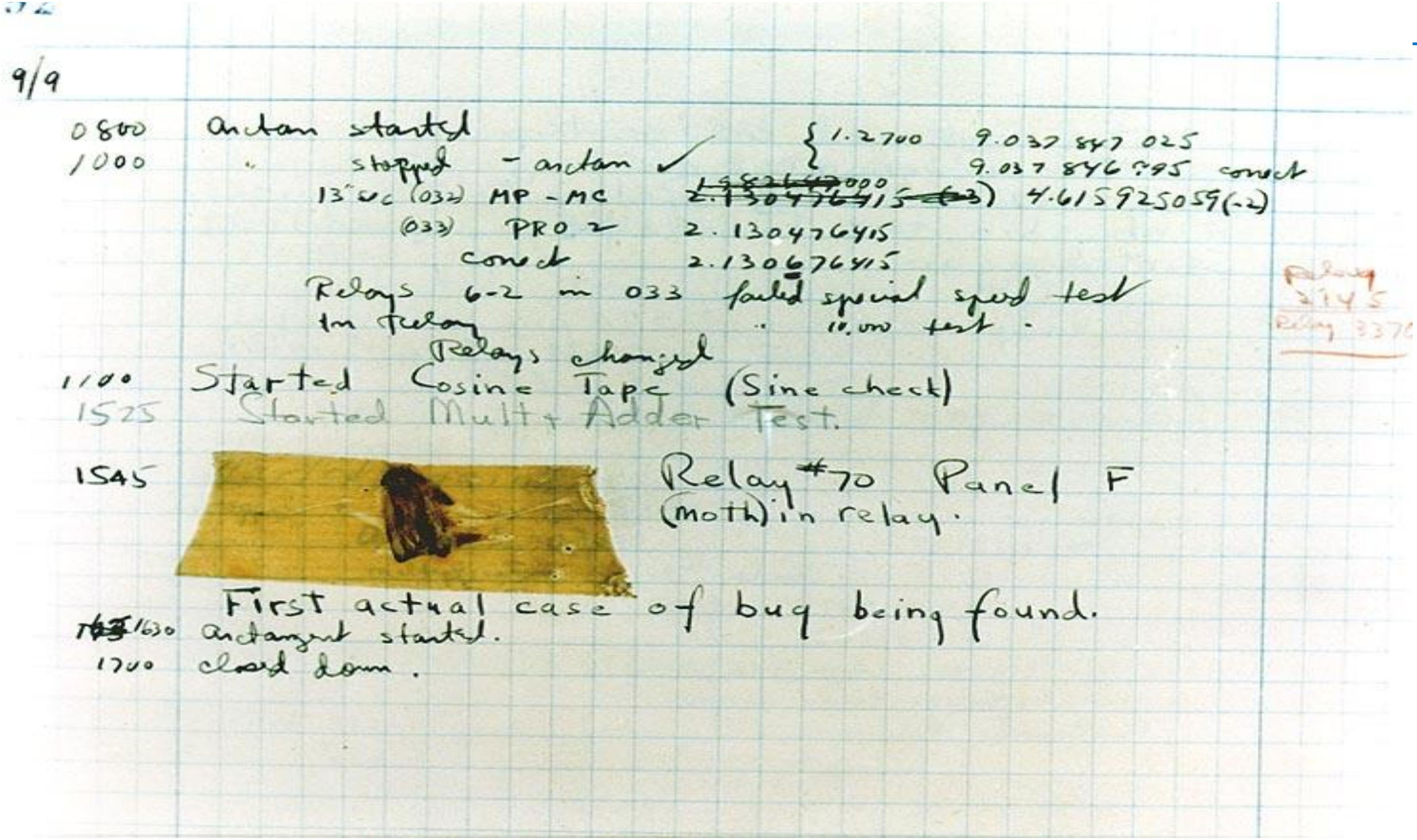
# Mark II logbook, Sep 9, 1947



# Debugging

CSE 403 Software Engineering

# Today's outline

- Debugging basics
- Delta debugging technique
  - In-class exercise on 11/15 will complement this material

**Background Reading**:
Simplifying and Isolating Failure-Inducing Input, Zeller and Hildebandt, 2002

# 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 ≠ 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 be systematic

- Carefully decide what to do (avoid fruitless avenues)
- Record everything that you do (actions and results)
  - Can replicate previous work
  - Or avoid the need to do so

Iterative scientific process:

Should have explanatory power
Not just “Maybe line 3 is wrong”

Formulate a hypothesis

Design an experiment

Should investigate cause

Perform the experiment

Interpret results

# 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**

# What's a good bug (issue) report look like?

**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"?

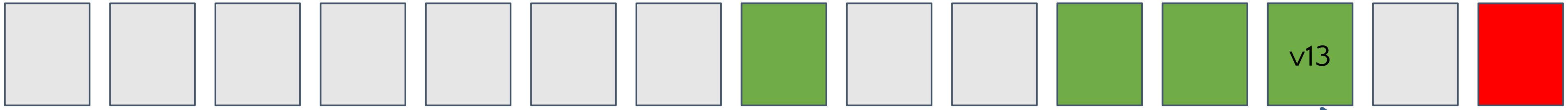
**A test case** should be as simple as possible

- Why?

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

Continuous integration runs:


Add a new test case:




After fix, passes

Still fails!

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.

**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>
</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="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>
</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:

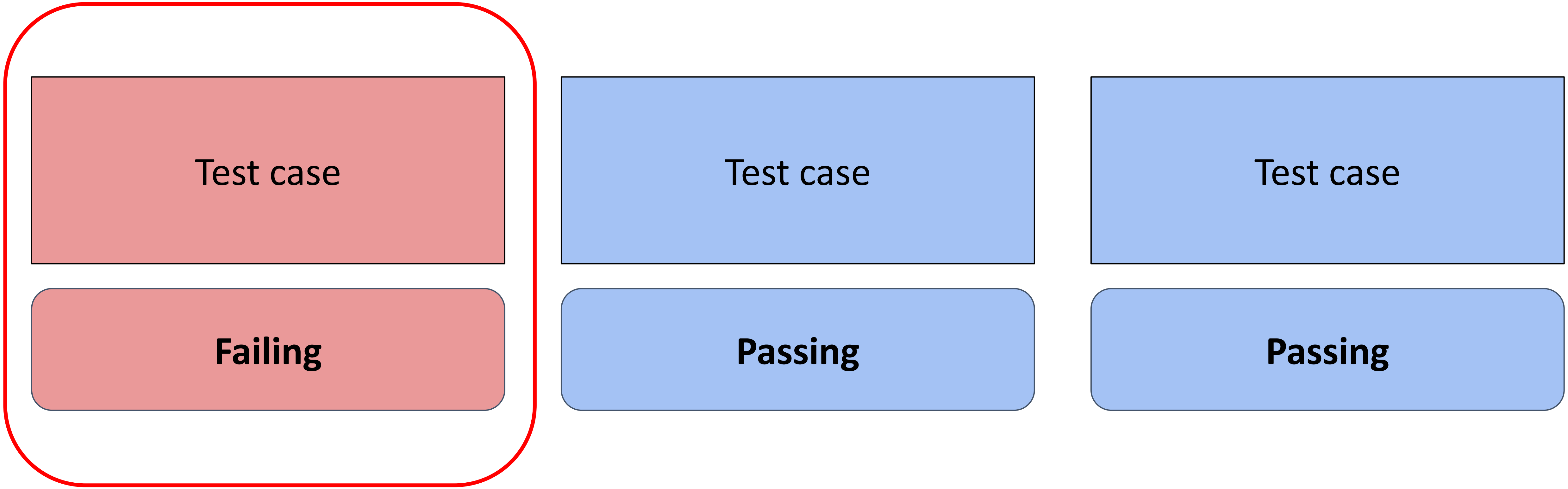
- Failing test case that is as small as possible

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

# Minimizing test cases

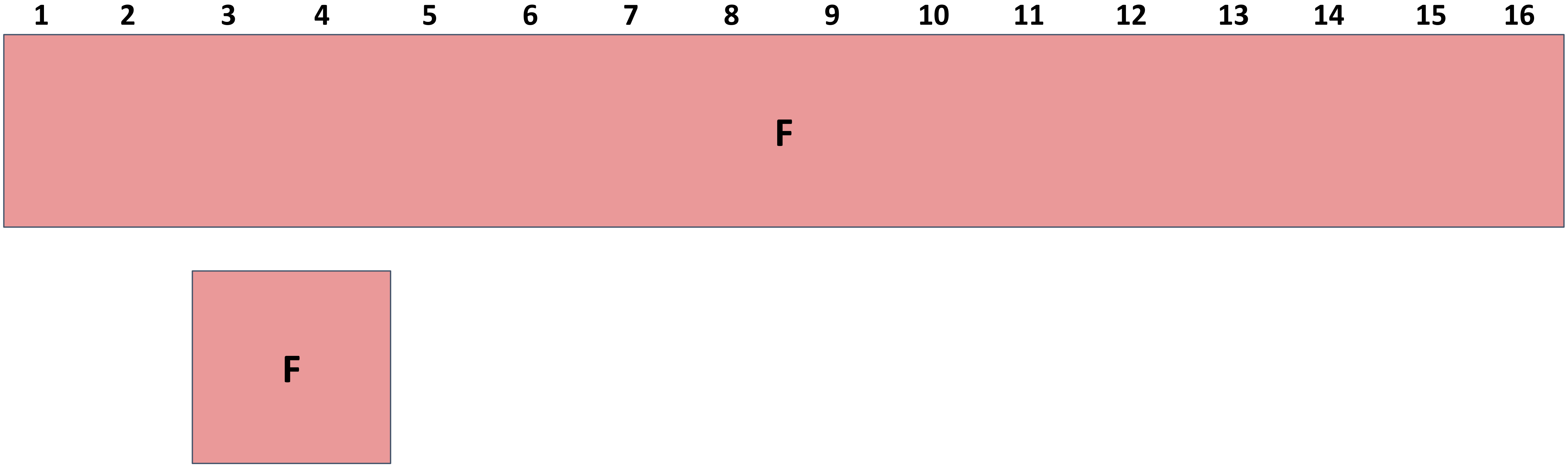
| | | |
|---|---|---|
| Test case | Test case | Test case |
| **Failing** | **Passing** | **Passing** |

