FIT1045: Algorithms and Programming Fundamentals in Python

Lecture 12

Decrease and Conquer

Objectives

Objectives of this lecture are to:

1. Know to search efficiently in ordered sequence (Binary Search)
2. Understand design paradigm decrease-and-conquer and recognise situations with logarithmic complexity
3. Demonstrate the time complexity of Euclid's algorithm

This covers learning outcomes:

- 2 – choose and implement appropriate problem solving strategies
- 5 – determine the computational cost and limitations of algorithms

Overview

1. The Ordered Search Problem
2. Binary Search
3. Revisiting Euclid's Algorithm

Search in Ordered Sequence

Gertrudis Atkinson 0463935372

Kiley Basinger 0411484152

Romana Brose 0418721183

Shayne Brotherton 0436242684

Calandra Clifton 0479753034

Roy Dupuis 0445778949

Leticia Fukushima 0436756947

Cherlyn Gayles 0483503919

…

Problem: find (the position of) a name in a phone book.

Search in Ordered Sequence

ato.gov.au 180.149.195.3

cancer.org.au 52.187.229.23

facebook.com 31.13.71.36

google.com 172.217.12.142

monash.edu 43.245.43.30

newscientist.com 45.60.19.101

news.com.au 23.221.48.198

wikipedia.org 208.80.154.224

…

Problem: find URL in DNS records.

Sequential search solves problem

```
def sequential_search(v, seq):
    """I: value v and sequ
       O: an index of seq
          (if no such inde
    """
    n = len(seq)
    i = 0
    while i < n:
        if seq[i] == v:
            return i
        i += 1
    return None
```

Quiz time (https://flux.qa)
Clayton: AXXULH
Malaysia: LWERDE

`n = len(seq)`	$O(0)$	$O(0)$
`i = 0`	$O(0)$	$O(0)$
`while i < n:`	$O(1)$	$O(n)$
`if seq[i] == v:`	$O(1)$	$O(n)$
`return i`	$O(1)$	$O(0)$
`i += 1`	$O(0)$	$O(n)$
`return None`	$O(0)$	$O(0)$
	$O(1)$	$O(n)$

Example Scenario (e.g. Web search):

- need to find 10.000.000 values from sequence of 2.000.000.000 entries
- cost of 1 elementary step is 1ns (and constant factor in O-notation is 1)

Outcome: need about 40 years

Can we solve problem *more efficiently* for ordered sequences?

```
def sequential_search(v, seq):
    """I: value v and ordered sequence seq
       O: an index of seq with value v or None
          (if no such index exists)
    """
    n = len(seq)
    i = 0
    while i < n:
        #I) v in seq[i:] or v not in seq
        if seq[i] == v:
            return i
        i += 1
    return None
```

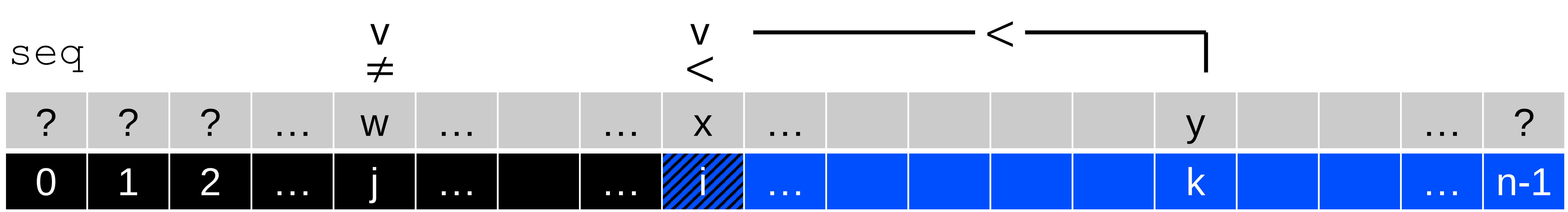
seq

v
≠

?	?	?	…	w	…		…	x	…								…	?
0	1	2	…	j	…		…	i	…								…	n-1

However: order of sequence *implies* many more comparisons

```
def sequential_search(v, seq):
    n = len(seq)
    i = 0
    while i < n:
        #I) v in seq[i:] or v not in seq
        if seq[i] == v:
            return i
        i += 1
    return None
```



However: order of sequence *implies* many more comparisons

```python
def sequential_search2(v, seq):
    n = len(seq)
    i = 0
    while i < n:
        #I) v in seq[i:] or v not in seq
        if seq[i] == v:
            return i
        if seq[i] > v:
            # v not in seq[:i]
            return None
        i += 1
    return None
```

Does that reduce worst-case complexity?

					v ≠				v <				<						
seq	?	?	?	…	w	…		…	x	…					y			…	?
	0	1	2	…	j	…		…	i	…					k			…	n-1

What single comparison rules out most positions (worst case)?

?	?	?	…														…	?
0	1	2	…														…	n-1

What single comparison rules out most positions (worst case)?

Case 1: v = q

v
=

≤q	≤q	≤q	…				…	≤q	q	≥q	…						…	≥q
0	1	2	…					…	n//2	…							…	n-1

What single comparison rules out most positions (worst case)?

Case 1: v = q

v
=

≤q	≤q	≤q	…				…	≤q	q	≥q	…						…	≥q
0	1	2	…					…	n//2	…							…	n-1

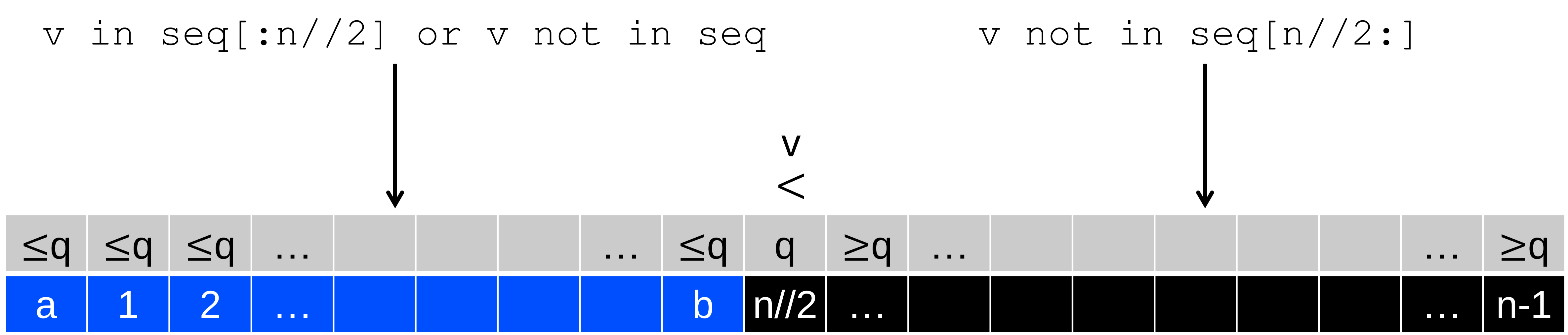
What single comparison rules out most positions (worst case)?

