Quotes about loops

“O! Thou hast damnable iteration and art, indeed, able to corrupt a saint.” Shakespeare, *Henry IV*, Pt I, 1 ii

“Use not vain repetition, as the heathen do.”
Matthew V, 48

Your “if” is the only peacemaker; much virtue in “if”.
Shakespeare, *As You Like It*.

ASYMPTOTIC COMPLEXITY
SEARCHING/SORTING

Lecture 10
CS2110 – Spring 2017

What Makes a Good Algorithm?

Suppose you have two possible algorithms that do the same thing; which is *better*?

What do we mean by *better*?

- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?

How do we measure time andspace of an algorithm?

Your time is most important!

FIRST, Aim for simplicity, ease of understanding, correctness.

SECOND, Worry about efficiency only when it is needed.

Basic Step: one "constant time" operation

Constant time operation: its time doesn't depend on the size or length of anything. Always roughly the same. Time is bounded above by some number

Basic step:

- Input/output of a number
- Access value of primitive-type variable, array element, or object field
- assign to variable, array element, or object field
- do one arithmetic or logical operation
- method call (not counting arg evaluation and execution of method body)

Basic Step: one “constant time” operation. Example of counting basic steps in a loop

```
// Store sum of 1..n in sum
sum= 0;
// inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1)
    sum= sum + n
```

All operations are basic steps, take constant time.
There are n loop iterations.
Therefore, takes time proportional to n.
Linear algorithm in n

Statement/ expression	Number of times done
`sum= 0;`	1
`k= 1;`	1
`k <= n`	n+1
`k= k+1;`	n
`sum= sum + n;`	n
Total basic steps executed	3n + 3

Basic Step: one “constant time” operation

```
// Store sum of 1..n in sum
sum= 0;
// inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1)
    sum= sum + n
```

All operations are basic steps, take constant time.
There are n loop iterations.
Therefore, takes time proportional to n.
Linear algorithm in n

```
// Store n copies of ‘c’ in s
s= “”;
// inv: s contains k-1 copies of ‘c’
for (int k= 1; k = n; k= k+1)
    s=  s + ‘c’;
```

All operations are basic steps, except for catenation. For each k, catenation creates and fills k array elements. Total number created:
$1 + 2 + 3 + \cdots + n$, or
$n(n+1)/2 = n*n/2 + 1/2$
Quadratic algorithm in n

Linear versus quadractic

```
// Store sum of 1..n in sum
sum= 0;
// inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1)
    sum= sum + n
```

Linear algorithm

```
// Store n copies of ‘c’ in s
s= “”;
// inv: s contains k-1 copies of ‘c’
for (int k= 1; k = n; k= k+1)
    s=  s + ‘c’;
```

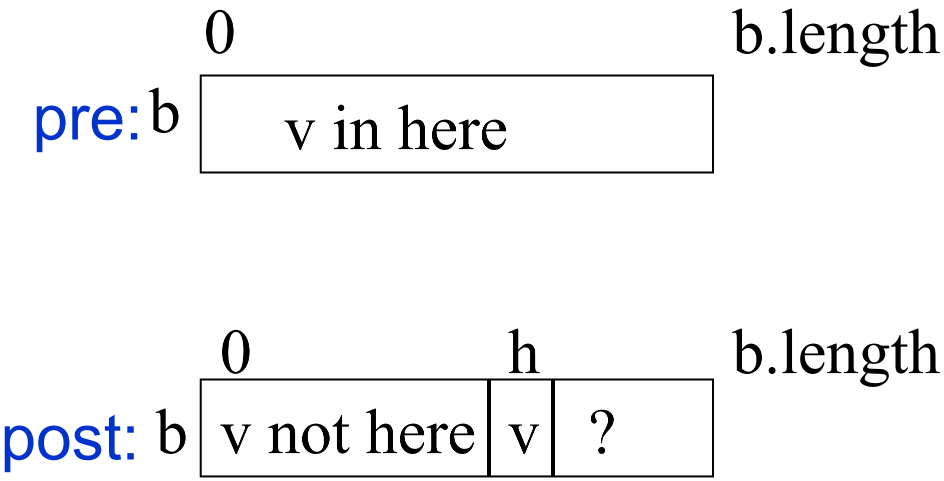
Quadratic algorithm

In comparing the runtimes of these algorithms, the exact number of basic steps is not important. What’s important is that

One is linear in n —takes time proportional to n

One is quadratic in n —takes time proportional to n^2

Linear search for v in b[0..]