# Minimizing test cases




**Goal: minimize the failing test case**

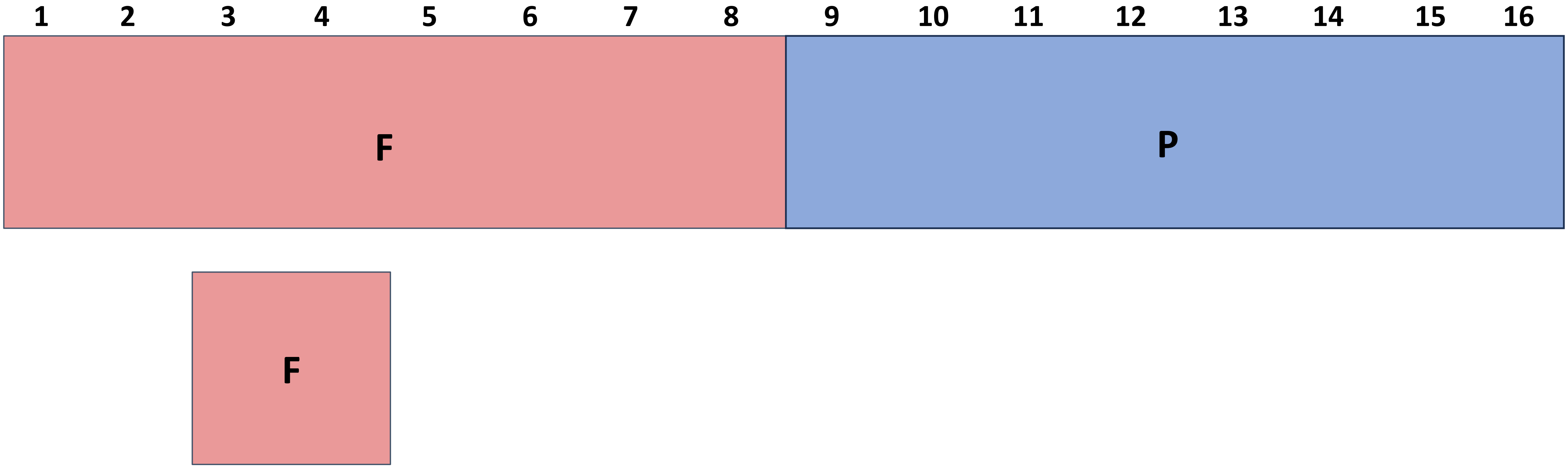
# The happy path: binary search



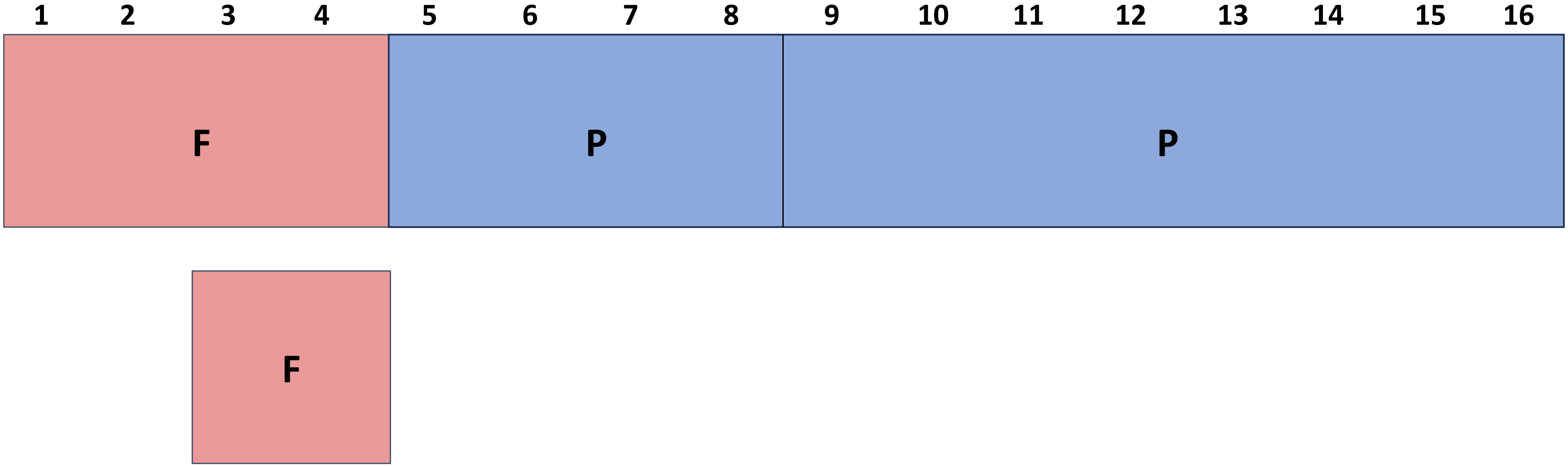

**Failing test with 16 lines**

**The minimal failing test has 2 lines: 3 and 4**

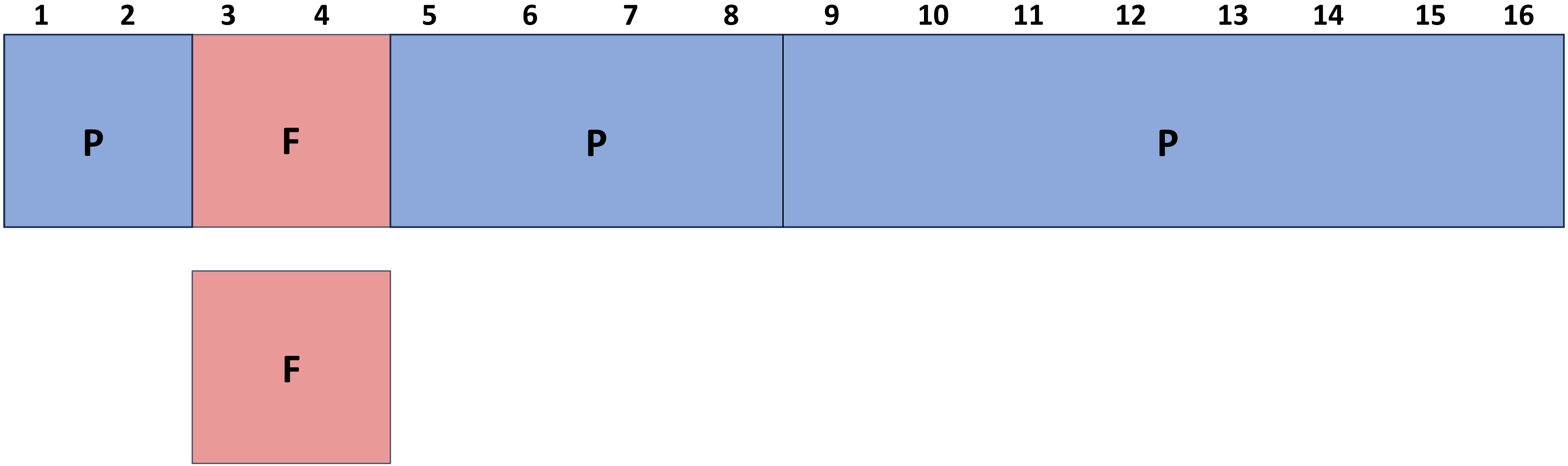
# The happy path: binary search



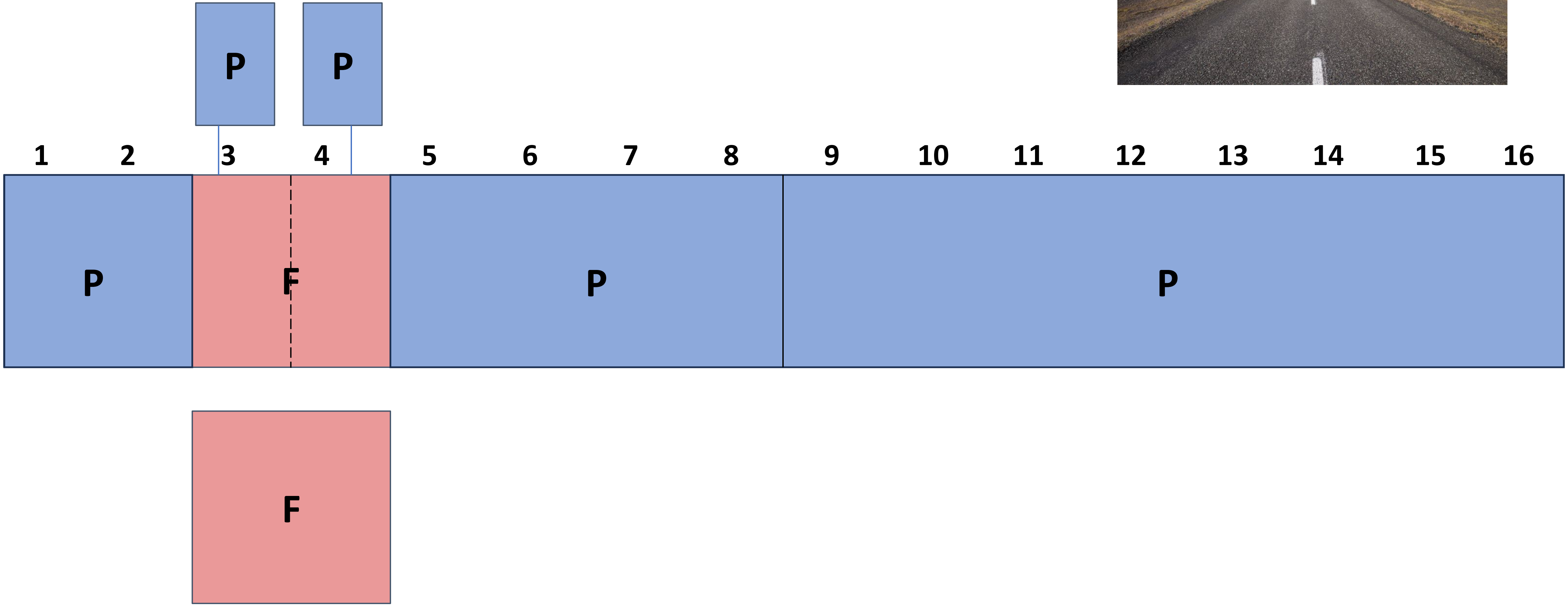