Case 2: v < q

									v									
									<									
≤q	≤q	≤q	…				…	≤q	q	≥q	…						…	≥q
0	1	2	…					…	n//2	…							…	n-1

What single comparison rules out most positions (worst case)?

Case 2: v < q



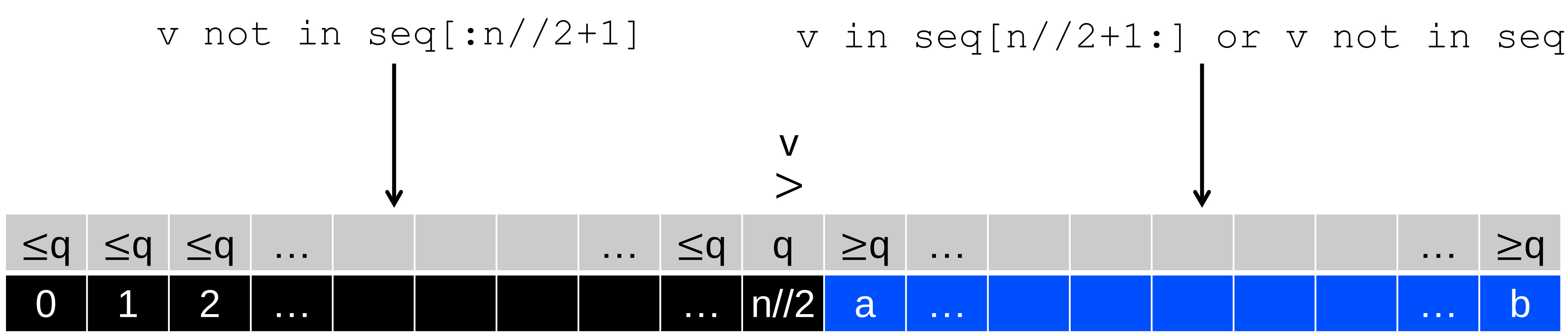
What single comparison rules out most positions (worst case)?

Case 3: $v > q$

									v $>$									
$\le q$	$\le q$	$\le q$	…				…	$\le q$	q	$\ge q$	…						…	$\ge q$
0	1	2	…					…	n//2	…							…	n-1

What single comparison rules out most positions (worst case)?

Case 3: v > q



Overview

1. The Ordered Search Problem
2. Binary Search
3. Revisiting Euclid's Algorithm

Decrease-and-Conquer: reduce problem to smaller subproblem

```python
def probing_search(v, seq):
    a, b = 0, len(seq)-1
    c = b // 2
    if seq[c] == v:
        return c
    elif v < seq[c]:
        b = c - 1
    else:
        a = c + 1

    # search between a and b
```

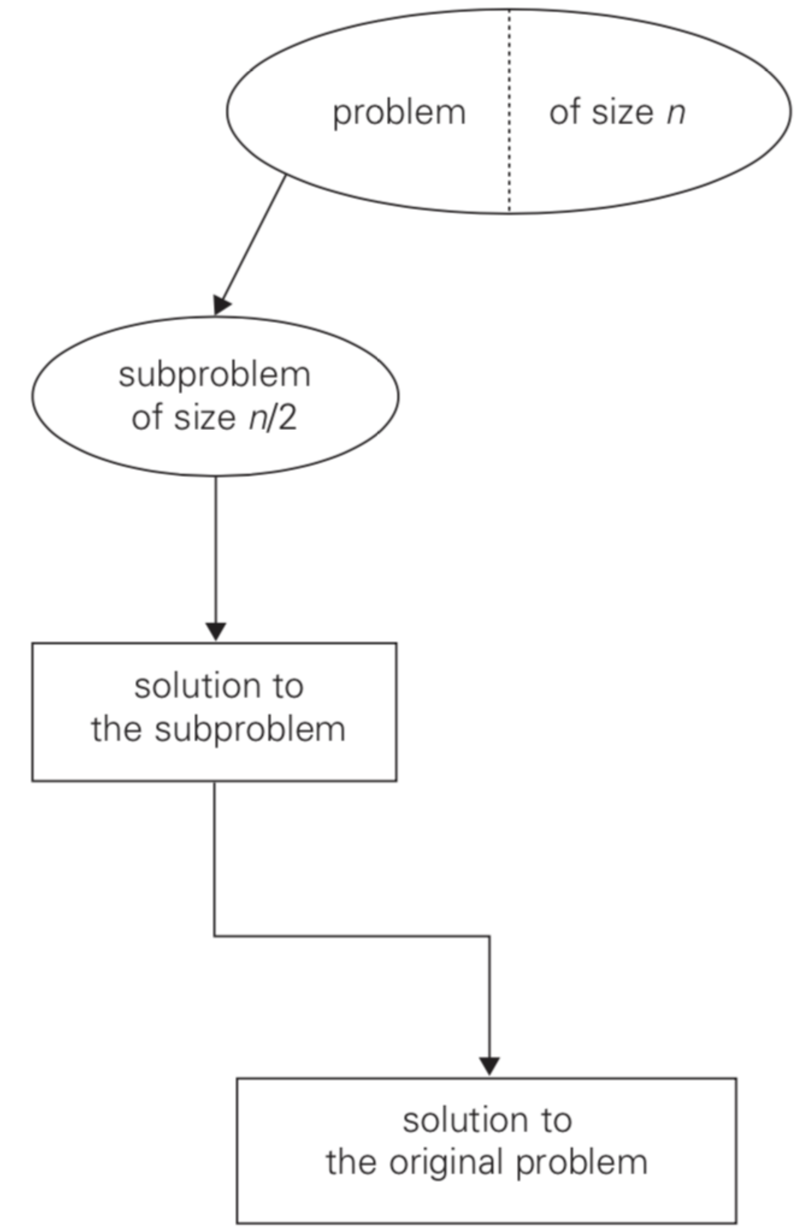


illustration: [Levitin, p. 133]

v
>

≤q	≤q	≤q	…				…	≤q	q	≥q	…						…	≥q
0	1	2	…					…	n//2	a	…						…	b

We could solve remaining sub-problem as before…

```python
def probing_search(v, seq):
    a, b = 0, len(seq)-1
    c = b // 2
    if seq[c] == v:
        return c
    elif v < seq[c]:
        b = c - 1
    else:
        a = c + 1
    i = a
    while a <= i <= b:
        if seq[i] == v:
            return i
        i += 1
    return None
```