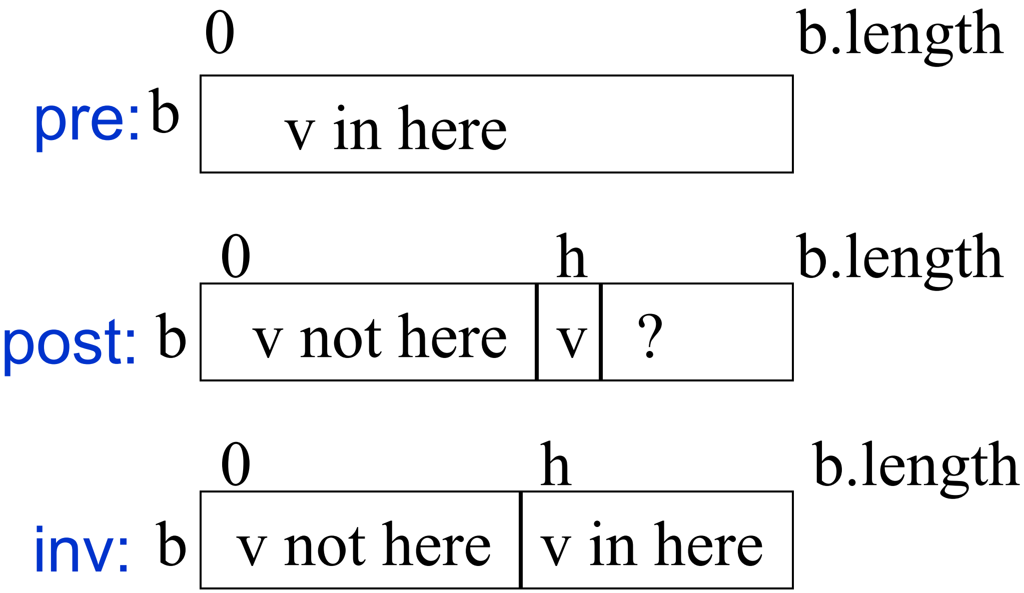
Methodology:

1. Draw the invariant as a combination of pre and post
2. Develop loop using 4 loopy questions.

Practice doing this!

Once you get the knack of doing this, you will *know* these algorithms not because you memorize code but because you can develop them at will from the pre- and post-conditions

Linear search for v in b[0..]



pre: b

0 b.length

v in here

post: b

0 h b.length

v not here | v | ?

inv: b

0 h b.length

v not here | v in here

```
h= 0;
while ( b[h] != v ) {
    h= h+1;
}
```

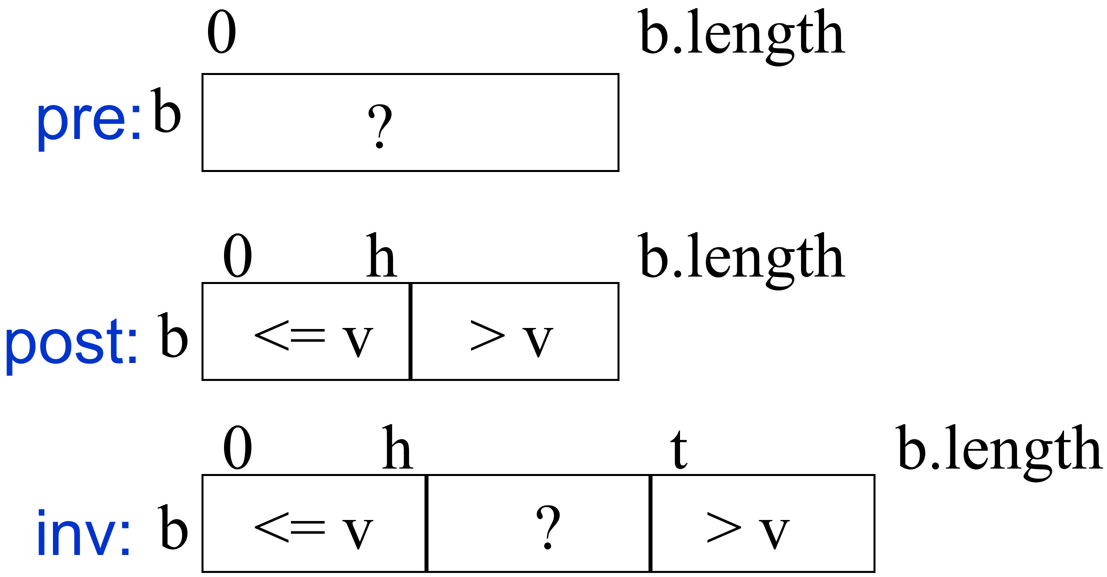
Each iteration takes constant time.

In the worst case, requires b.length iterations.

Worst case time: proportional to b.length.

Average (expected) time: A little statistics tells you b.length/2 iterations, still proportional to b.length

Linear search as in problem set: **b is sorted**



pre: b — 0 … b.length: ?

post: b — 0, h … b.length: <= v | > v

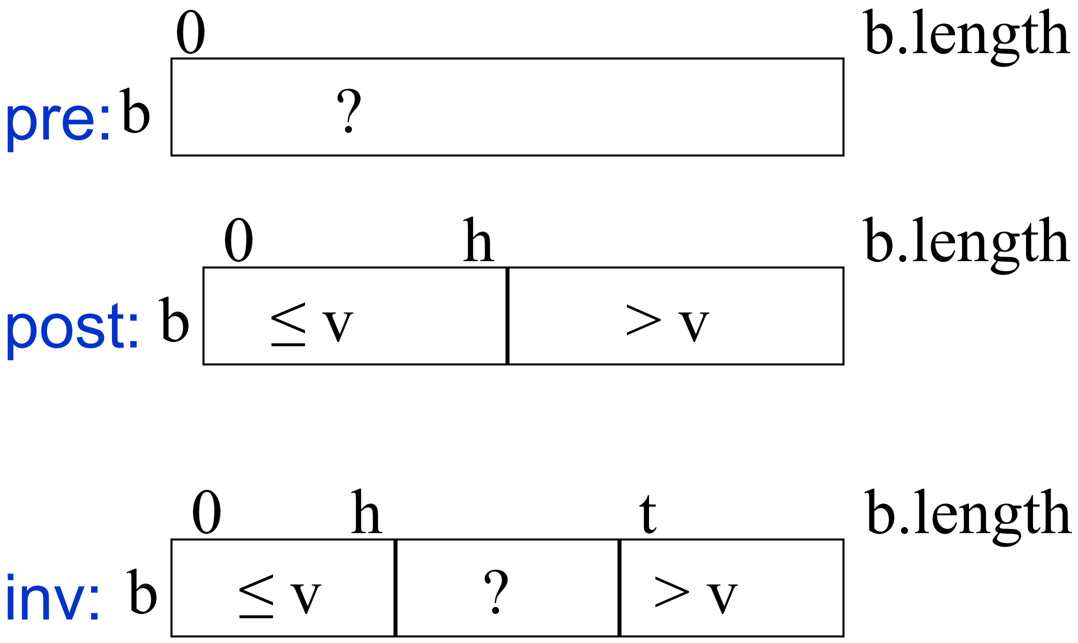
inv: b — 0, h, t … b.length: <= v | ? | > v

```
h= –1;  t= b.length;
while  ( h+1 != t ) {
    if (b[h+1] <= v)
        h=  h+1;
    else  t= h+1;
}
```

