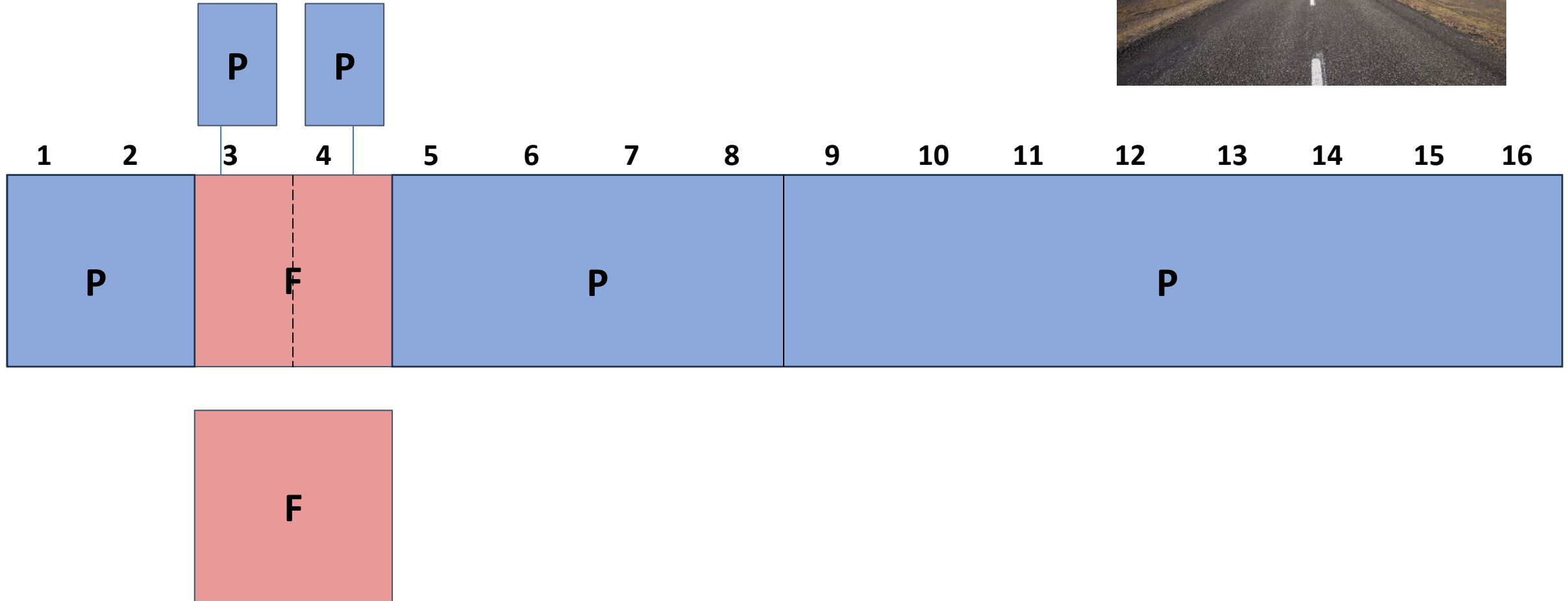
# The happy path: binary search



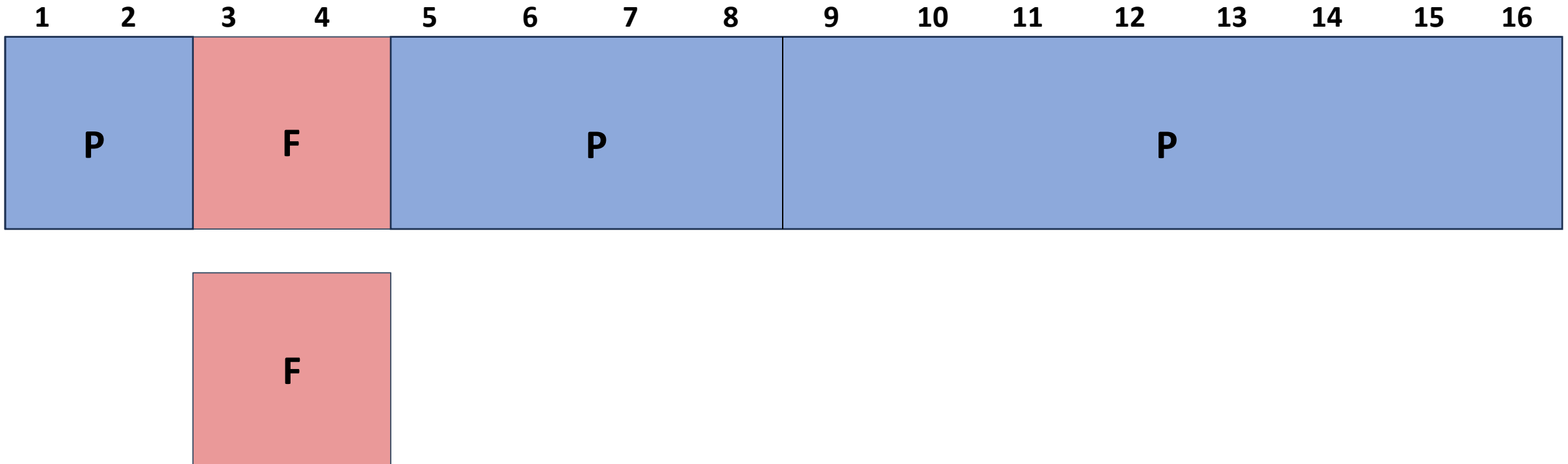
# The happy path: binary search



# The happy path: binary search



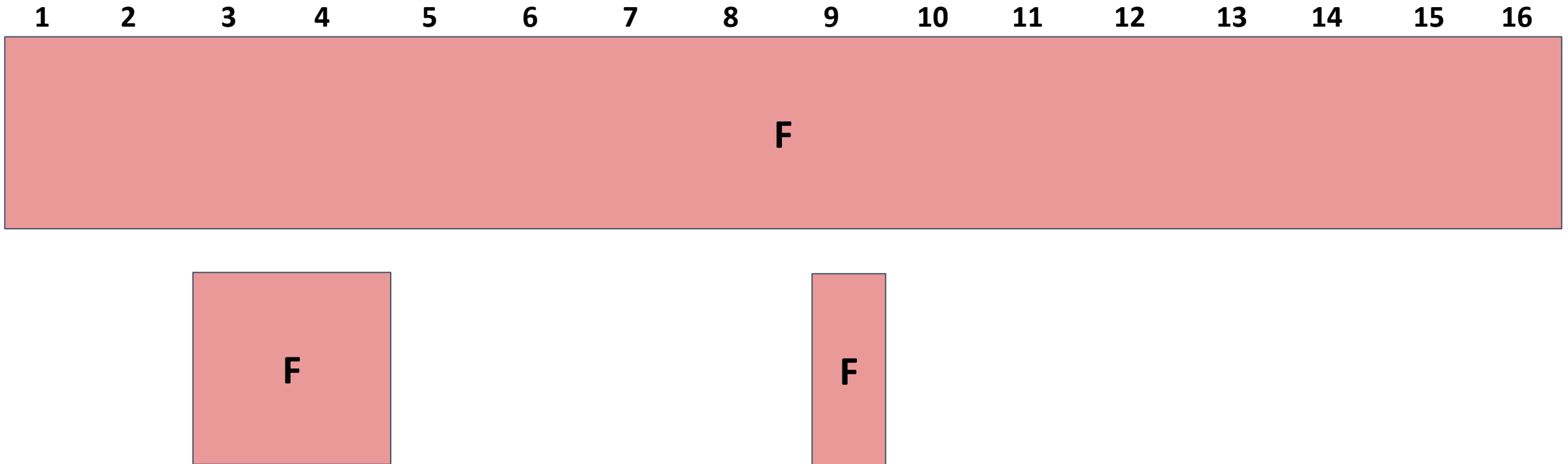
# The happy path: binary search



Successfully minimized the failing test to 2 lines

# The not so happy path...

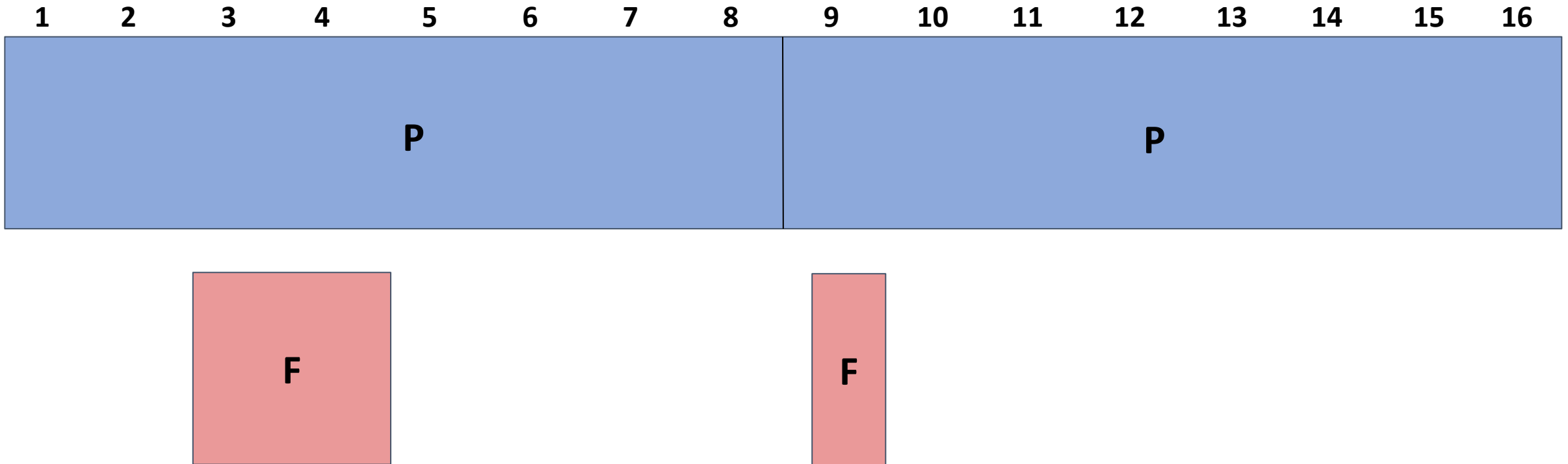
---



Suppose the failure pattern is more complex  
All three lines must exist in a failing test case: 3, 4, and 9

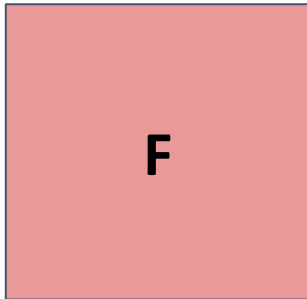
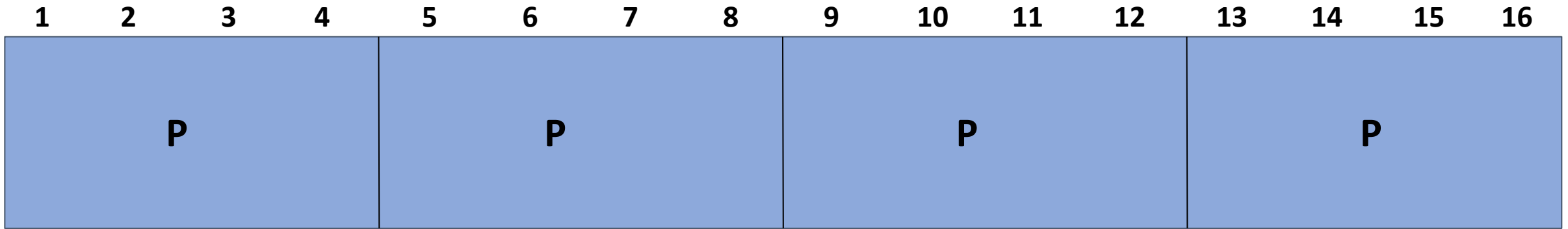
# The not so happy path...

---



# The not so happy path...

---



Binary search is no help

**Delta Debugging = binary search  
+ account for multiple  
types of test outcomes**

See paper:

Simplifying and Isolating Failure-Inducing Input

Zeller and Hildebrandt, 2002

# The Delta Debugging algorithm

## Four basic phases:

1. Test each subset  
(= binary subdivision)
2. Test each complement
3. Increase granularity  
(increase # subsets)
4. Reduce (= recurse)

Complement example:

Input = 1, 2, 3, 4

A subset is { 1 }

Its complement is { 2, 3, 4 }

## Minimizing Delta Debugging Algorithm

Let  $test$  and  $c_{\mathbf{x}}$  be given such that  $test(\emptyset) = \checkmark \wedge test(c_{\mathbf{x}}) = \mathbf{x}$  hold.

The goal is to find  $c'_{\mathbf{x}} = ddmin(c_{\mathbf{x}})$  such that  $c'_{\mathbf{x}} \subseteq c_{\mathbf{x}}$ ,  $test(c'_{\mathbf{x}}) = \mathbf{x}$ , and  $c'_{\mathbf{x}}$  is 1-minimal.