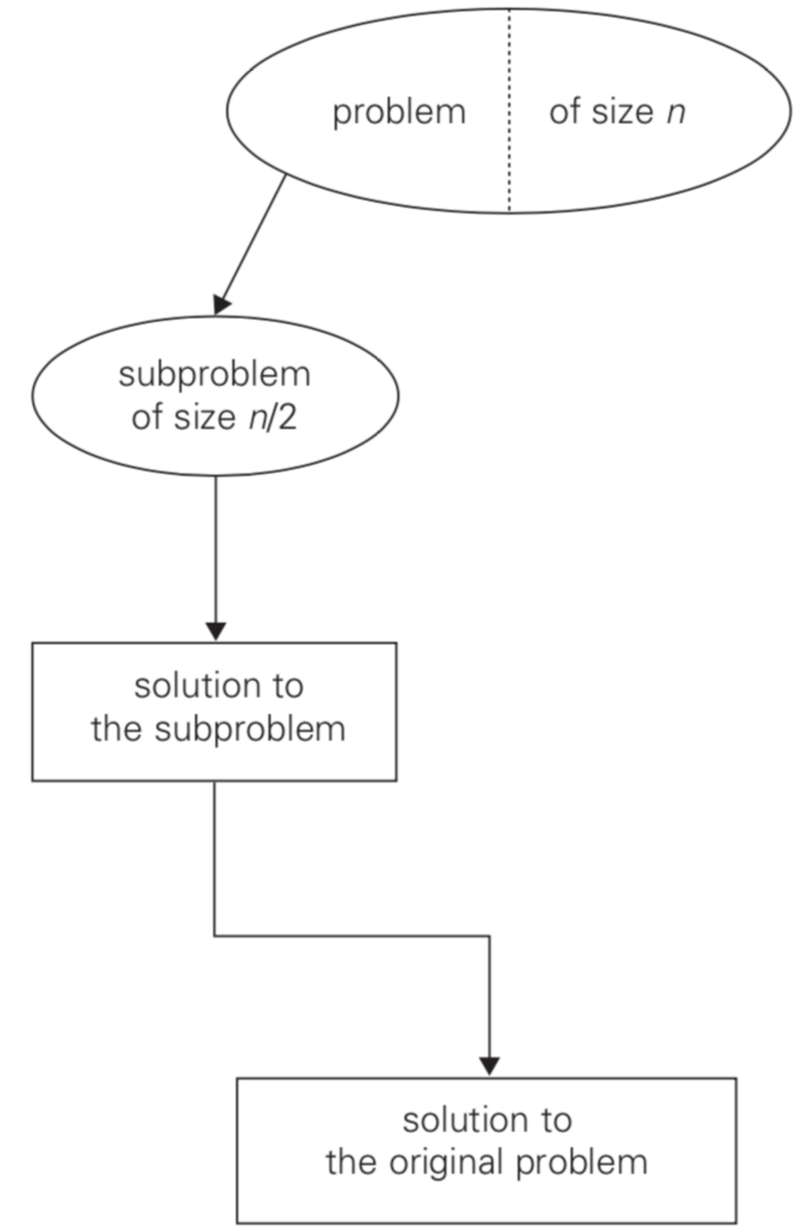


illustration: [Levitin, p. 133]

v
>

≤q	≤q	≤q	…				…	≤q	q	≥q	…						…	≥q
0	1	2	…					…	n//2	a	…						…	b

Instead: let's re-apply same principle

```python
def probing_search(v, seq):
    a, b = 0, len(seq)-1
    c = b // 2
    if seq[c] == v:
        return c
    elif v < seq[c]:
        b = c - 1
    else:
        a = c + 1
    i = a
    while a <= i <= b:
        if seq[i] == v:
            return i
        i += 1
    return None
```