b[0] > v?   one iteration.

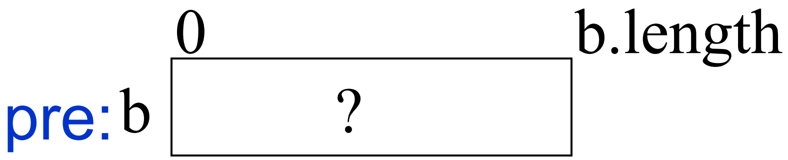
b[b.length-1] ≤ v?   b.length iterations
Worst case time: proportional to size of b

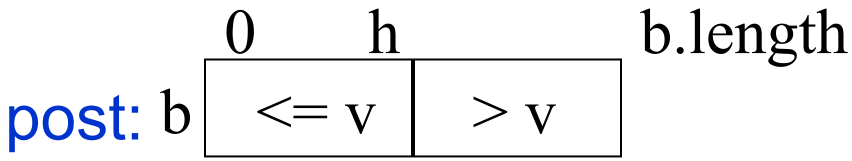
b is sorted ---use a binary search?

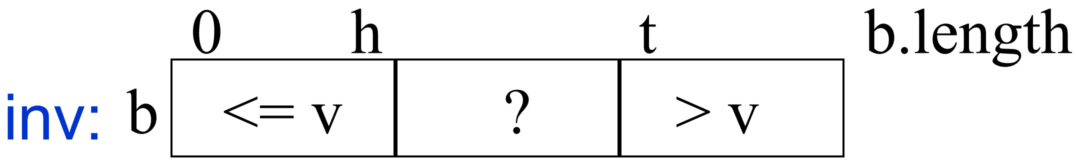


Since b is sorted, can cut ? segment in half.
As in a dictionary search

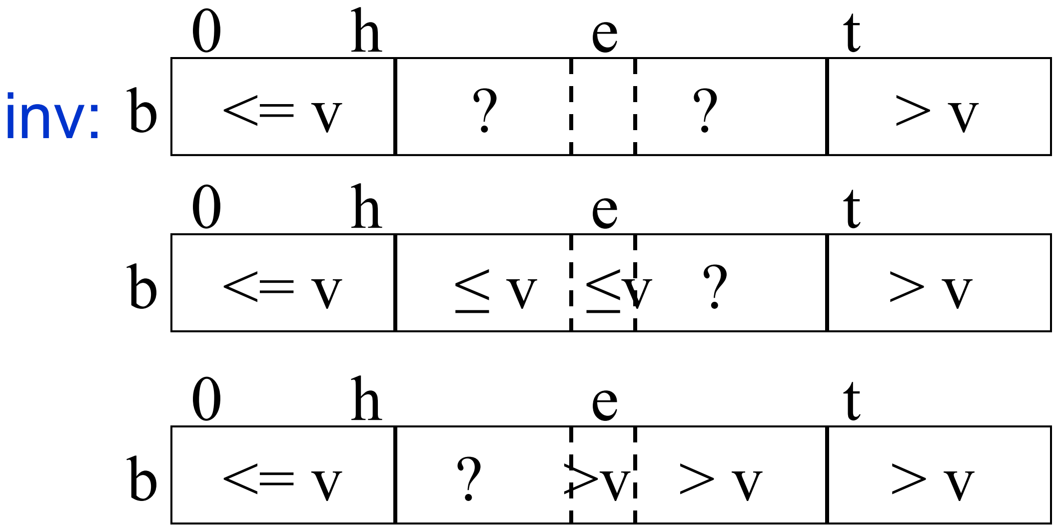
Binary search for v in b: b is sorted



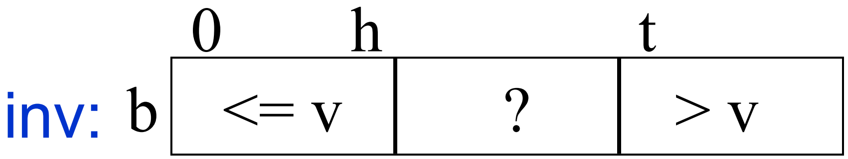




```
h= –1;  t= b.length;
while  (h != t–1) {
      int e= (h + t) / 2;
      // h < e < t
      if (b[e] <= v)  h= e;
      else t= e;
}
```