The minimizing Delta Debugging algorithm  $ddmin(c)$  is

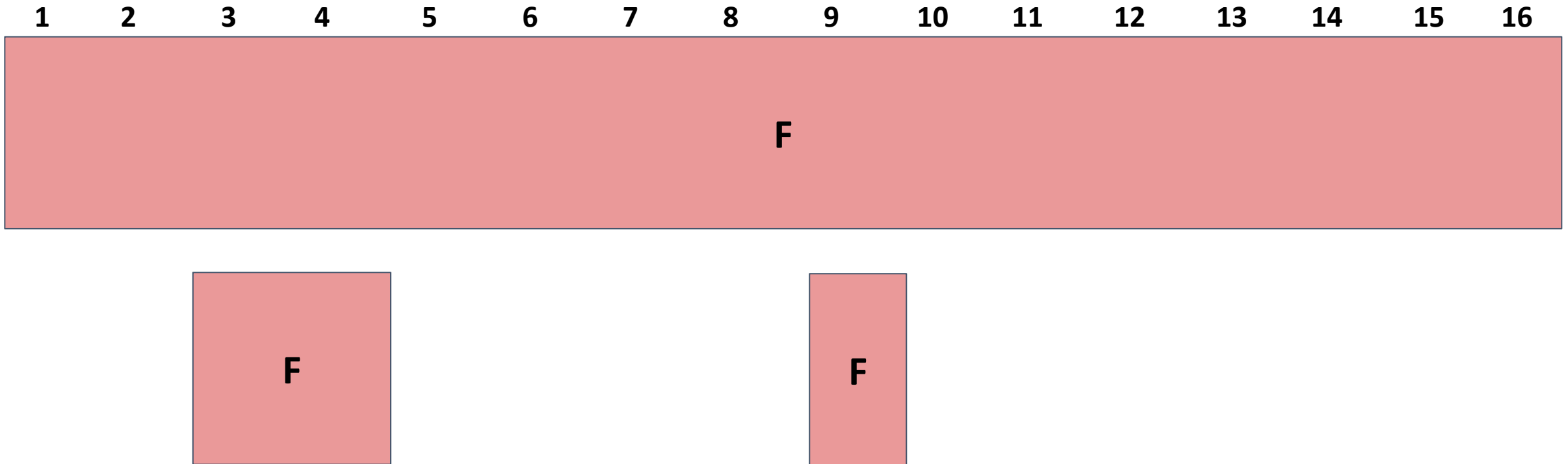
$$ddmin(c_{\mathbf{x}}) = ddmin_2(c_{\mathbf{x}}, 2) \quad \text{where}$$
$$ddmin_2(c'_{\mathbf{x}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \dots, n\} \cdot test(\Delta_i) = \mathbf{x} \text{ ("reduce to subset")} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \dots, n\} \cdot test(\nabla_i) = \mathbf{x} \text{ ("reduce to complement")} \\ ddmin_2(c'_{\mathbf{x}}, \min(|c'_{\mathbf{x}}|, 2n)) & \text{else if } n < |c'_{\mathbf{x}}| \text{ ("increase granularity")} \\ c'_{\mathbf{x}} & \text{otherwise ("done")}. \end{cases}$$

where  $\nabla_i = c'_{\mathbf{x}} - \Delta_i$ ,  $c'_{\mathbf{x}} = \Delta_1 \cup \Delta_2 \cup \dots \cup \Delta_n$ , all  $\Delta_i$  are pairwise disjoint, and  $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\mathbf{x}}|/n$  holds.

The recursion invariant (and thus precondition) for  $ddmin_2$  is  $test(c'_{\mathbf{x}}) = \mathbf{x} \wedge n \leq |c'_{\mathbf{x}}|$ .