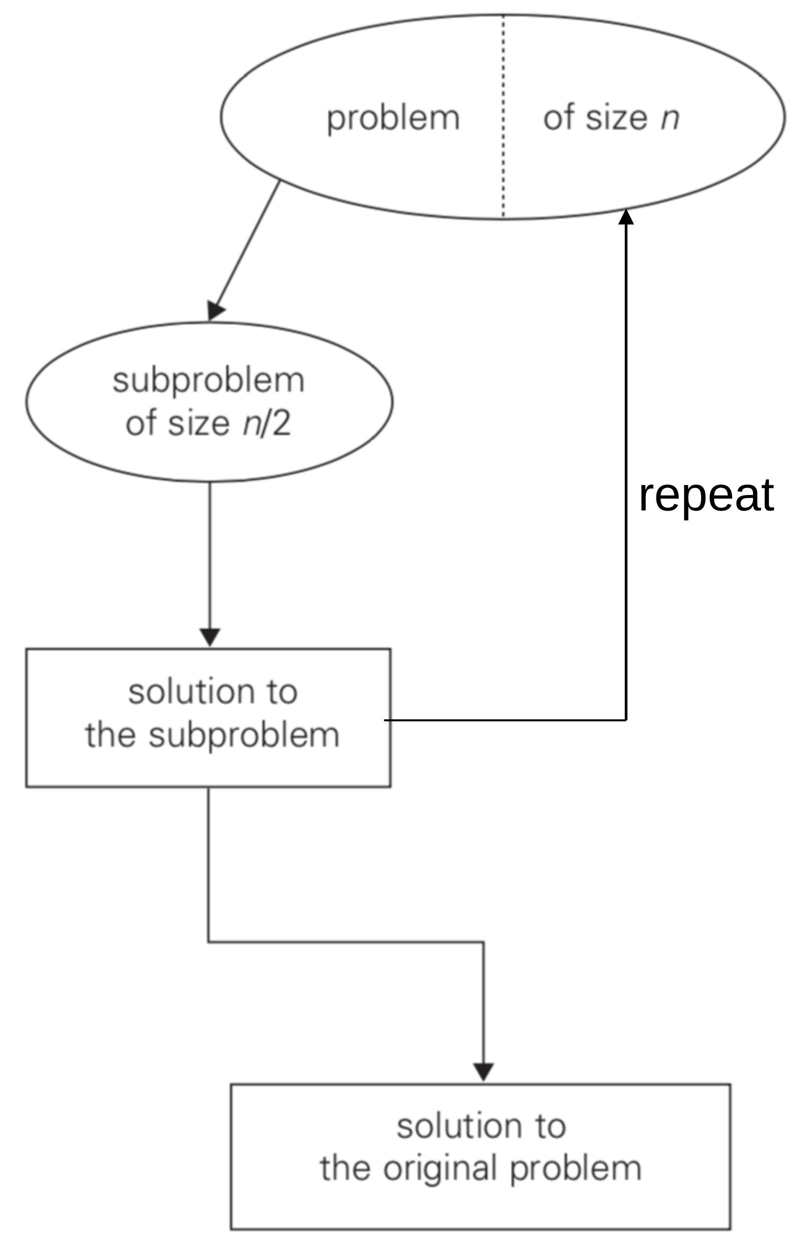
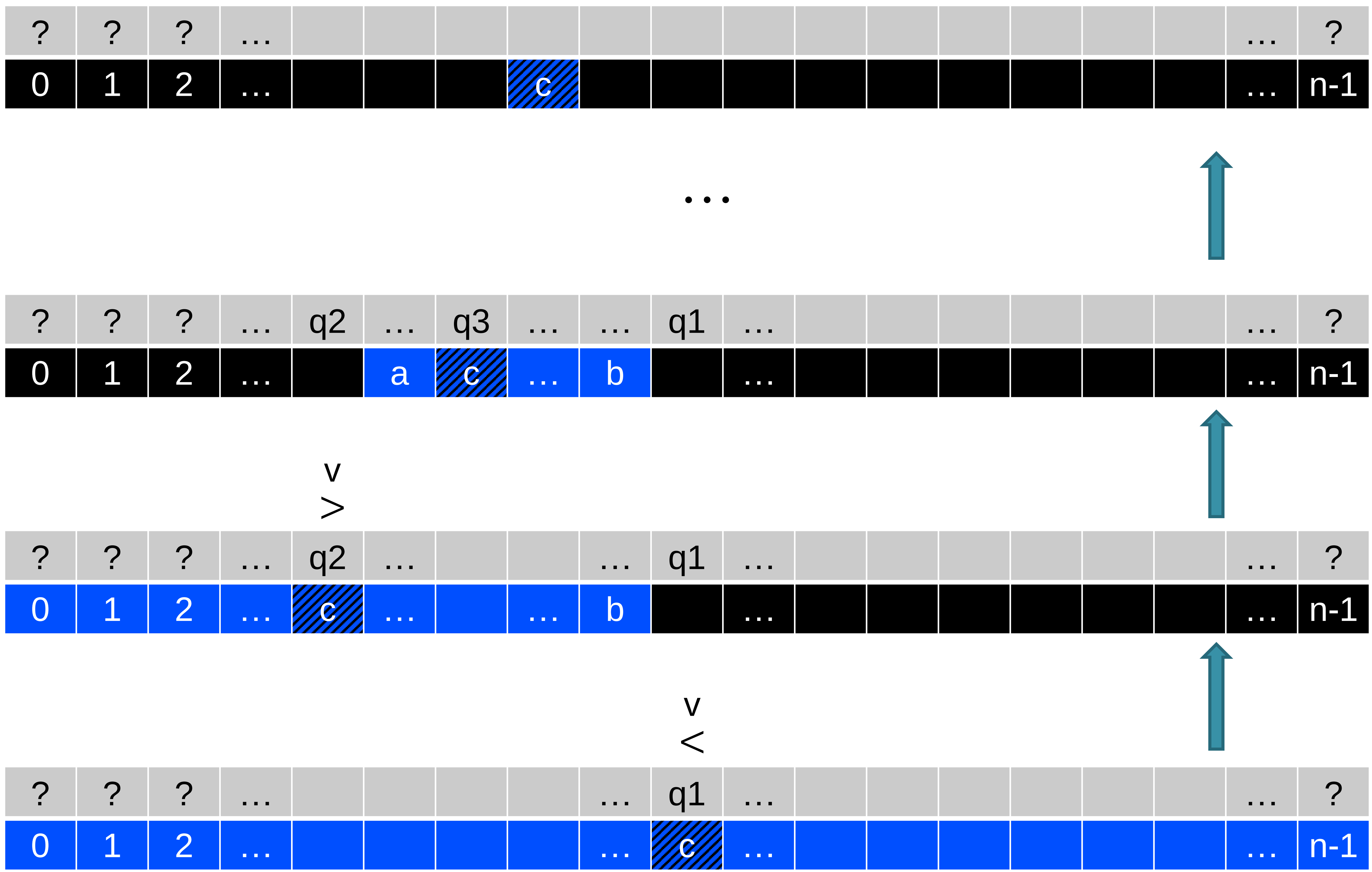


illustration: [Levitin, p. 133]