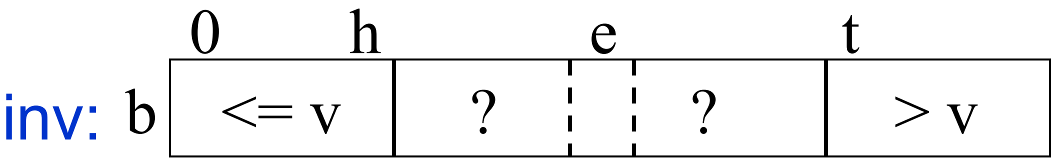


Binary search: an O(log n) algorithm



b.length = n

```
h= –1;  t= b.length;
while  (h != t–1) {
    int  e=  (h+t)/2;
    if (b[e] <= v)  h=  e;
    else  t=  e;
}
```



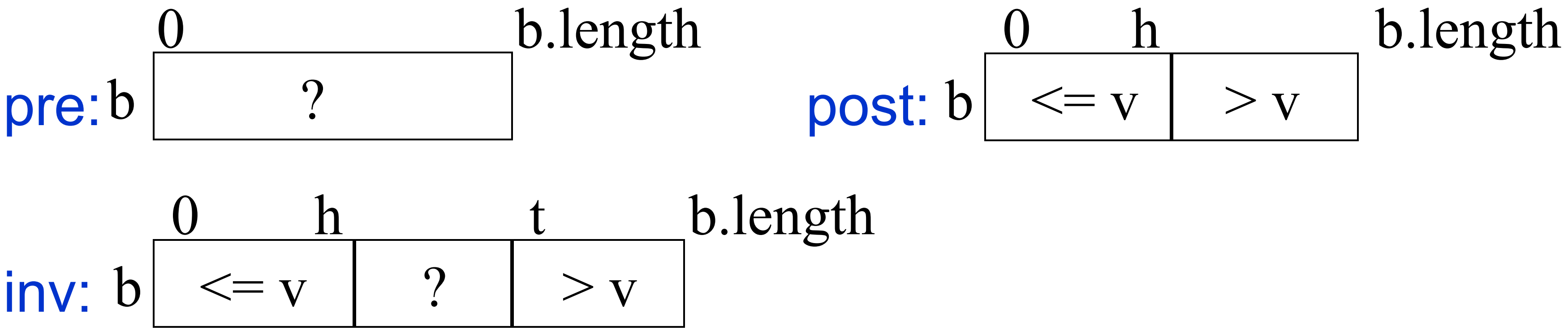
n = 2**k ?   About k iterations

Each iteration cuts the size of the ? segment in half.

Time taken is proportional to k, or log n.

A logarithmic algorithm

Binary search for v in b: b is sorted



0 b.length

pre: b ?

0 h b.length

post: b <= v > v

0 h t b.length

inv: b <= v ? > v

```
h= –1;  t= b.length;
while  (h != t–1) {
        int e= (h + t) / 2;
        // h < e < t
        if (b[e] <= v)  h= e;
        else t= e;
}
```

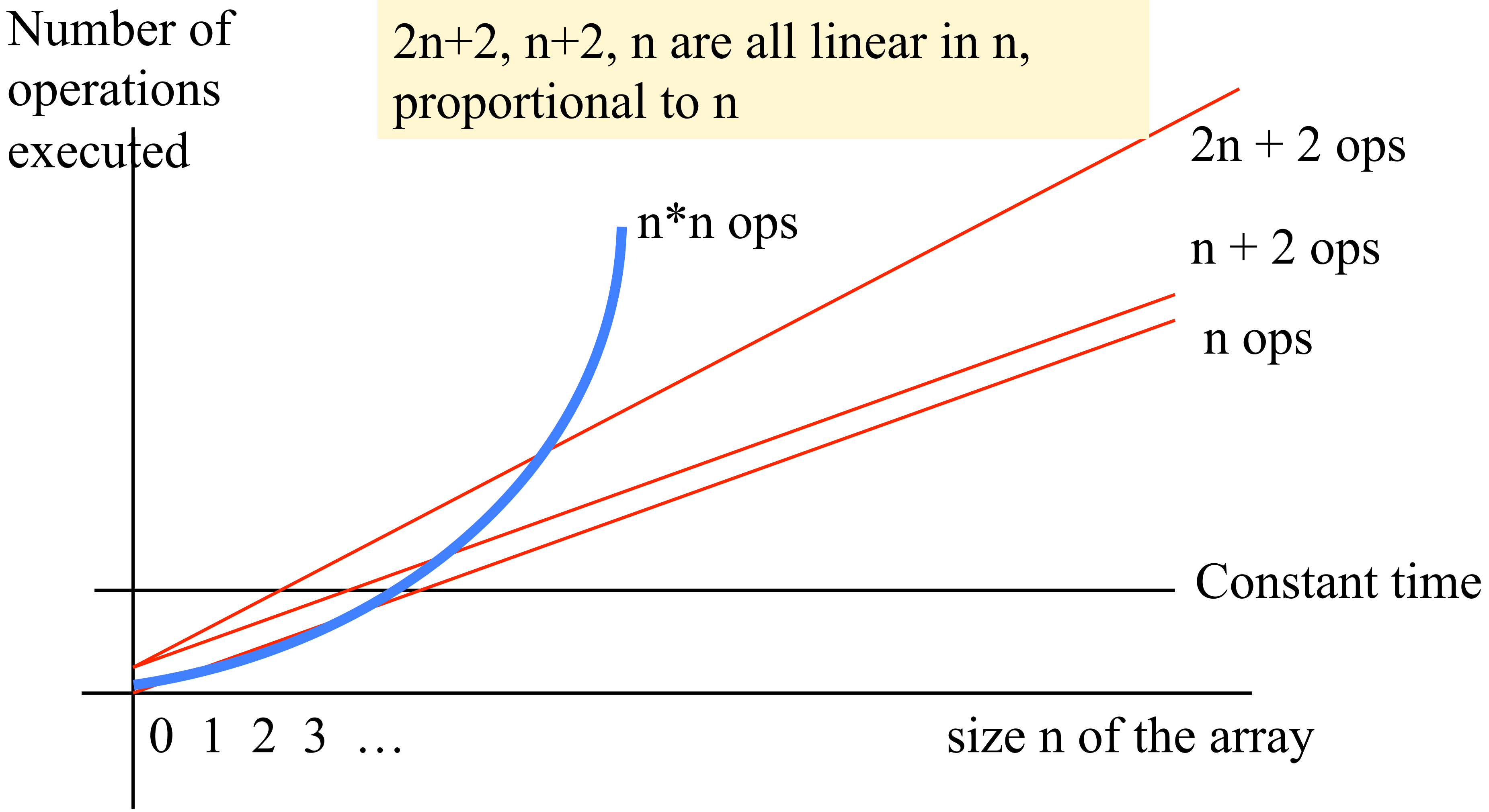
This algorithm is better than binary searches that stop when v is found.