# Delta Debugging is mostly binary search

---



# Delta Debugging: test subsets

---

Notation for  
subset

$\Delta_1$

$\Delta_2$

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

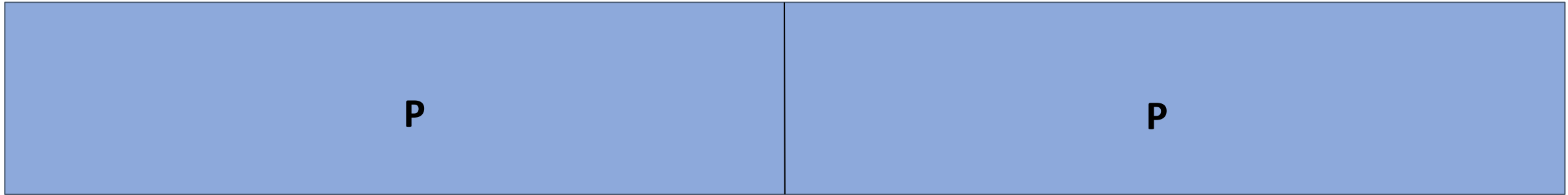
16

P

P

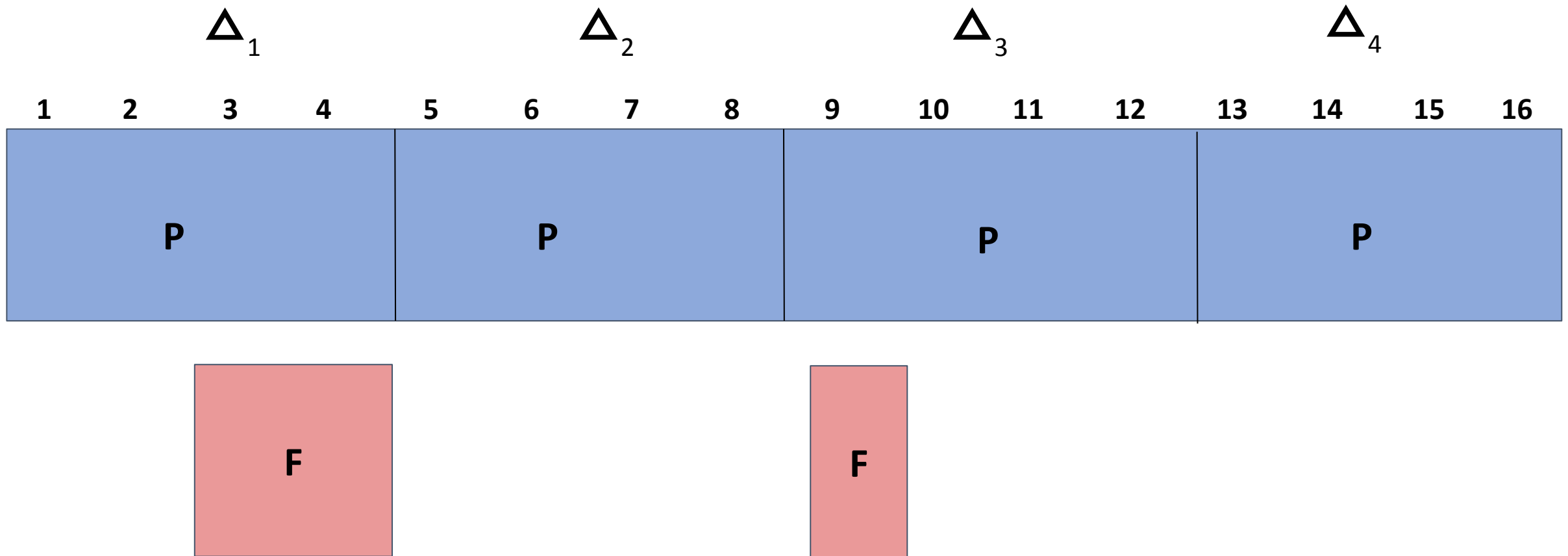
F

F



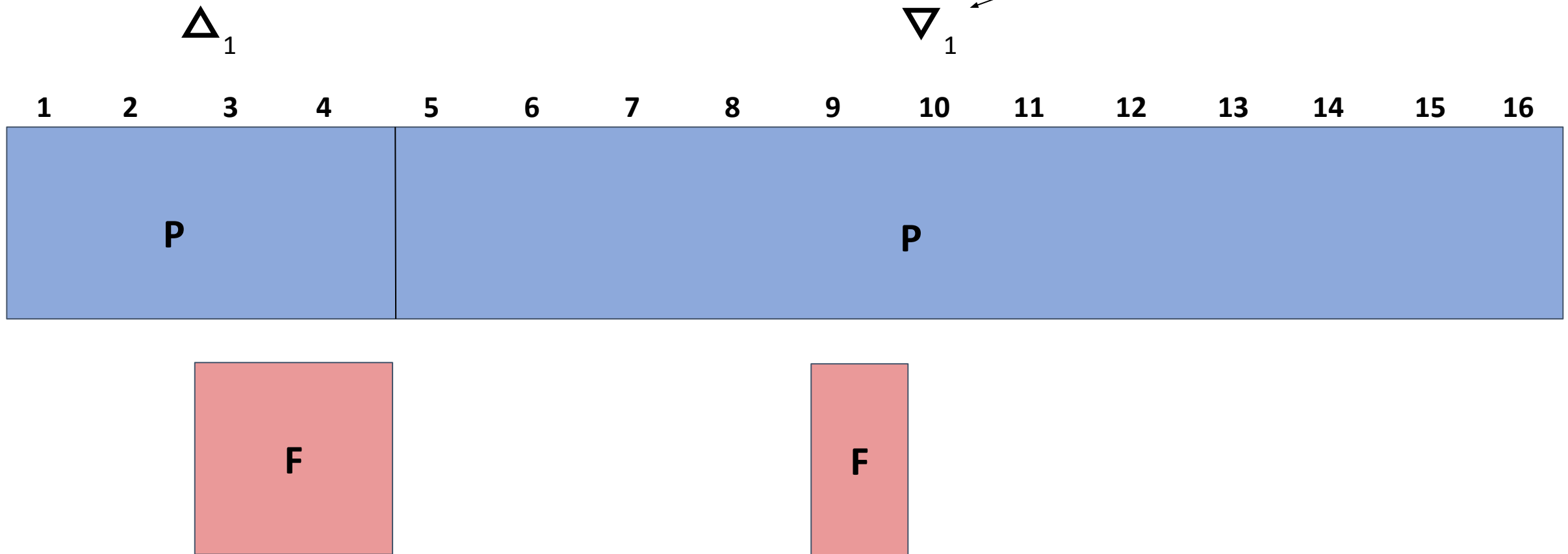
# Delta Debugging: increase granularity

---



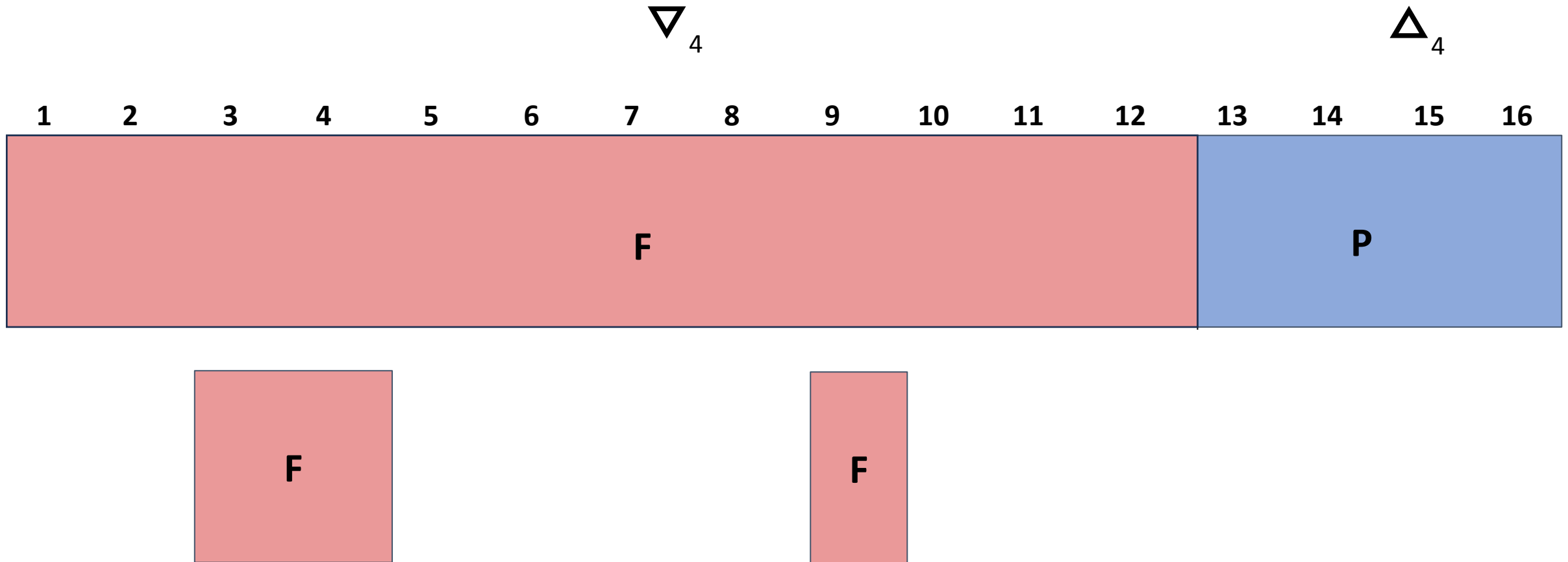
# Delta Debugging: complements

Notation for  
complement  
of subset 1



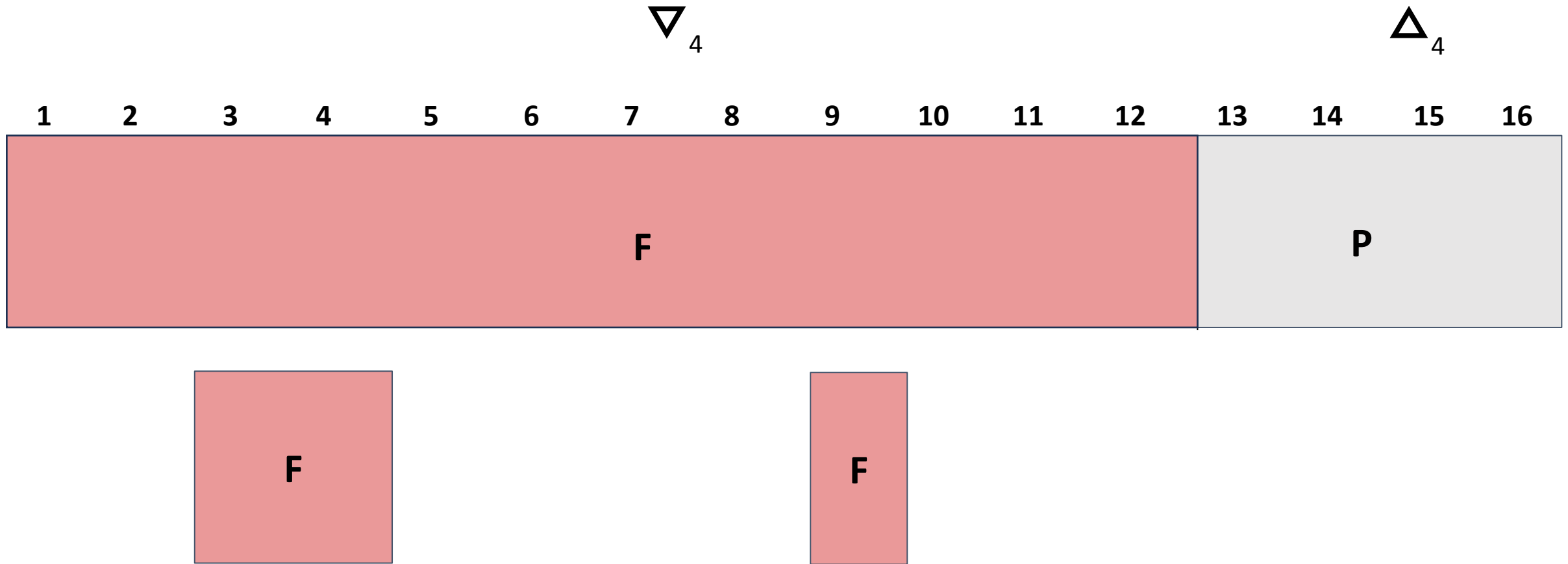
# Delta Debugging: complements

---



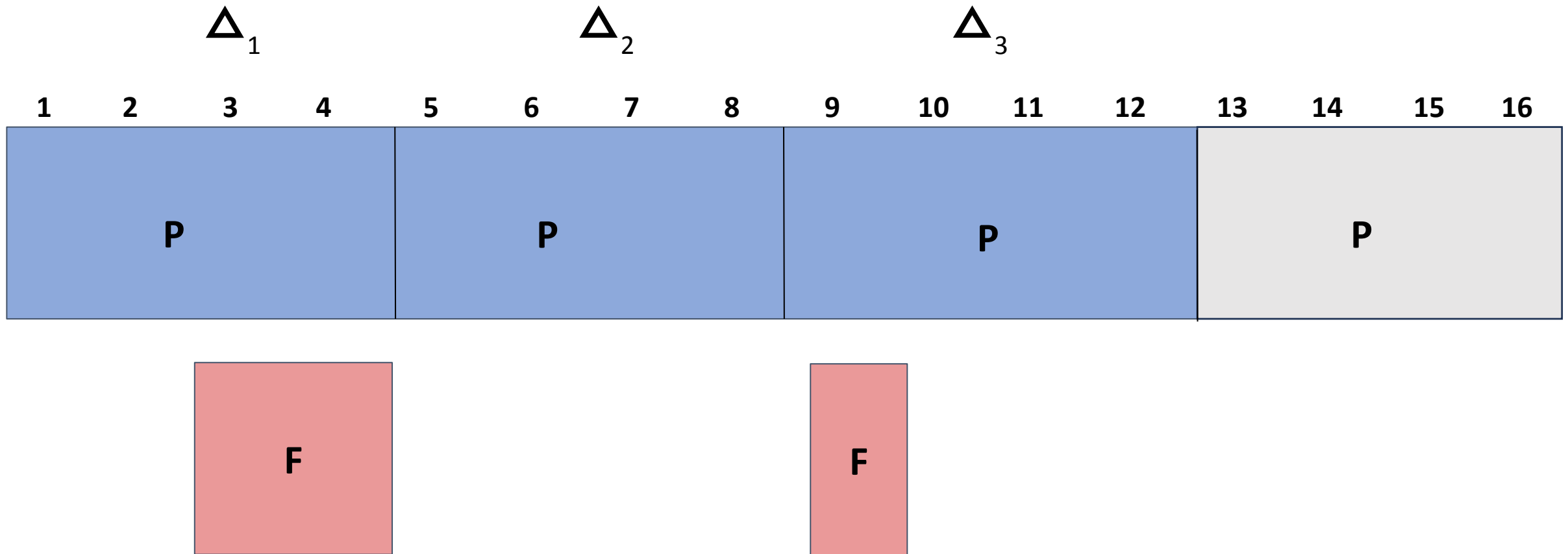
# Delta Debugging: reduce

---



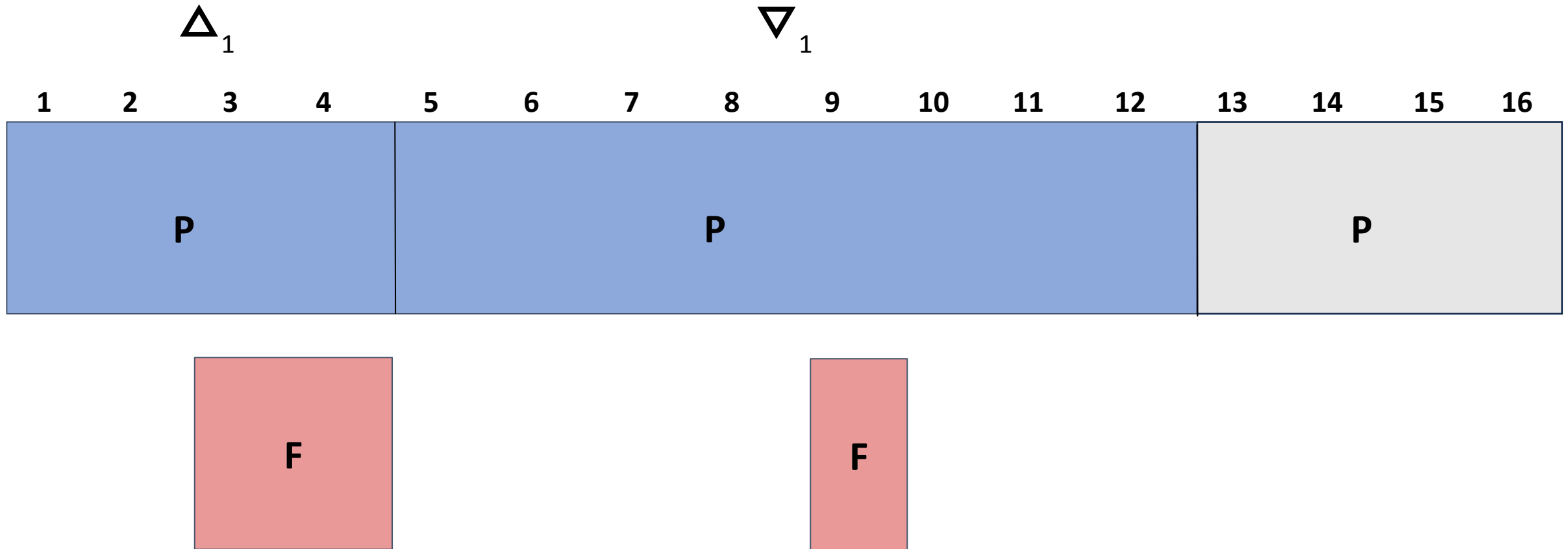
# Delta Debugging: test subsets

---



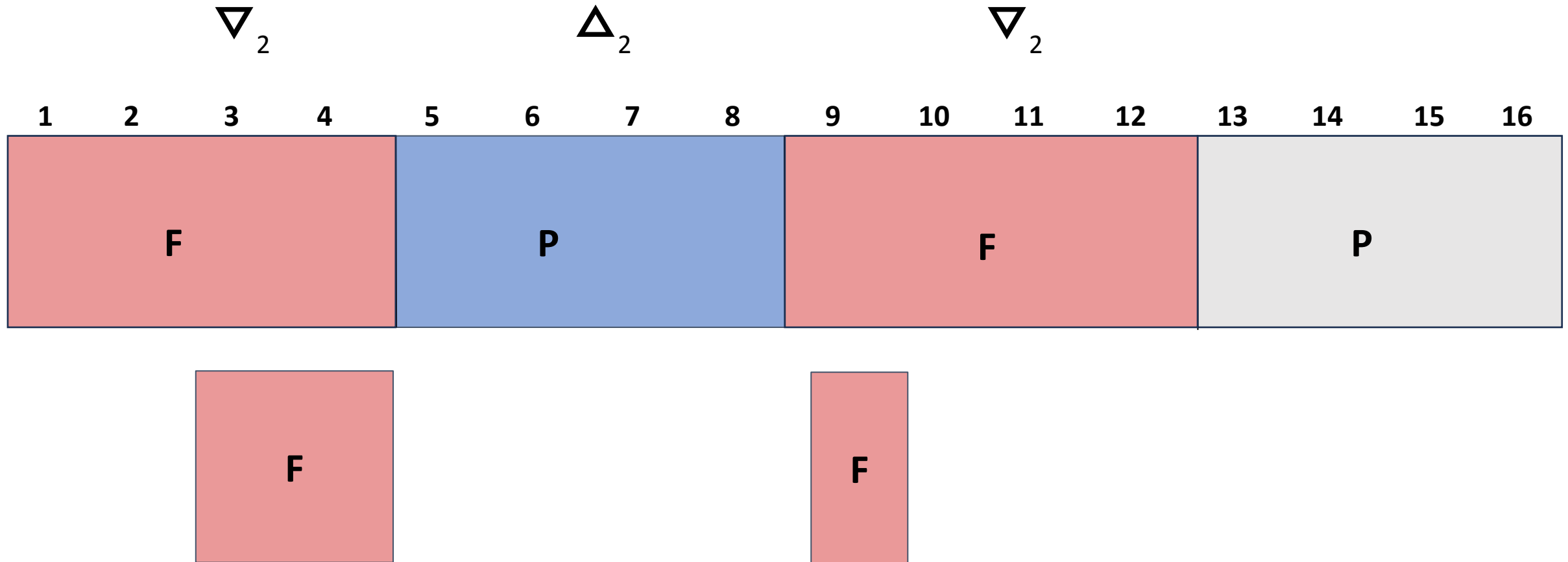