v
>

≤q	≤q	≤q	…				…	≤q	q	≥q	…						…	≥q
0	1	2	…					…	n//2	a	…						…	b

Instead: let’s re-apply same principle



Binary Search

```
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
```

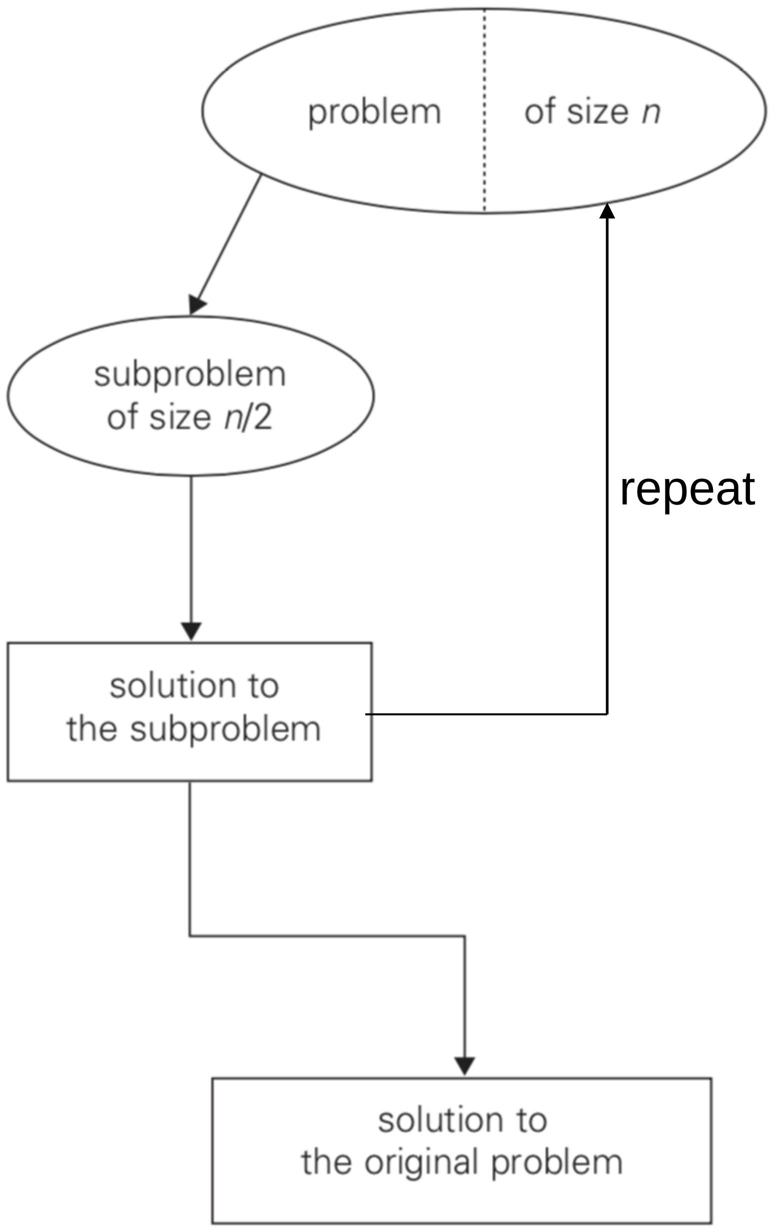


illustration: [Levitin, p. 133]

Binary Search

```
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
    # in iteration i:
        c = (a + b) // 2
        if seq[c] == v:
            return c
        elif seq[c] > v:
            b = c - 1
        else:
            a = c + 1
```

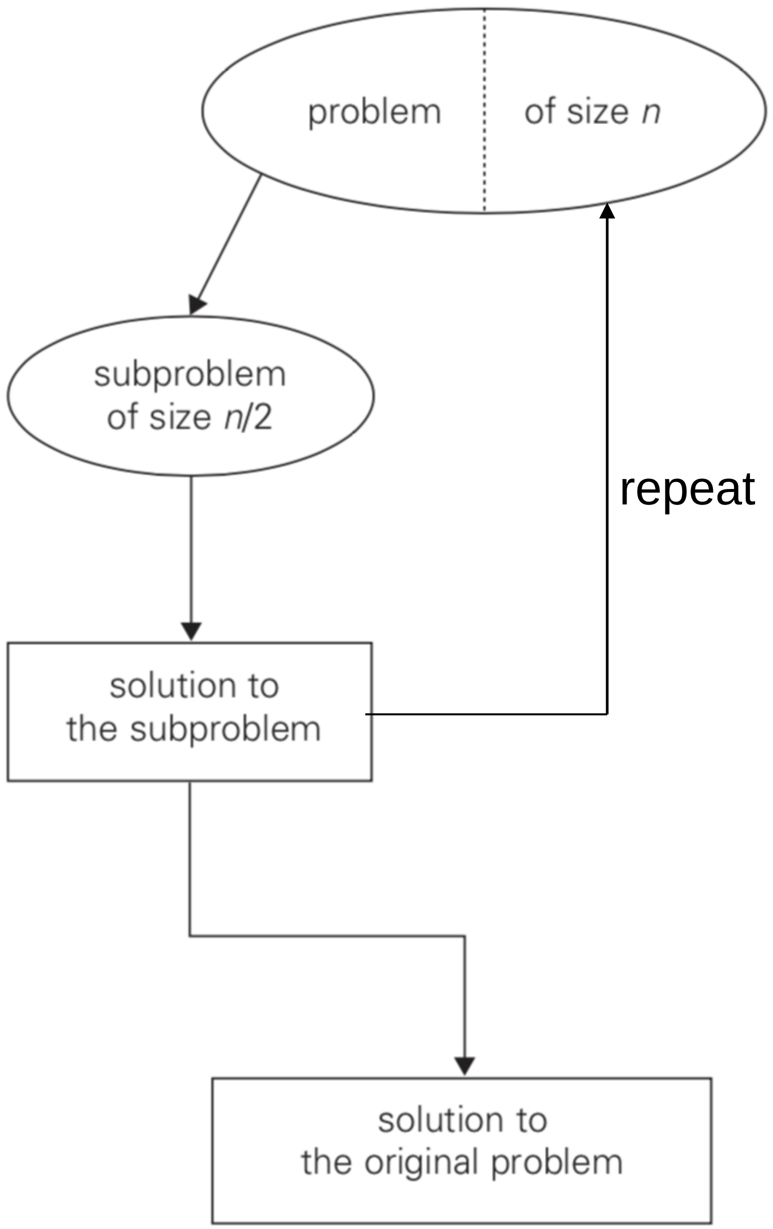


illustration: [Levitin, p. 133]

Binary Search

```
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
    while a <= b:
        c = (a + b) // 2
        if seq[c] == v:
            return c
        elif seq[c] > v:
            b = c - 1
        else:
            a = c + 1
    return None
```