# The happy path: binary search



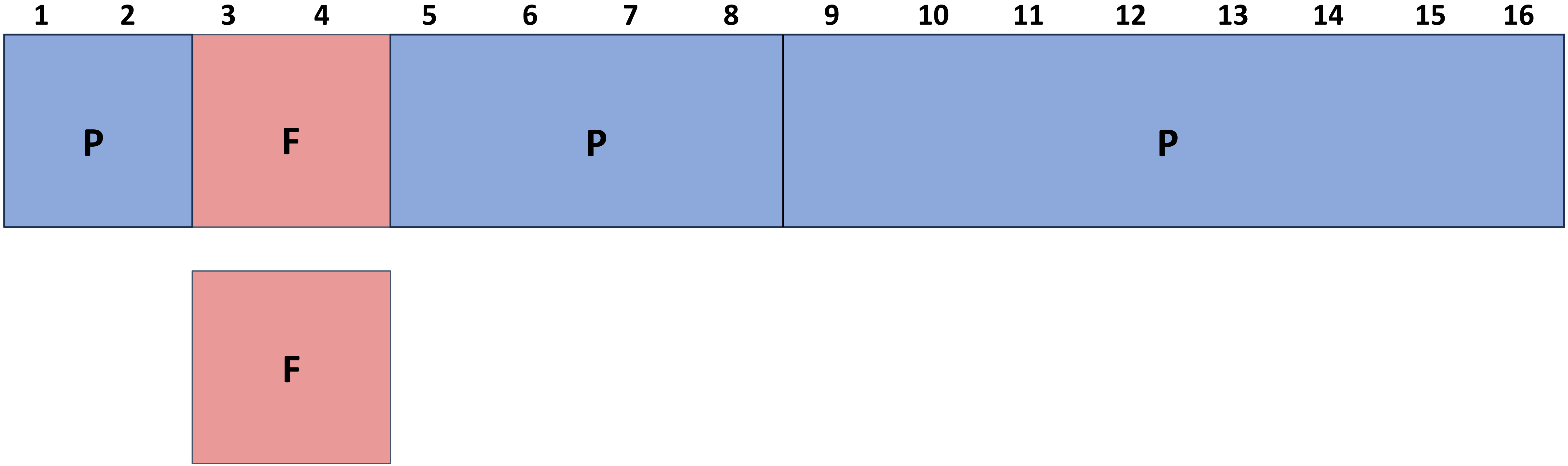
# The happy path: binary search



# The happy path: binary search

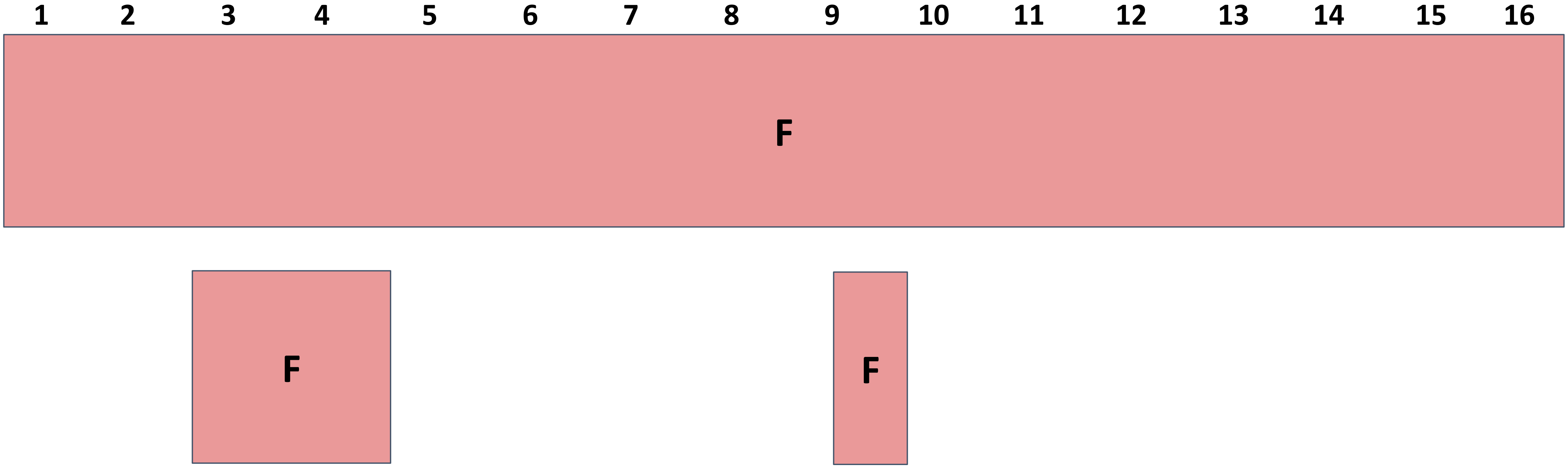


# The happy path: binary search




Successfully minimized the failing test to 2 lines

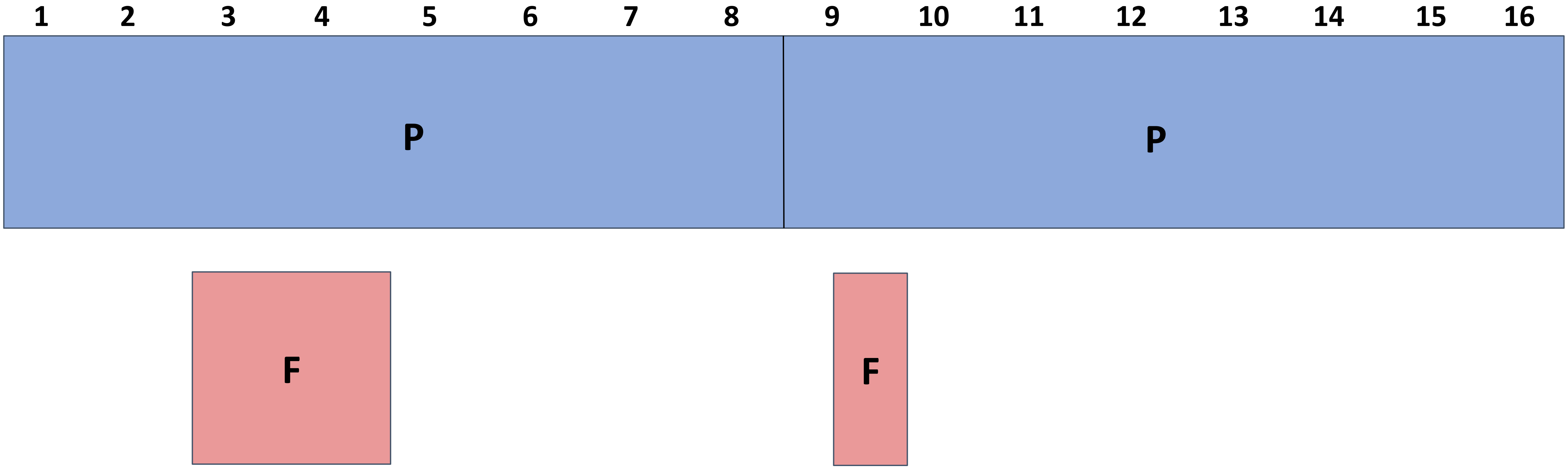
# The not so happy path...