1. Gives good info when v not in b.
2. Works when b is empty.
3. Finds rightmost occurrence of v, not arbitrary one.
4. Correctness, including making progress, easily seen using invariant

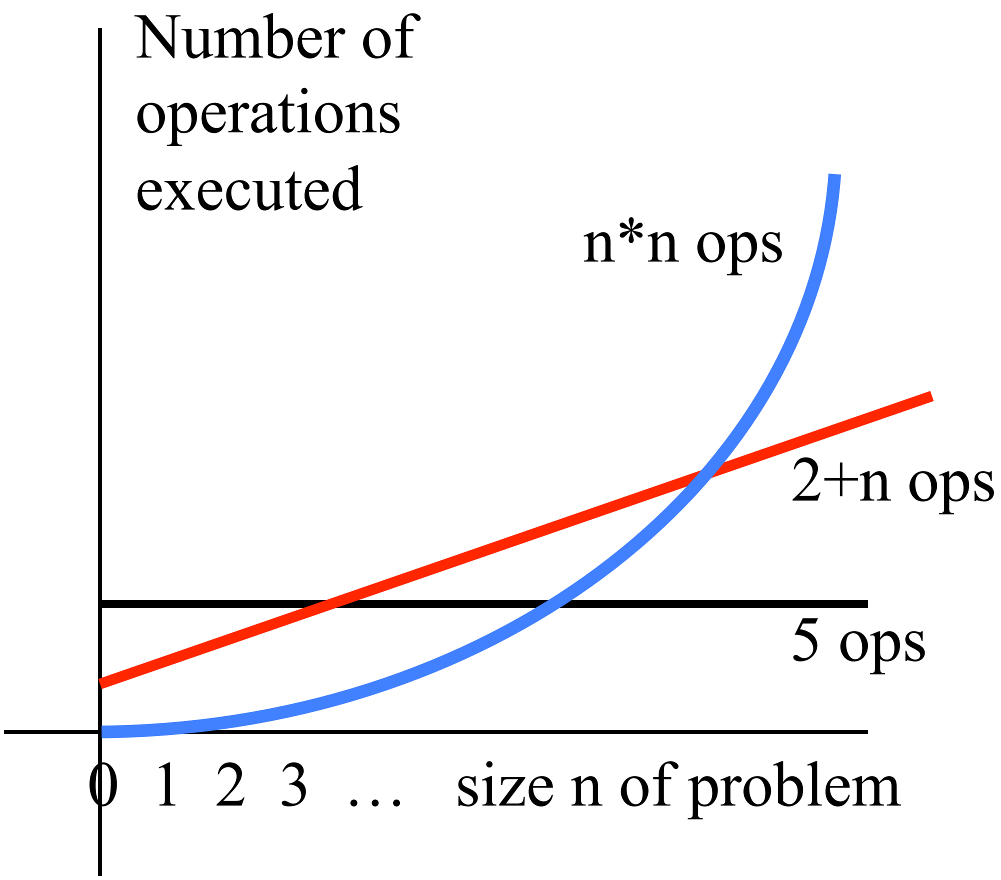
Looking at execution speed

Process an array of size n

2n+2, n+2, n are all linear in n, proportional to n



What do we want from a definition of "runtime complexity"?