# Delta Debugging: complements

---



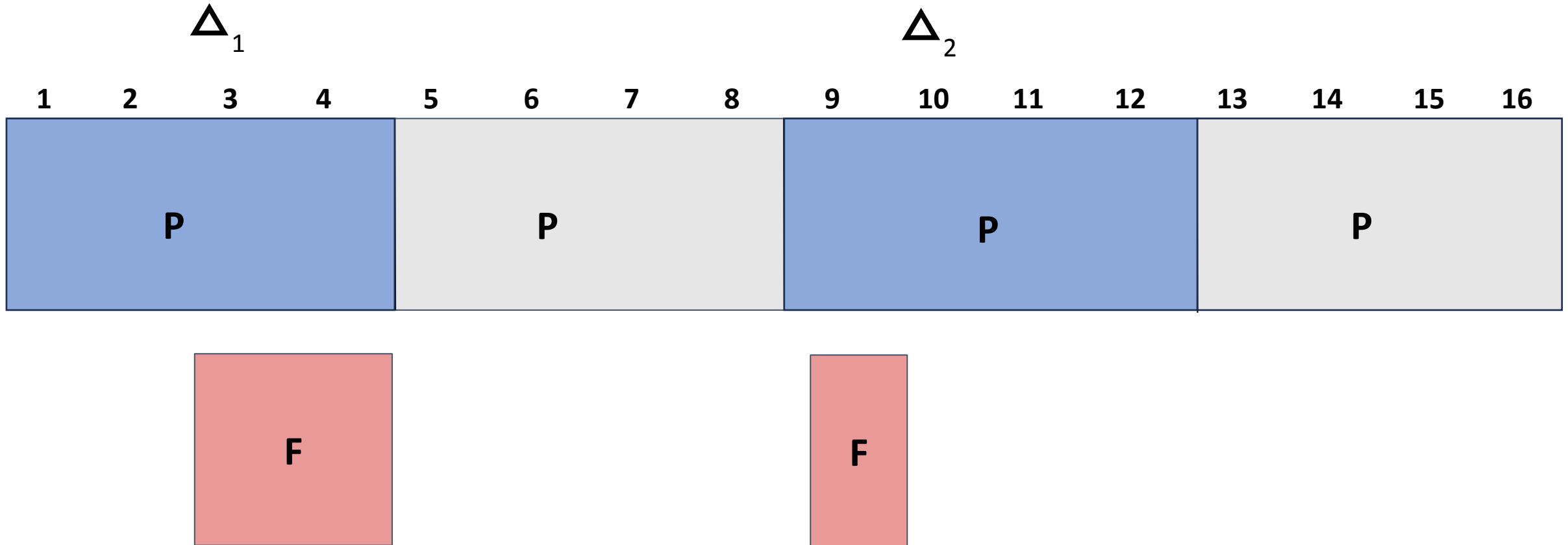
# Delta Debugging: complements

---

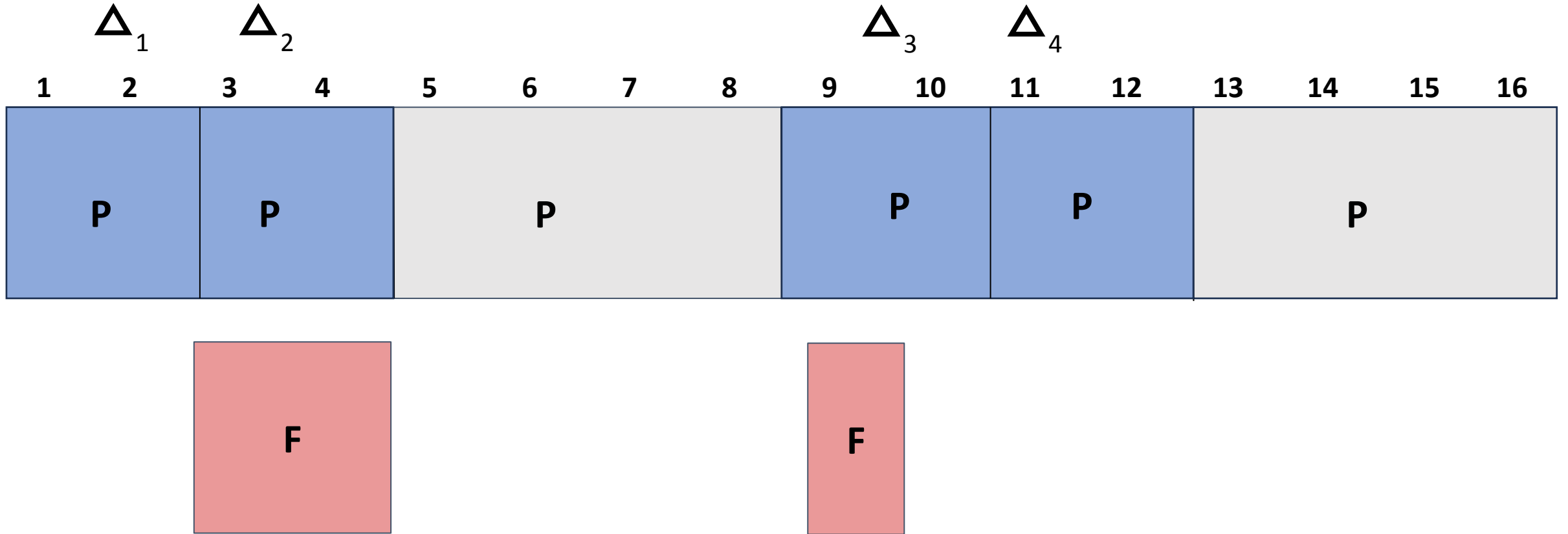


# Delta Debugging: reduce

---

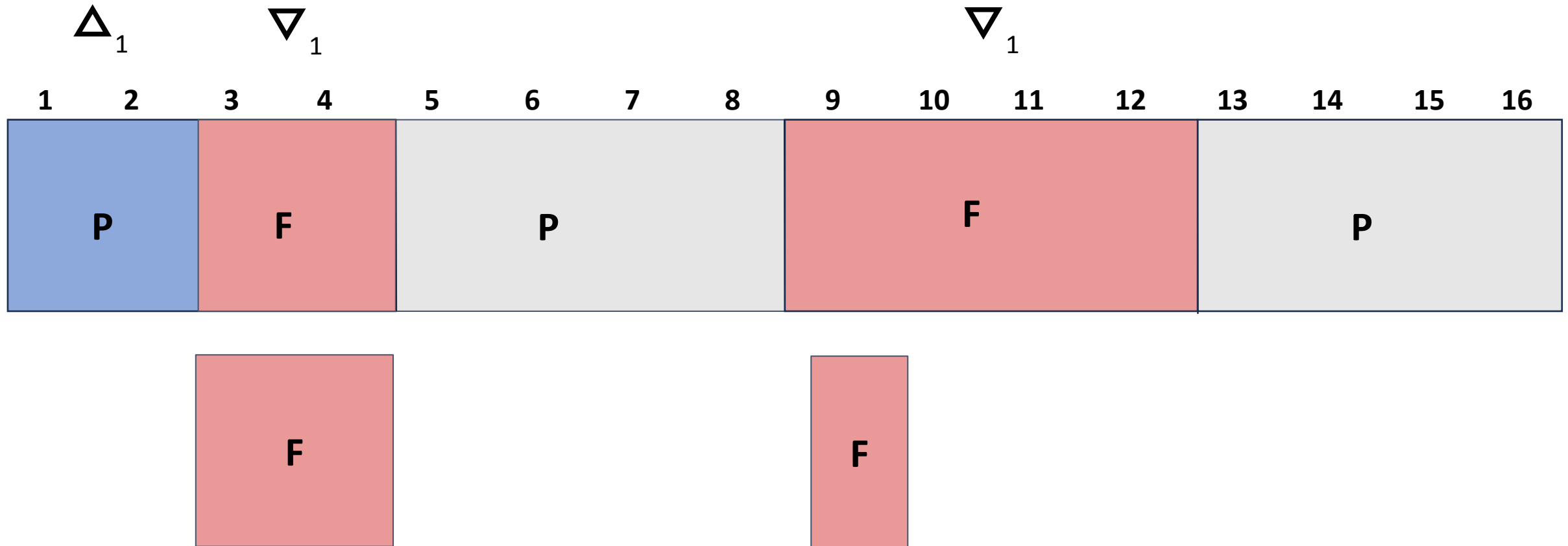


# Delta Debugging: increase granularity



# Delta Debugging: complements

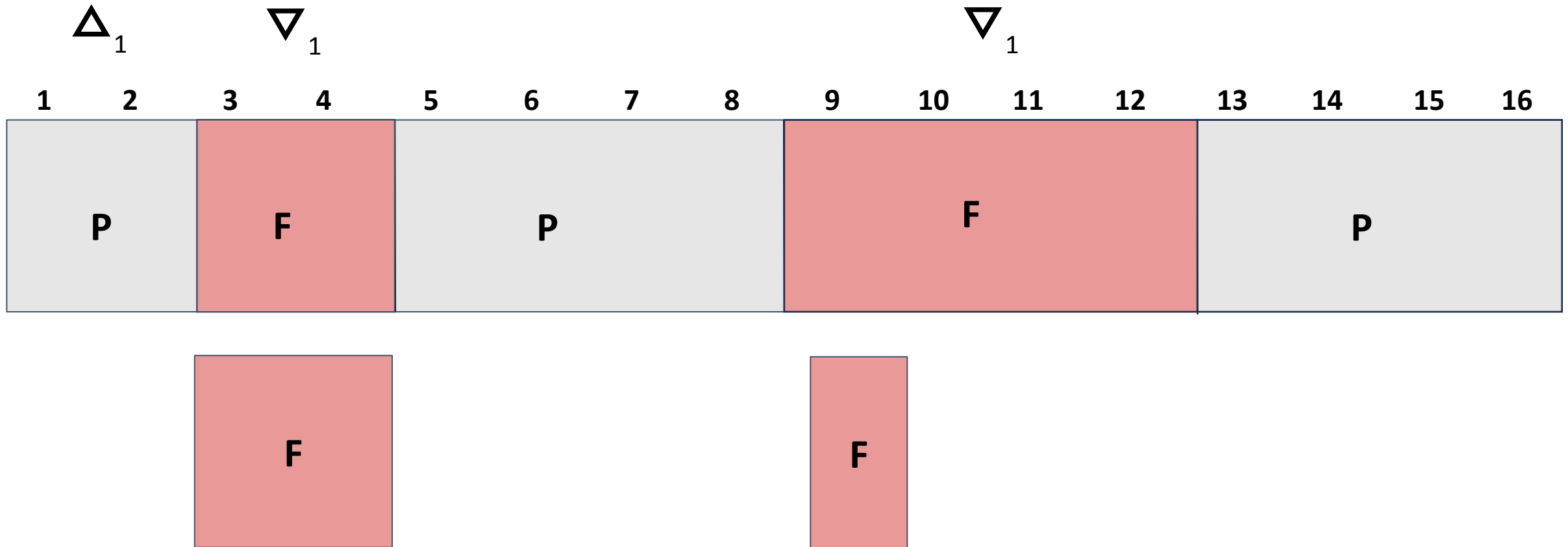
---



And so on...

# Delta Debugging: reduction

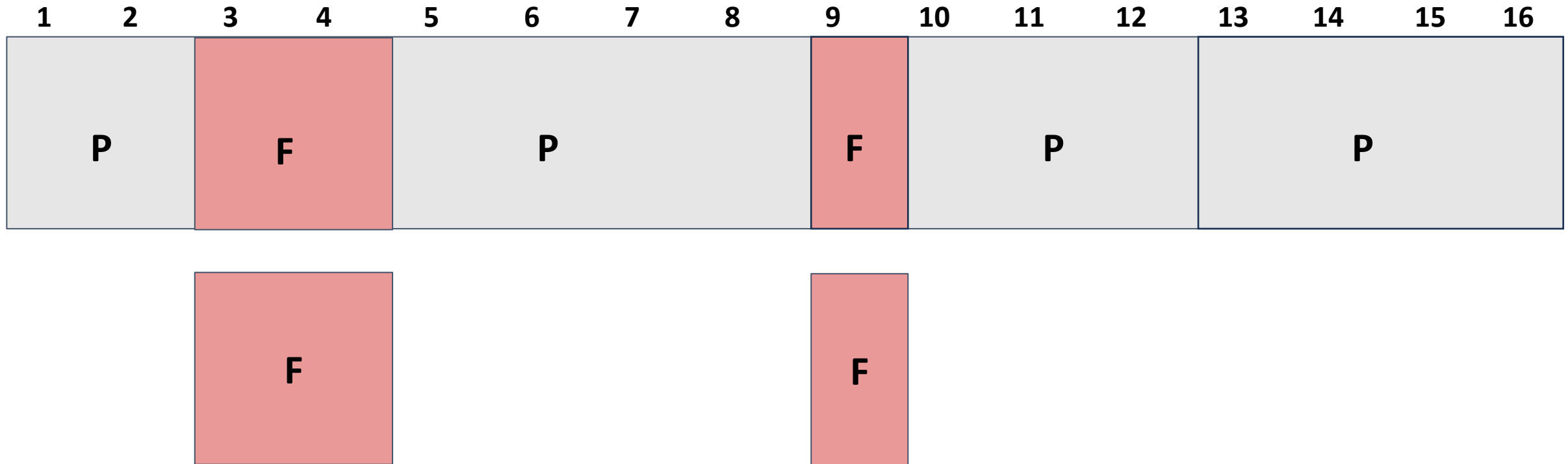
---



And so on...

# Delta Debugging finds a “1-minimal” solution

---



Failing test cases must be deterministic and monotone

# Delta debugging: live example

# Delta Debugging: live example

---

## Program and initial test case

- Program **P** **crashes** whenever the input contains **1 7 8**