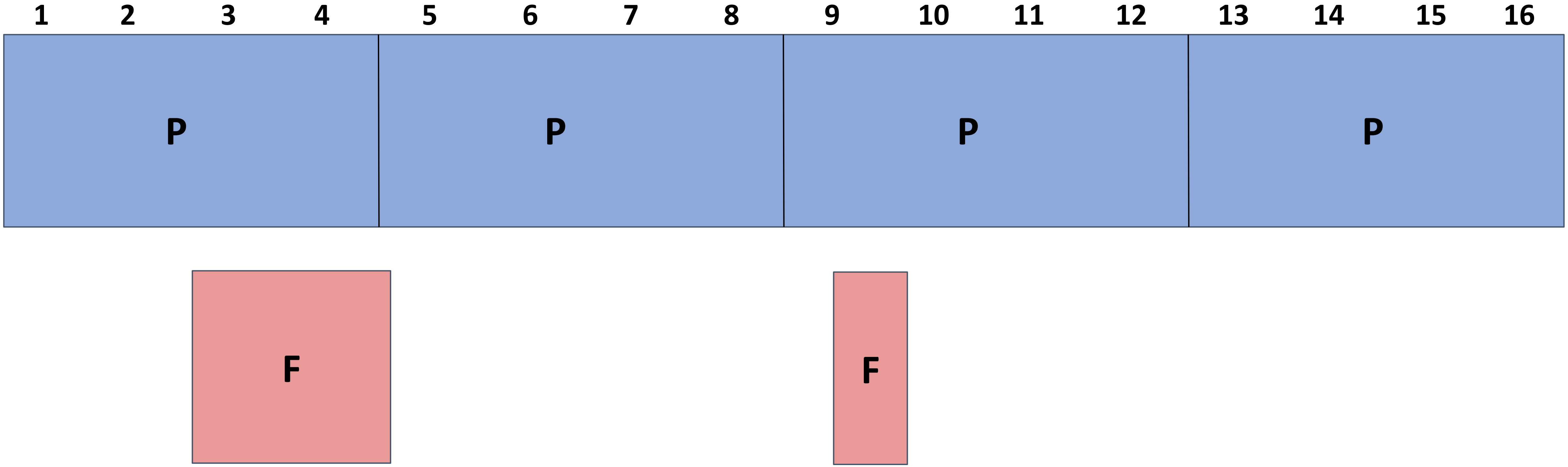

**Suppose the failure pattern is more complex**

**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 Hildebandt, 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*

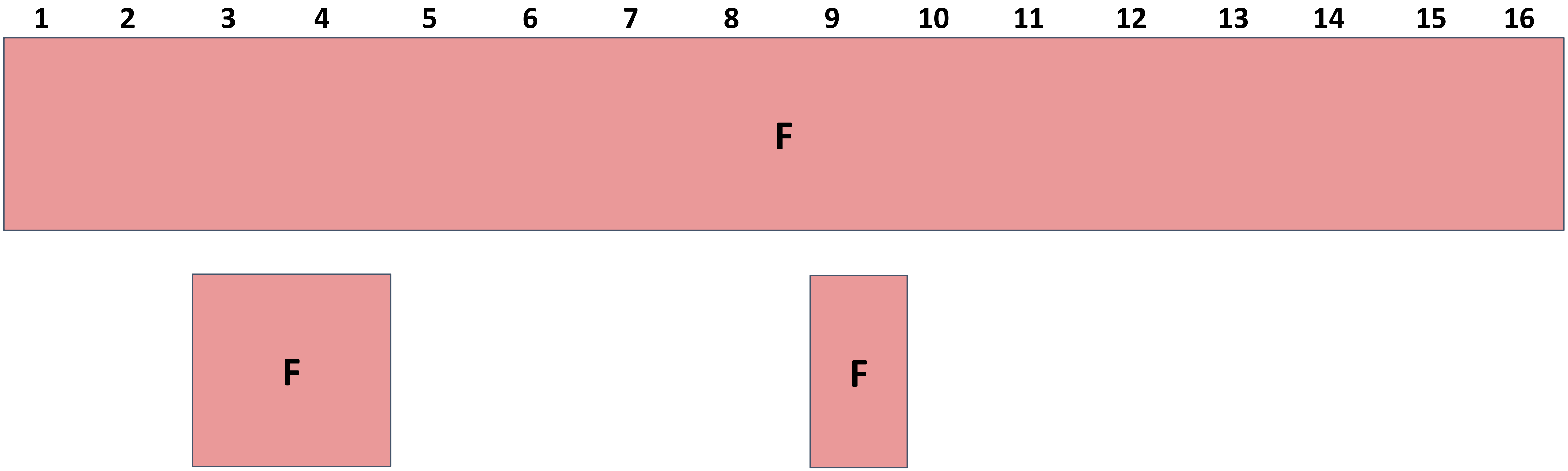
Let *test* and $c_{✘}$ be given such that $test(\emptyset) = ✔ \wedge test(c_{✘}) = ✘$ hold.
The goal is to find $c'_{✘} = ddmin(c_{✘})$ such that $c'_{✘} \subseteq c_{✘}$, $test(c'_{✘}) = ✘$, and $c'_{✘}$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_{✘}) = ddmin_2(c_{✘}, 2) \quad \text{where}$$

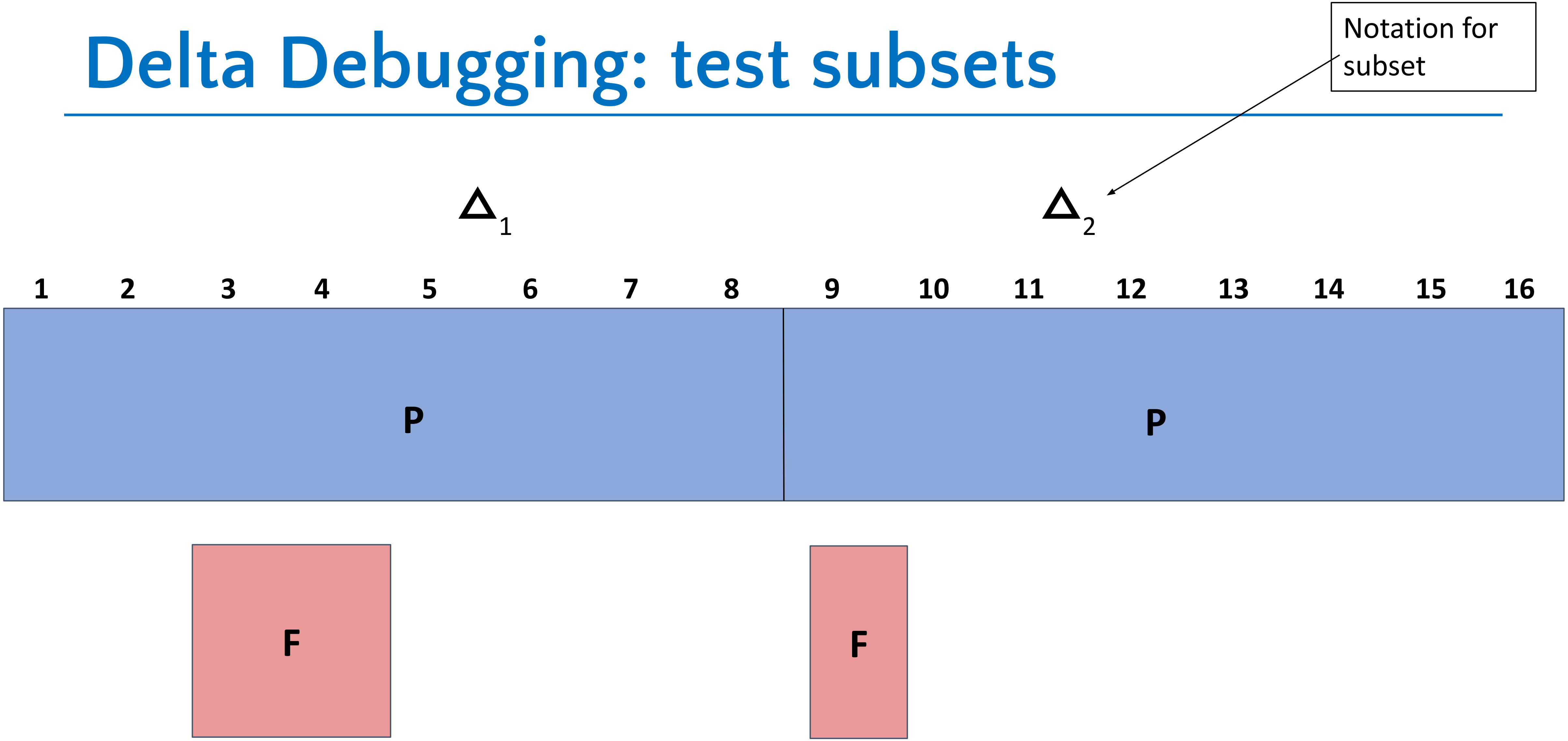
$$ddmin_2(c'_{✘}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = ✘ \text{ ("reduce to subset")} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = ✘ \text{ ("reduce to complement")} \\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

where $\nabla_i = c'_{✘} - \Delta_i$, $c'_{✘} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all $\Delta_i$ are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{✘}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{✘}) = ✘ \wedge n \leq |c'_{✘}|$.