1. Distinguish among cases for large n, not small n

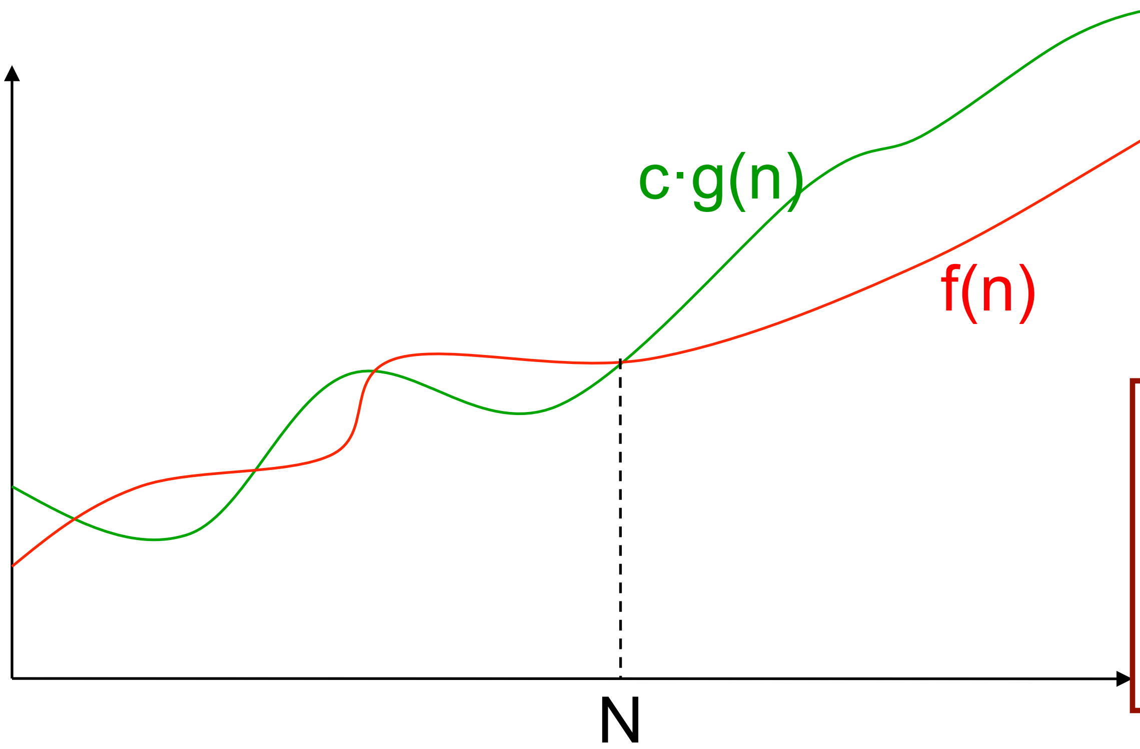
2. Distinguish among important cases, like
- n*n basic operations
- n basic operations
- log n basic operations
- 5 basic operations

3. Don't distinguish among trivially different cases.
- 5 or 50 operations
- n, n+2, or 4n operations

Definition of O(…)

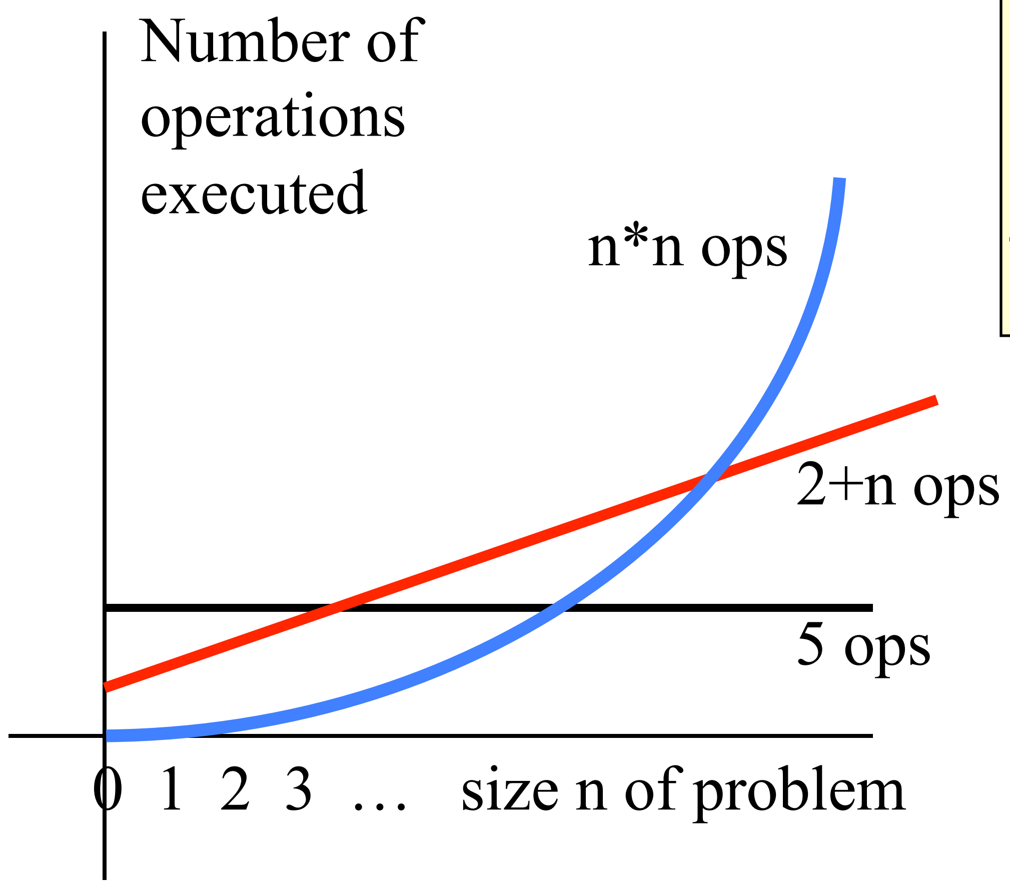
Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Graphical view



Get out far enough (for $n \geq N$) f(n) is at most $c \cdot g(n)$

What do we want from a definition of "runtime complexity"?