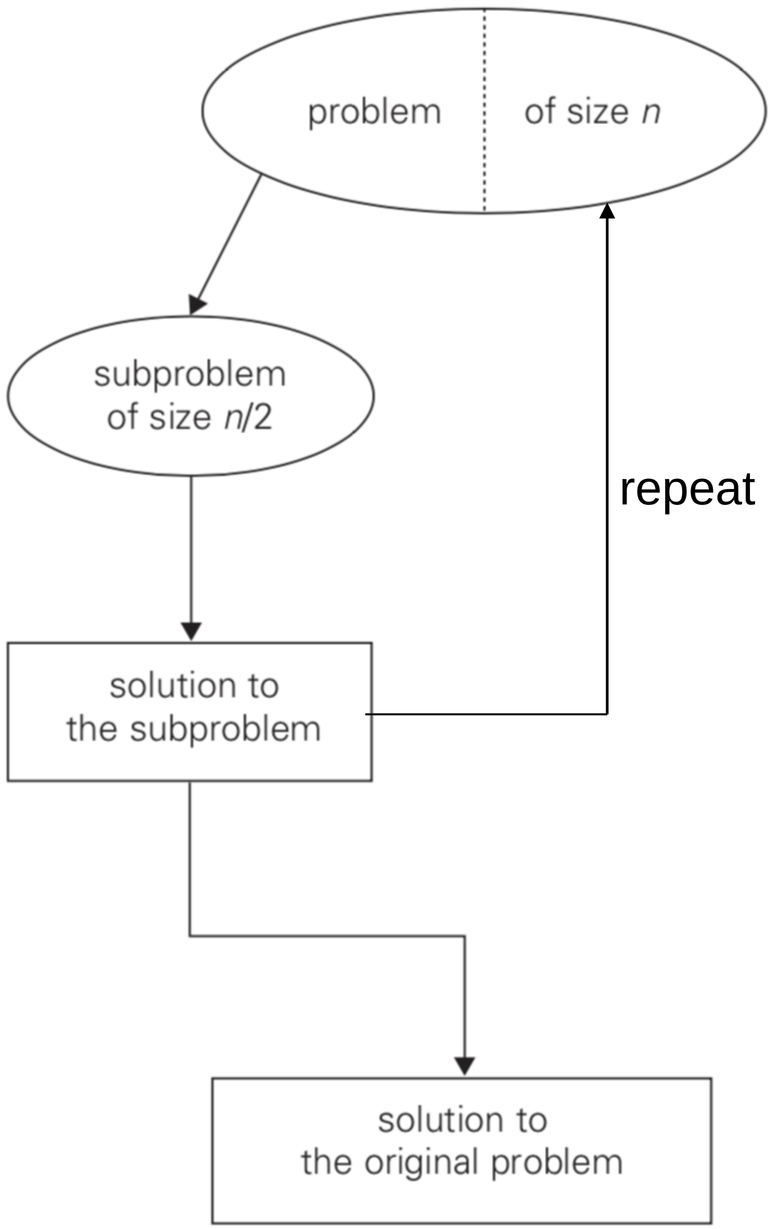
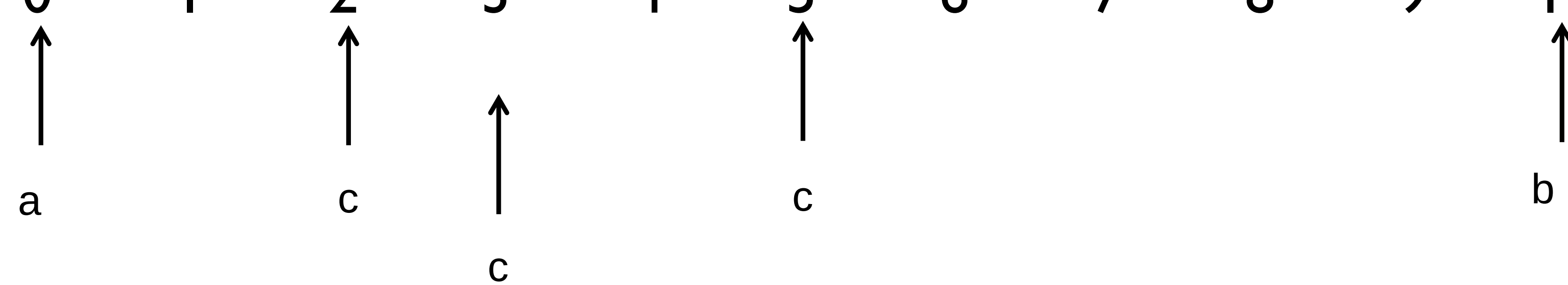


illustration: [Levitin, p. 133]

Binary Search

```python
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
    while a <= b:
        c = (a + b) // 2
        if seq[c] == v:
            return c
        elif seq[c] > v:
            b = c - 1
        else:
            a = c + 1
    return None
```

seq:	5	10	15	20	25	30	35	40	45	50	55
(v = 20)	0	1	2	3	4	5	6	7	8	9	10



Computational complexity of Binary Search

```python
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
    while a <= b:
        c = (a + b) // 2
        if seq[c] == v:
            return c
        elif seq[c] > v:
            b = c - 1
        else:
            a = c + 1
    return None
```

`def binary_search(v, seq):`	$n = \text{len}(\text{lst})$
`a = 0`	$O(0)$
`b = len(seq) - 1`	$O(1)$
`while a <= b:`	$O(1)$
`c = (a + b) // 2`	$O(1)$
`if seq[c] == v:` … `a = c + 1`	$O(1)$
`return None`	$O(0)$
	$O(1)$

Quiz time (https://flux.qa)