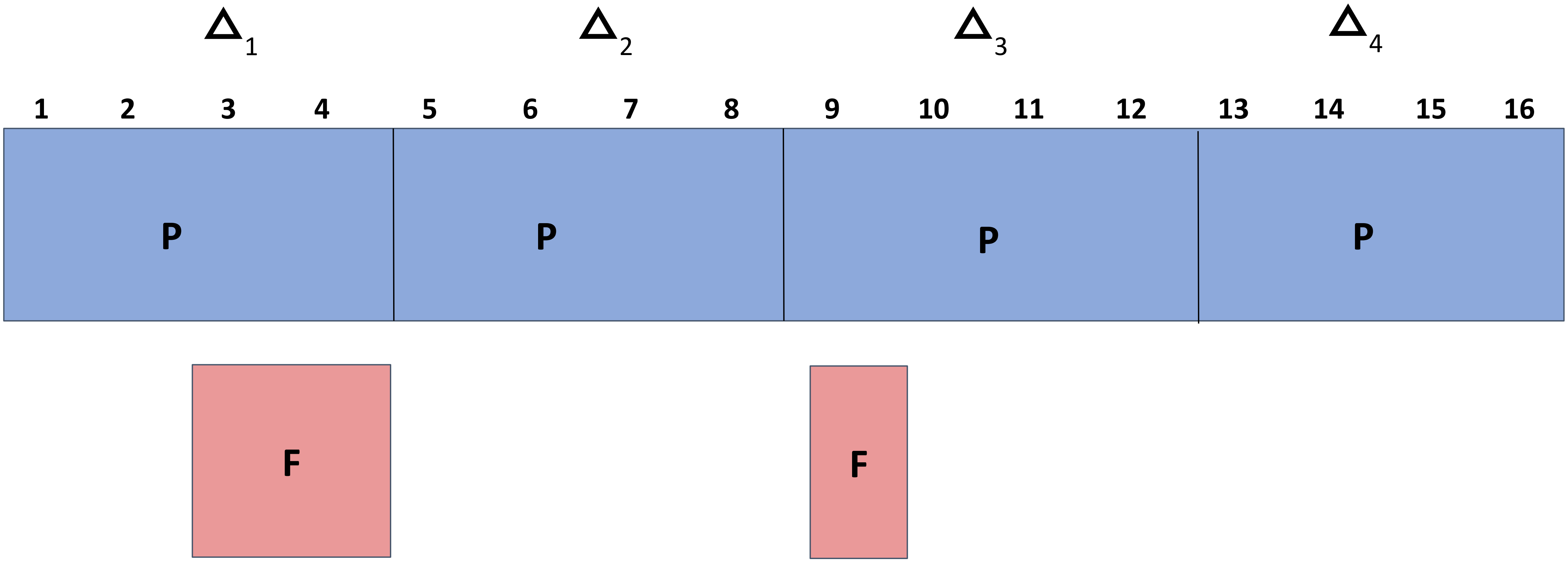
# Delta Debugging is mostly binary search