Formal definition: f(n) is O(g(n)) if there exist constants c > 0 and N ≥ 0 such that for all n ≥ N,

$f(n) \le c \cdot g(n)$

Roughly, f(n) is O(g(n)) means that f(n) grows like g(n) or slower, to within a constant factor

Prove that $(n^2 + n)$ is $O(n^2)$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Example: Prove that $(n^2 + n)$ is $O(n^2)$

Methodology:

Start with f(n) and slowly transform into c · g(n):

- ☐ Use = and <= and < steps
- ☐ At appropriate point, can choose N to help calculation
- ☐ At appropriate point, can choose c to help calculation

Prove that $(n^2 + n)$ is $O(n^2)$

Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Example: Prove that $(n^2 + n)$ is $O(n^2)$

$$
\begin{aligned}
& f(n) \\
= \quad & \langle \text{definition of } f(n) \rangle \\
& n^2 + n \\
<= \quad & \langle \text{for } n \geq 1,\ n \leq n^2 \rangle \\
& n^2 + n^2 \\
= \quad & \langle \text{arith} \rangle \\
& 2*n^2 \\
= \quad & \langle \text{definition of } g(n) = n^2 \rangle \\
& 2*g(n)
\end{aligned}
$$

Transform f(n) into c·g(n):

- Use =, <= , < steps
- Choose N to help calc.
- Choose c to help calc

Choose
N = 1 and c = 2

Prove that 100 n + log n is O(n)

Formal definition: f(n) is O(g(n)) if there exist constants c and N such that for all n ≥ N, f(n) ≤ c·g(n)

f(n)

= <put in what f(n) is>

100 n + log n

<= <We know log n ≤ n for n ≥ 1>

100 n + n

= <arith>

101 n

= <g(n) = n>

101 g(n)

Choose
N = 1 and c = 101

Do NOT say or write f(n) = O(g(n))

Formal definition: f(n) is O(g(n)) if there exist constants c and N such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

f(n) = O(g(n)) is simply WRONG. Mathematically, it is a disaster. You see it sometimes, even in textbooks. Don't read such things.

Here's an example to show what happens when we use = this way.

We know that n+2 is O(n) and n+3 is O(n). Suppose we use =

$$n+2 = O(n)$$
$$n+3 = O(n)$$

But then, by transitivity of equality, we have $n+2 = n+3$.

We have proved something that is false. Not good.

O(…) Examples

Let $f(n) = 3n^2 + 6n - 7$

- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^3)$
- $f(n)$ is $O(n^4)$
- …

$p(n) = 4 n \log n + 34 n - 89$

- $p(n)$ is $O(n \log n)$
- $p(n)$ is $O(n^2)$

$h(n) = 20 \cdot 2^n + 40n$

$h(n)$ is $O(2^n)$

$a(n) = 34$

- $a(n)$ is $O(1)$

Only the *leading* term (the term that grows most rapidly) matters

If it's $O(n^2)$, it's also $O(n^3)$ etc! However, we always use the smallest one

Commonly Seen Time Bounds

O(1)	constant	excellent
O(log n)	logarithmic	excellent
O(n)	linear	good
O(n log n)	n log n	pretty good
$O(n^2)$	quadratic	OK
$O(n^3)$	cubic	maybe OK
$O(2^n)$	exponential	too slow

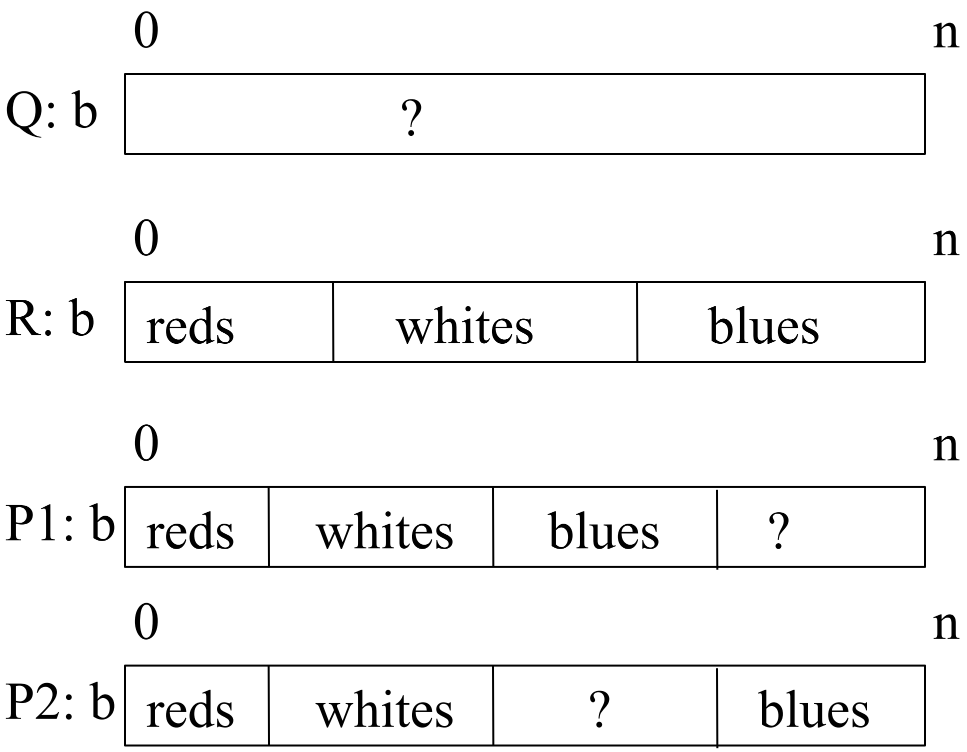
Problem-size examples

- Suppose a computer can execute 1000 operations per second; how large a problem can we solve?