Clayton: AXXULH

Malaysia: LWERDE

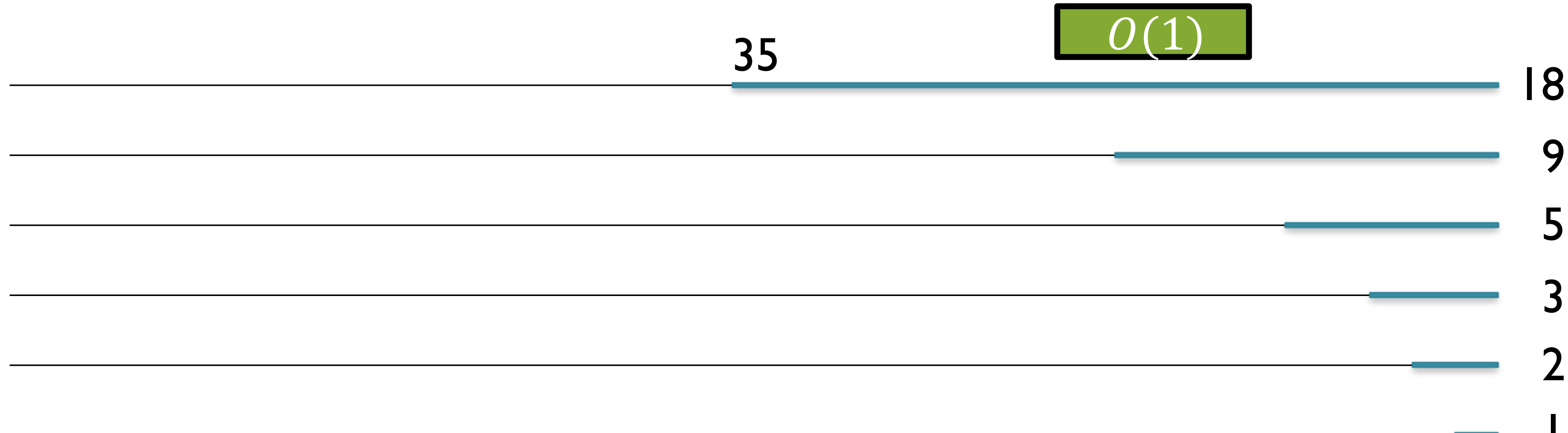
Computational complexity of Binary Search

```python
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
    while a <= b:
        c = (a + b) // 2
        if seq[c] == v:
            return c
        elif seq[c] > v:
            b = c - 1
        else:
            a = c + 1
    return None
```

$n = \text{len(lst)}$

$O(0)$	$O(0)$
$O(1)$	$O(1)$
$O(1)$	?
$O(1)$	
$O(1)$	
$O(0)$	

$\boxed{O(1)}$



Computational complexity of Binary Search

```
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
    while a <= b:
        c = (a + b) // 2
        if seq[c] == v:
            return c
        elif seq[c] > v:
            b = c - 1
        else:
            a = c + 1
    return None
```

Line		
`def binary_search(v, seq):`	$n = \text{len(lst)}$	
`a = 0`	$O(0)$	$O(0)$
`b = len(seq) - 1`	$O(1)$	$O(1)$
`while a <= b:`	$O(1)$	$O(\log n)$
`c = (a + b) // 2`	$O(1)$	$O(\log n)$
`if … a = c + 1`	$O(1)$	$O(\log n)$
`return None`	$O(0)$	$O(1)$
	$O(1)$	$O(\log n)$

- Let $n_i = b_i - a_i + 1$ be problem size after i iterations of loop
- In the beginning: $n_0 = n$
- In every iteration size is cut in half: $n_i = \lceil n_{i-1}/2 \rceil$, i.e., $n_i = \lceil n/2^i \rceil$
- After $k = \lceil \log_2 n \rceil$ iterations: $n_k = \lceil n/2^{\log_2 n} \rceil = 1$, i.e., $b_k = a_k$
- So at most $O(\log_2 n)$ loop iterations

Computational complexity of Binary Search

```python
def binary_search(v, seq):
    a = 0
    b = len(seq) - 1
    while a <= b:
        c = (a + b) // 2
        if seq[c] == v:
            return c
        elif seq[c] > v:
            b = c - 1
        else:
            a = c + 1
    return None
```

Code	$n = \mathrm{len(lst)}$	
`a = 0`	$O(0)$	$O(0)$
`b = len(seq) - 1`	$O(1)$	$O(1)$
`while a <= b:`	$O(1)$	$O(\log n)$
`c = (a + b) // 2`	$O(1)$	$O(\log n)$
`if` … `a = c + 1`	$O(1)$	$O(\log n)$
`return None`	$O(0)$	$O(1)$
	$O(1)$	$O(\log n)$

Example Scenario (e.g. Web search):

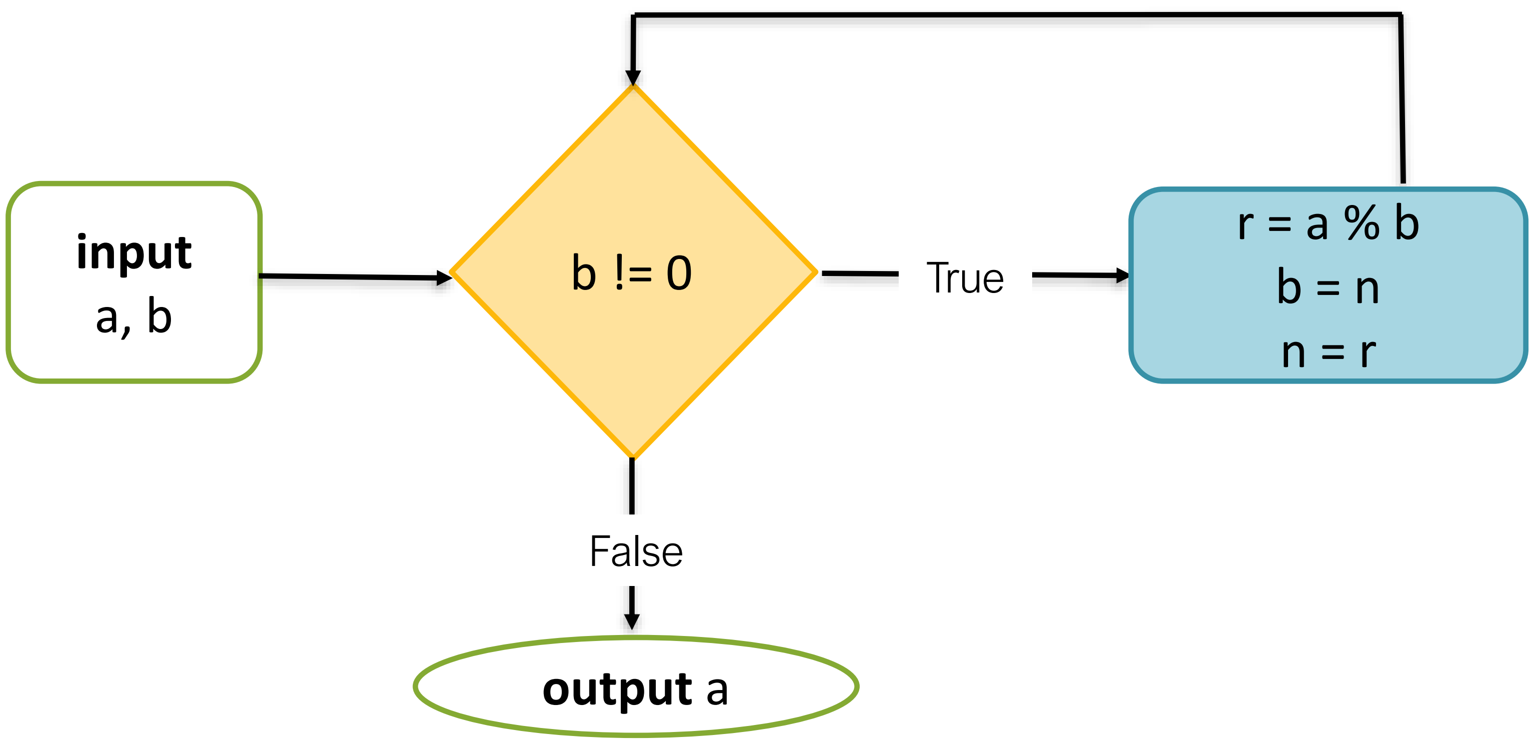
- need to find 10.000.000 values from sequence of 2.000.000.000 entries
- cost of 1 elementary step is 1ns (and constant factor in O-notation is 1)