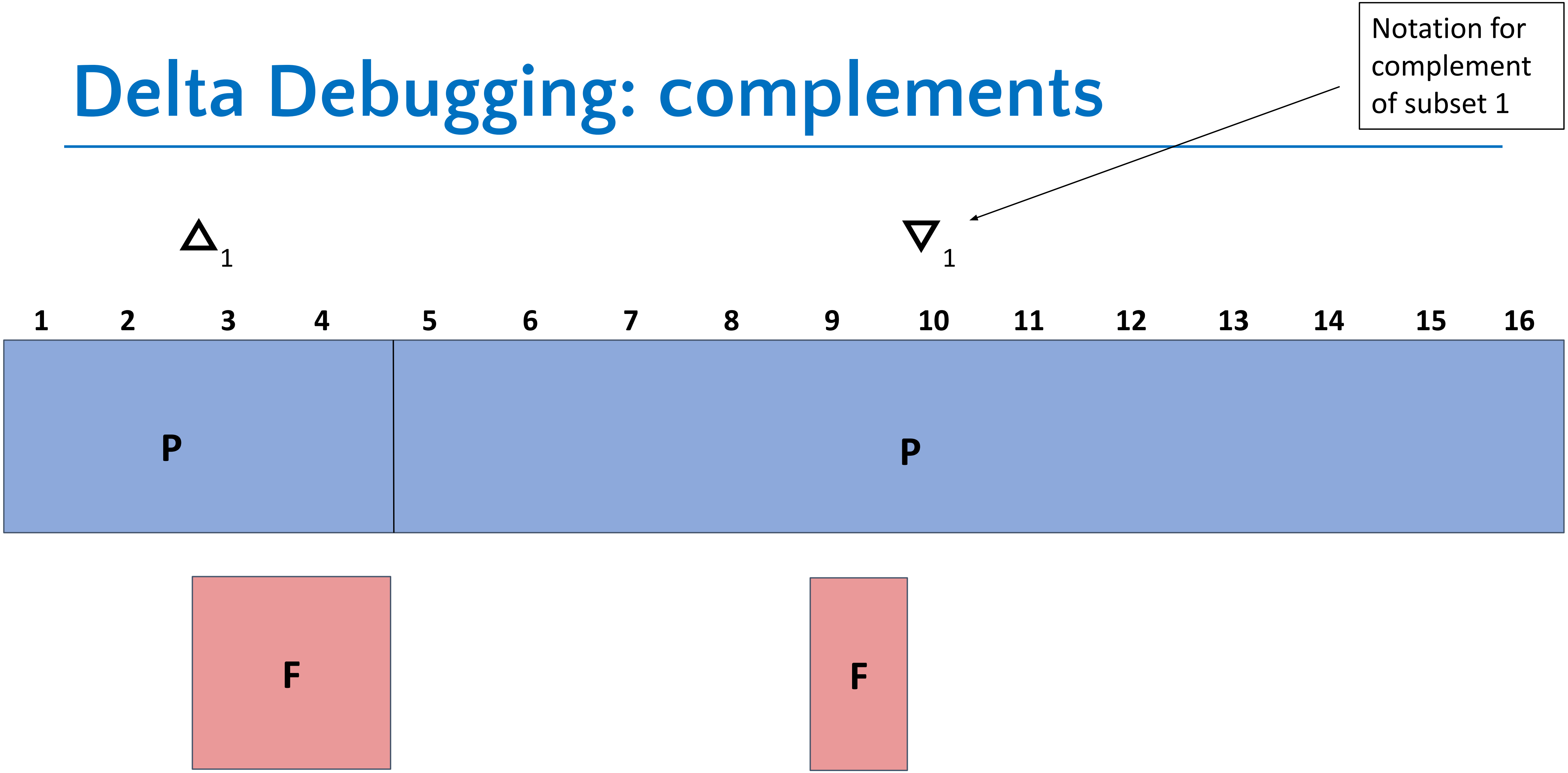
# Delta Debugging: test subsets



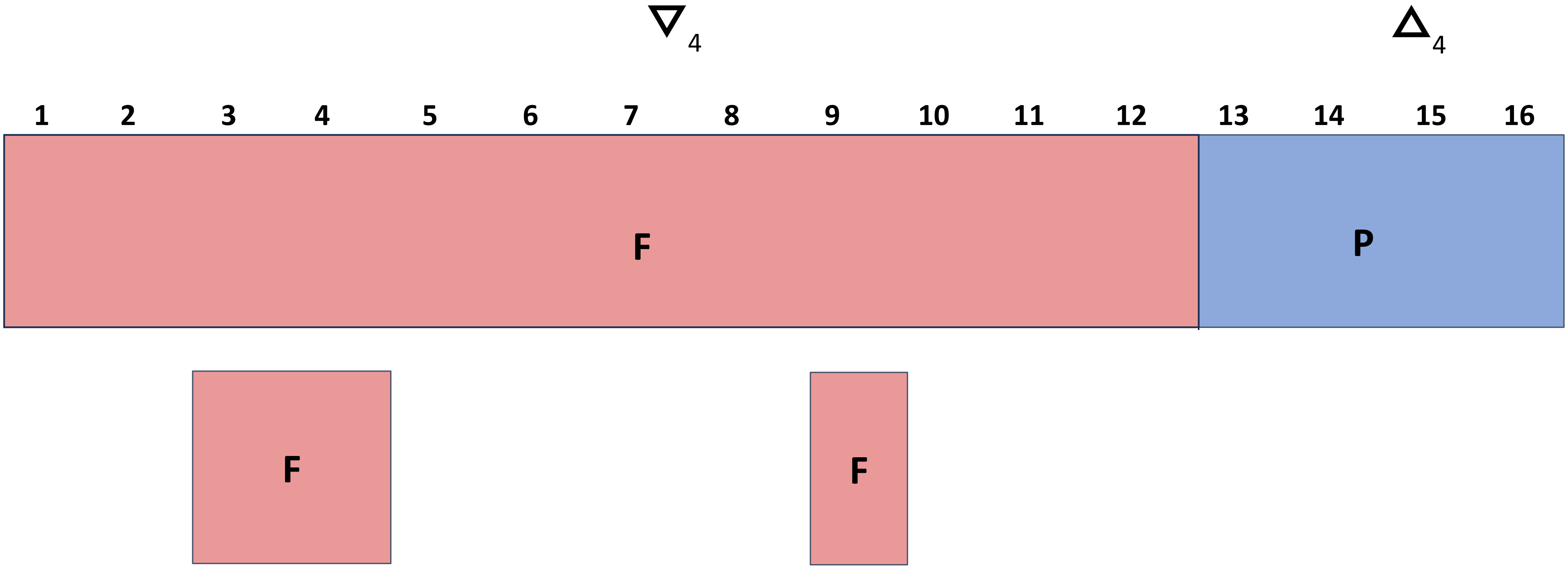
# Delta Debugging: increase granularity



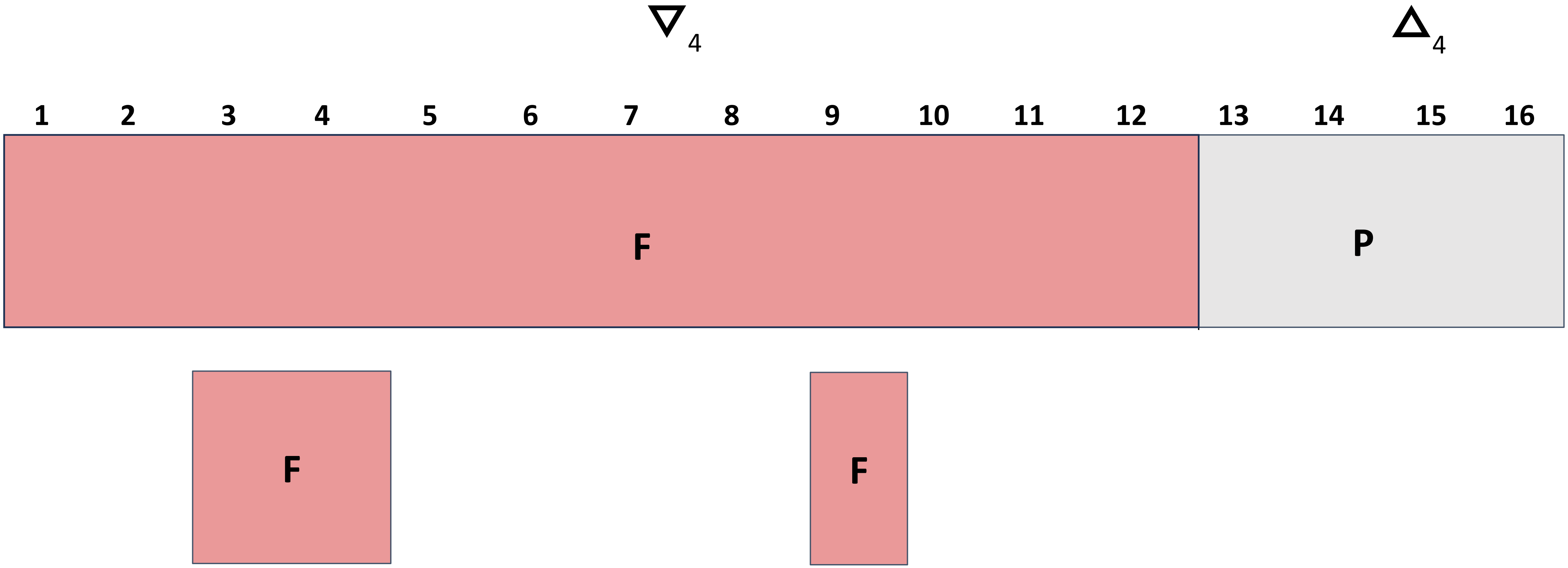
# Delta Debugging: complements



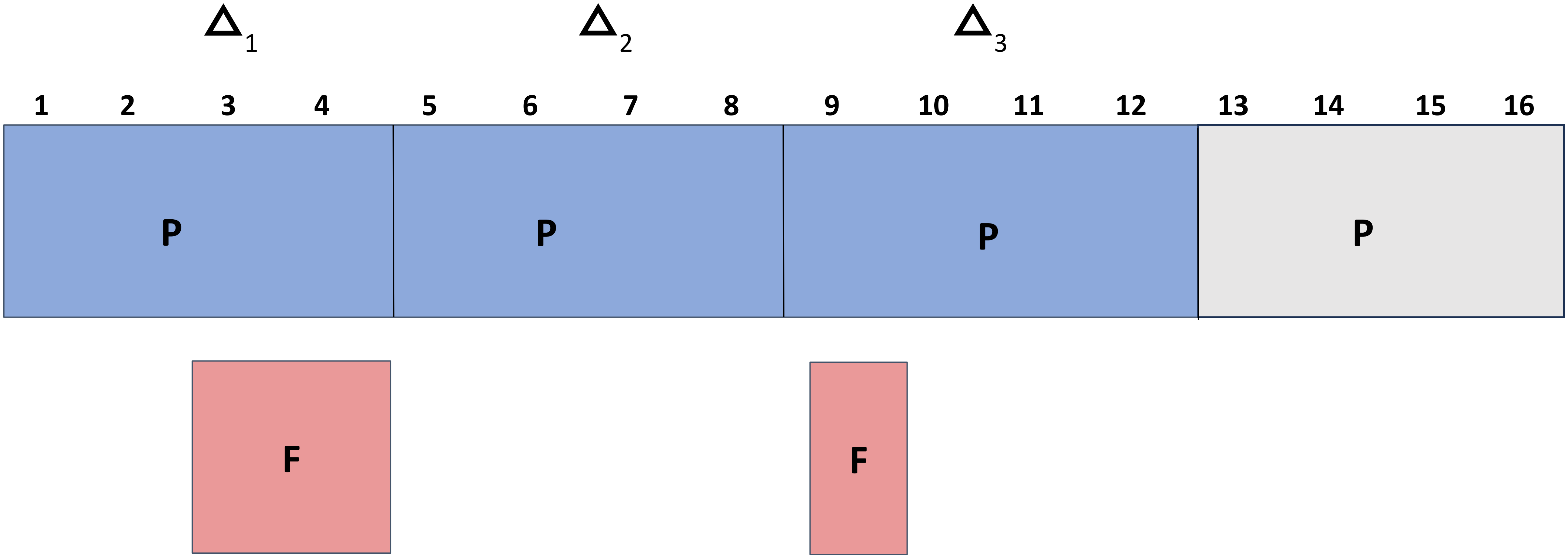
# Delta Debugging: complements



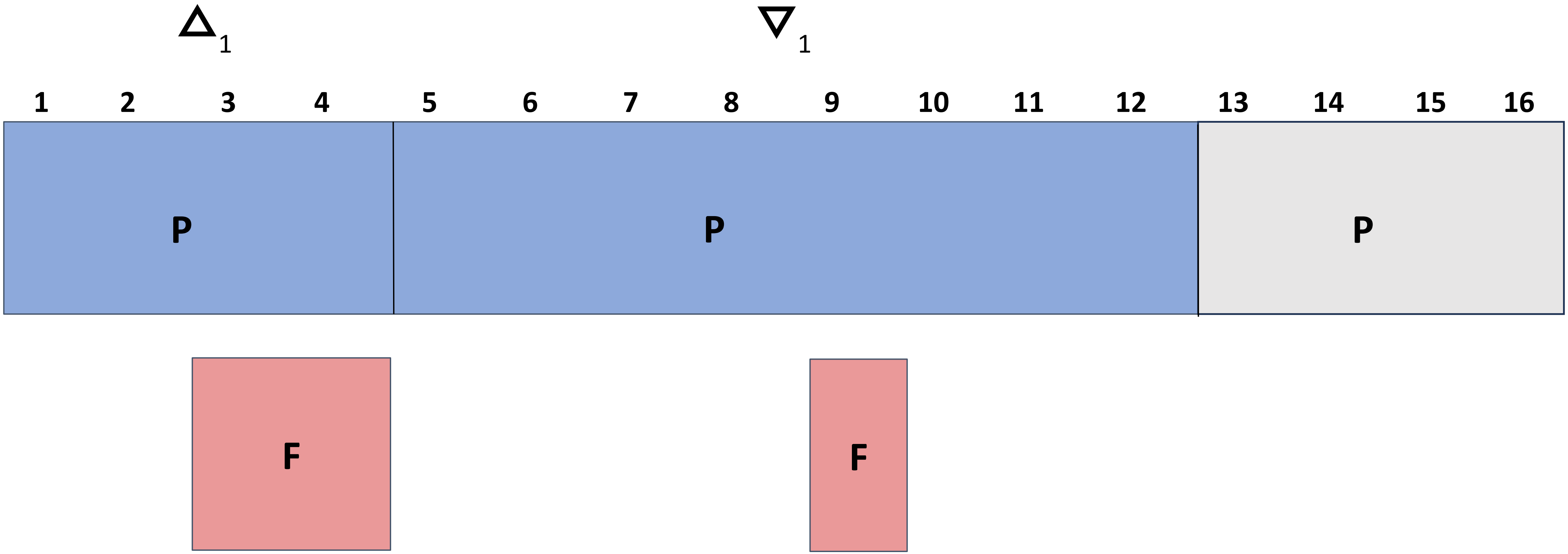
# Delta Debugging: reduce



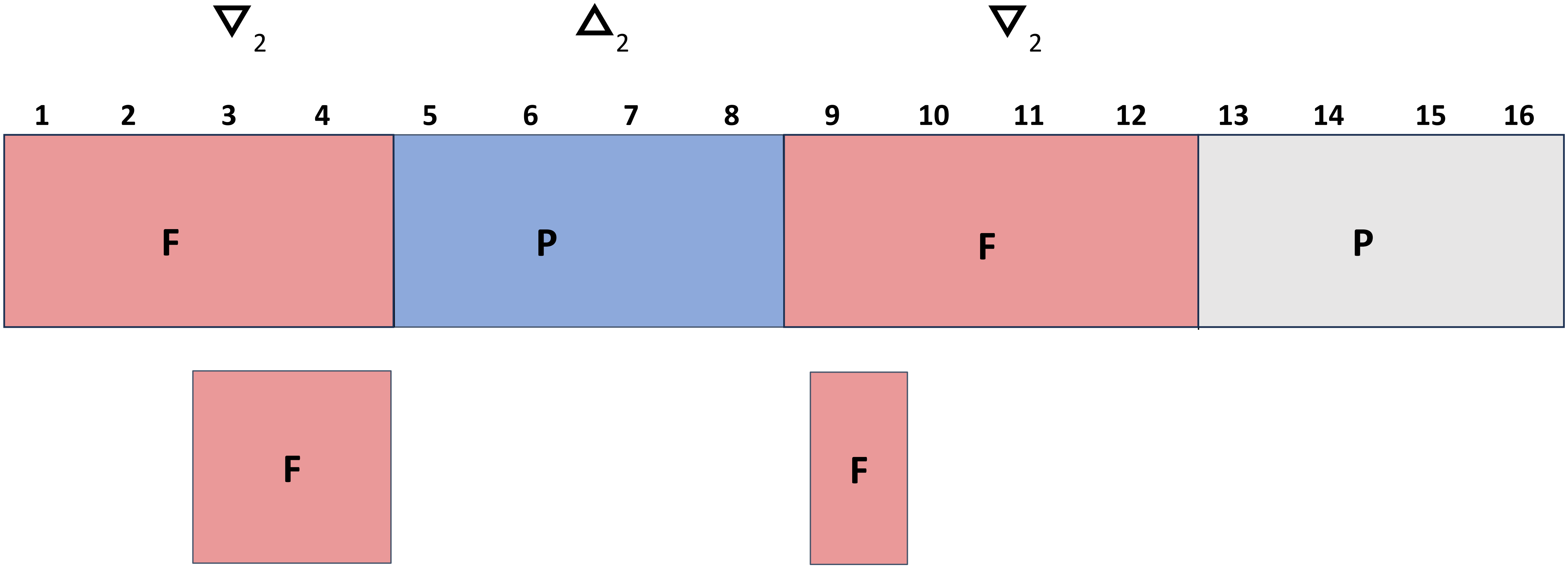
# Delta Debugging: test subsets



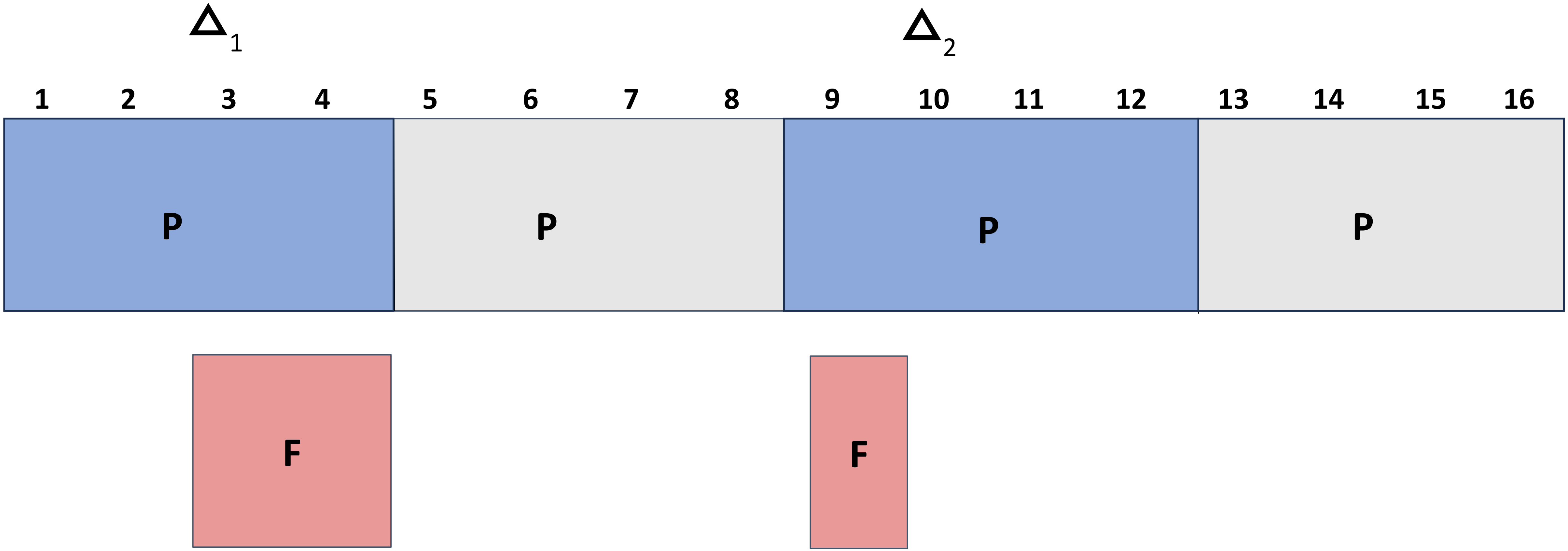
# Delta Debugging: complements



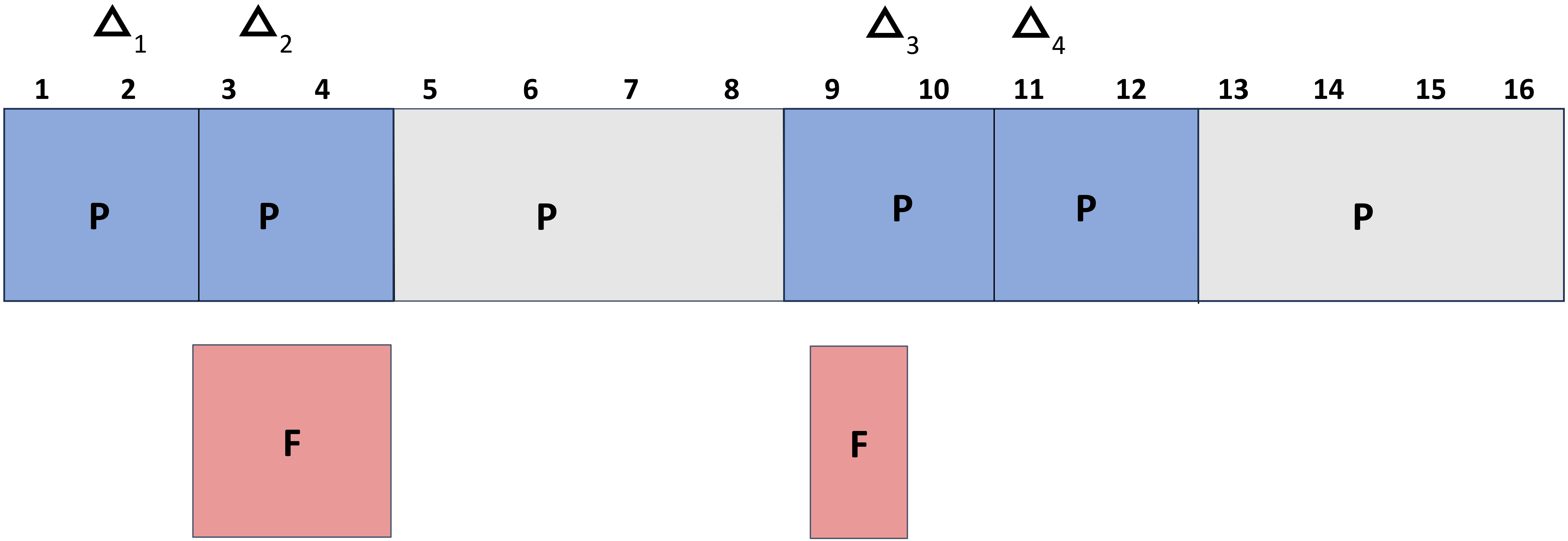