- Initial crashing test case is: **1 2 3 4 5 6 7 8**

## Syntax:

- % `./delta -test=./test.sh -cp_minimal=./min.txt < failing.txt`
- `test.sh` returns 0 if input causes failure, 1 if input passes

## Delta debugging approach:

- Test each subset
- Test each complement
- Increase granularity
- Reduce

# Delta debugging: one more example

# Let's try one more

---

## Program and initial test case

- Program  $P$  takes as **input** a **list of integers**  $I$ .
- $P$  **crashes** whenever  $I$  **contains** 4,2.
- **Initial** crashing **test case** is: 2,4,2,4

## Complete the following table

| Iteration | n   | input | $\Delta_1, \dots, \Delta_n$<br>$\nabla_1, \dots, \nabla_n$ |
|-----------|-----|-------|------------------------------------------------------------|
| 1         | 2   | 2424  | ...                                                        |
| 2         | ... | ...   | ...                                                        |

# Let's try one more

---

## Program and initial test case

- Program *P* takes as input a list of integers *I*.
- *P* crashes whenever *I* contains 4,2.
- Initial crashing test case is: 2,4,2,4

## Complete the following table

| Iteration | n | input | $\Delta_1, \dots, \Delta_n$<br>$\nabla_1, \dots, \nabla_n$ |
|-----------|---|-------|------------------------------------------------------------|
| 1         | 2 | 2424  | 24, (24)                                                   |
| 2         | 4 | 2424  | 2, 4, (2), (4),<br><b>424</b> , (224), (244),<br>(242)     |
| 3         | 3 | 424   | (4), (2), (4),<br>(24), 44, <b>42</b>                      |
| 4         | 2 | 42    | (4),( 2)                                                   |

**Delta debugging: in-class exercise soon**

# Examples from Monday demo

**Oracle test** (look for (1 and 2) or 8, 0=test fails, 1=test passes)

**test.sh**

```
#!/bin/sh  
cat $1 | tr -d '\n' | grep -q -E "12|8"
```

**review-files.sh**

```
#!/bin/sh  
for file in tmp0/*.c ; do cat $file | tr -d '\n' ; echo ; done
```

**Four basic phases:**

1. Test subsets
2. Test complements
3. Increase  
granularity
4. Reduce

# Sample oracle test if you don't have one yet

test.sh

```
#!/bin/sh
```

```
# *-sh-*-
```

```
if timeout 0.5s perl ../../mysort.pl $1 >/dev/null; then exit 1; fi
```