operations	1 second	1 minute	1 hour
n	1000	60,000	3,600,000
n log n	140	4893	200,000
n^2	31	244	1897
$3n^2$	18	144	1096
n^3	10	39	153
2^n	9	15	21

Dutch National Flag Algorithm

Dutch national flag. Swap b[0..n-1] to put the reds first, then the whites, then the blues. That is, given precondition Q, swap values of b[0.n] to truthify postcondition R:



0 n

Q: b | ? |

(values in 0..n-1 unknown)

0 n

R: b | reds | whites | blues |

0 n

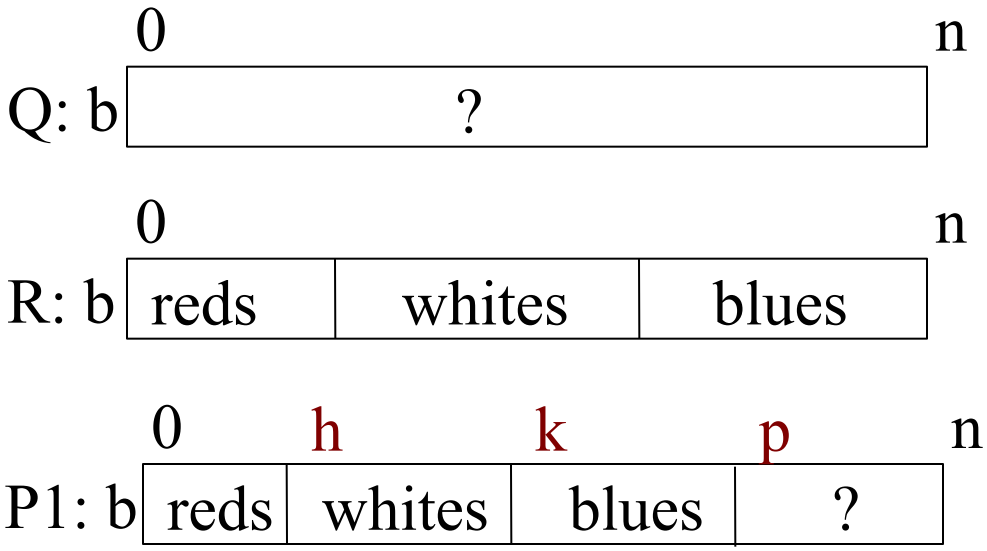
P1: b | reds | whites | blues | ? |

0 n

P2: b | reds | whites | ? | blues |

Four possible
simple invariants.
We show two.

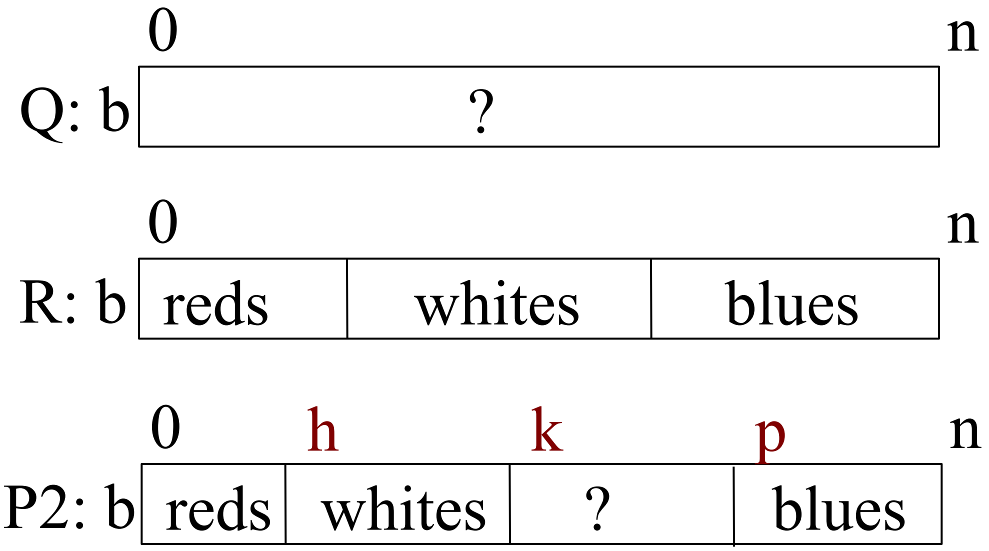
Dutch National Flag Algorithm: invariant P1



```
h= 0; k= h; p= k;
while (      p != n        ) {
    if (b[p] blue)     p=  p+1;
    else if (b[p] white) {
          swap b[p], b[k];
          p= p+1; k= k+1;
    }
    else { // b[p] red
           //REQUIRES
           // TWO SWAPS
    }
           // you can finish it
}
```

Don’t need long mnemonic names for these variables! The invariant gives you all the info you need about them!

Dutch National Flag Algorithm: invariant P2



0 n
Q: b ?

0 n
R: b reds whites blues

0 h k p n
P2: b reds whites ? blues

```
h= 0; k= h; p= n;
while (     k != p      ) {
    if (b[k] white)  k=  k+1;
    else if (b[p] blue) {
            p= p-1;
            swap b[k], b[p];
    }
    else { // b[k] is red
            swap b[k], b[h];
            h= h+1; k= k+1;
    }
}
```

Use inv P1:
perhaps 2 swaps per iteration.

Use inv P2:
at most 1 swap per iteration.