# Delta Debugging: complements



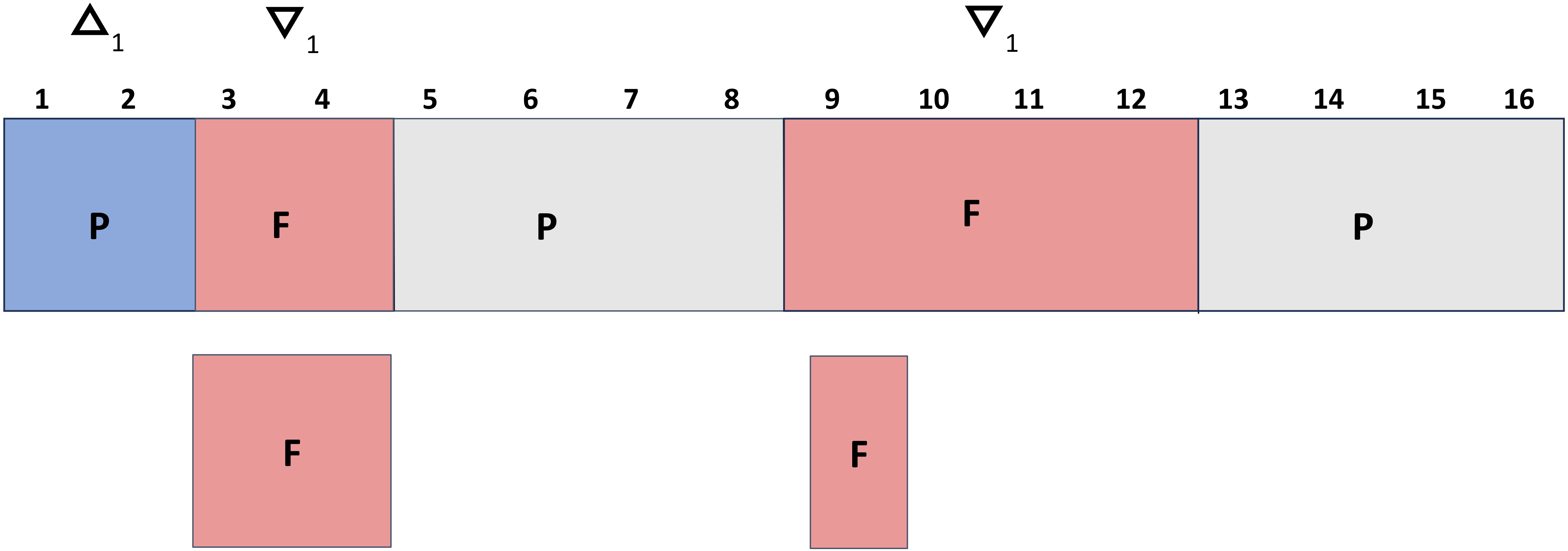
# Delta Debugging: reduce



# Delta Debugging: increase granularity

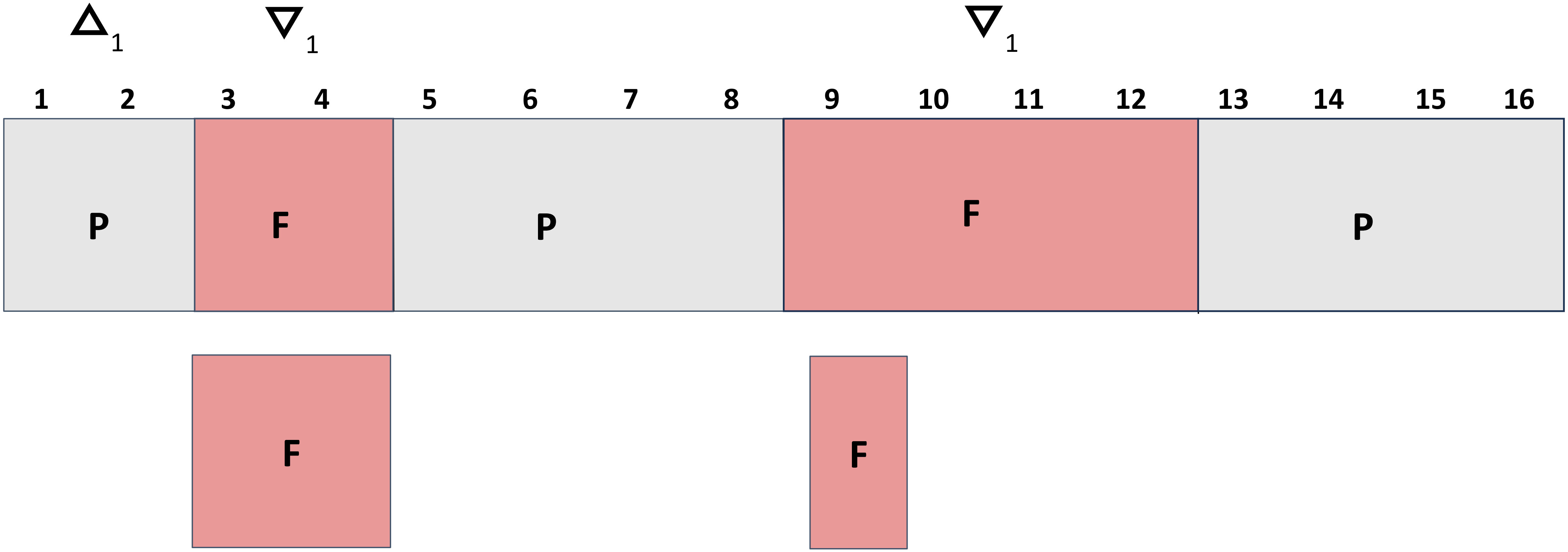


# Delta Debugging: complements



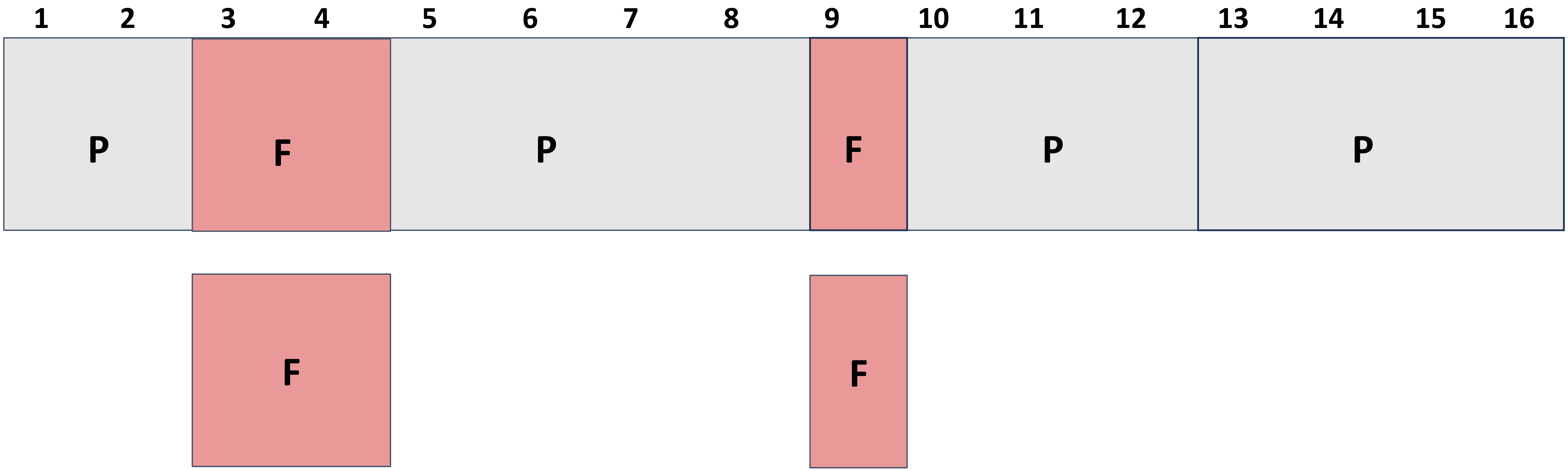

And so on...

# Delta Debugging: reduction




And so on...

# Delta Debugging finds a “1-minimal” solution




**Failing test cases must be deterministic and monotone**

# Delta debugging: one more example

# Let's try one more

**Program and initial test case**

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

**Complete the following table**

| Iteration | n | input | $\Delta_1, \ldots, \Delta_n$ $\nabla_1, \ldots, \nabla_n$ |
|---|---|---|---|
| 1 | 2 | 2424 | … |
| 2 | … | … | … |

# Let's try one more

**Program and initial test case**

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

**Complete the following table**

| Iteration | n | input | $\triangle_1, \ldots, \triangle_n$<br>$\nabla_1, \ldots, \nabla_n$ |
|---|---|---|---|
| 1 | 2 | 2424 | 24, (24) |
| 2 | 4 | 2424 | 2, 4, (2), (4), **424**, (224), (244), (242) |
| 3 | 3 | 424 | (4), (2), (4), (24), 44, **42** |
| 4 | 2 | 42 | (4),( 2) |