Outcome: need about 0.3 seconds

Overview

1. The Ordered Search Problem
2. Binary Search
3. Revisiting Euclid's Algorithm

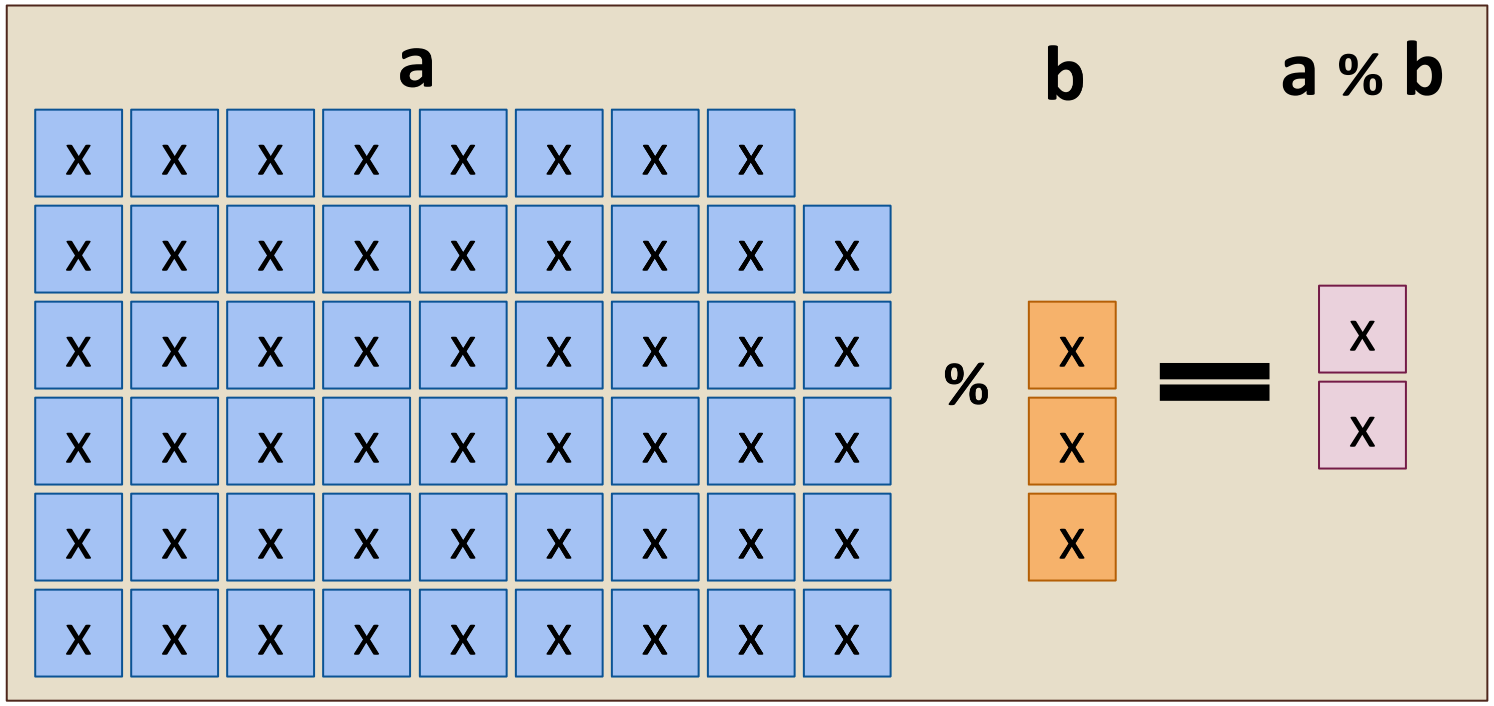
Recall Euclid’s Algorithm

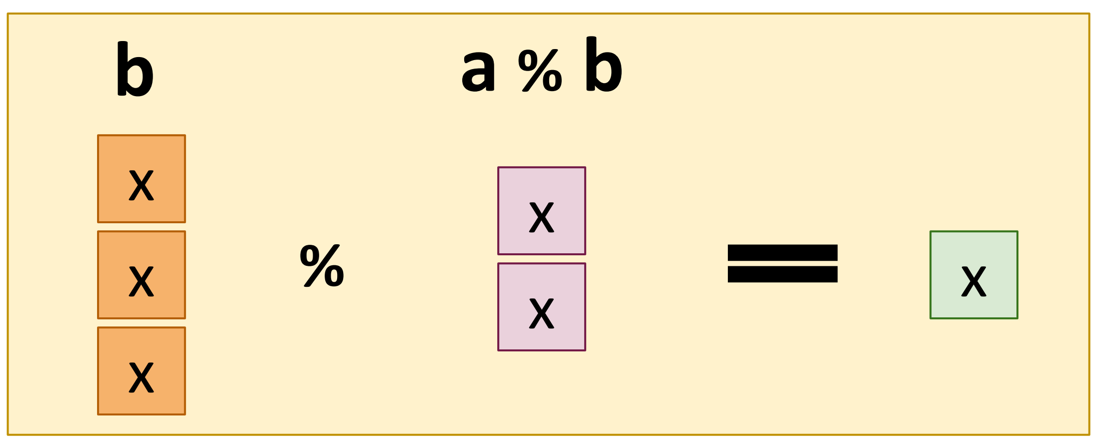


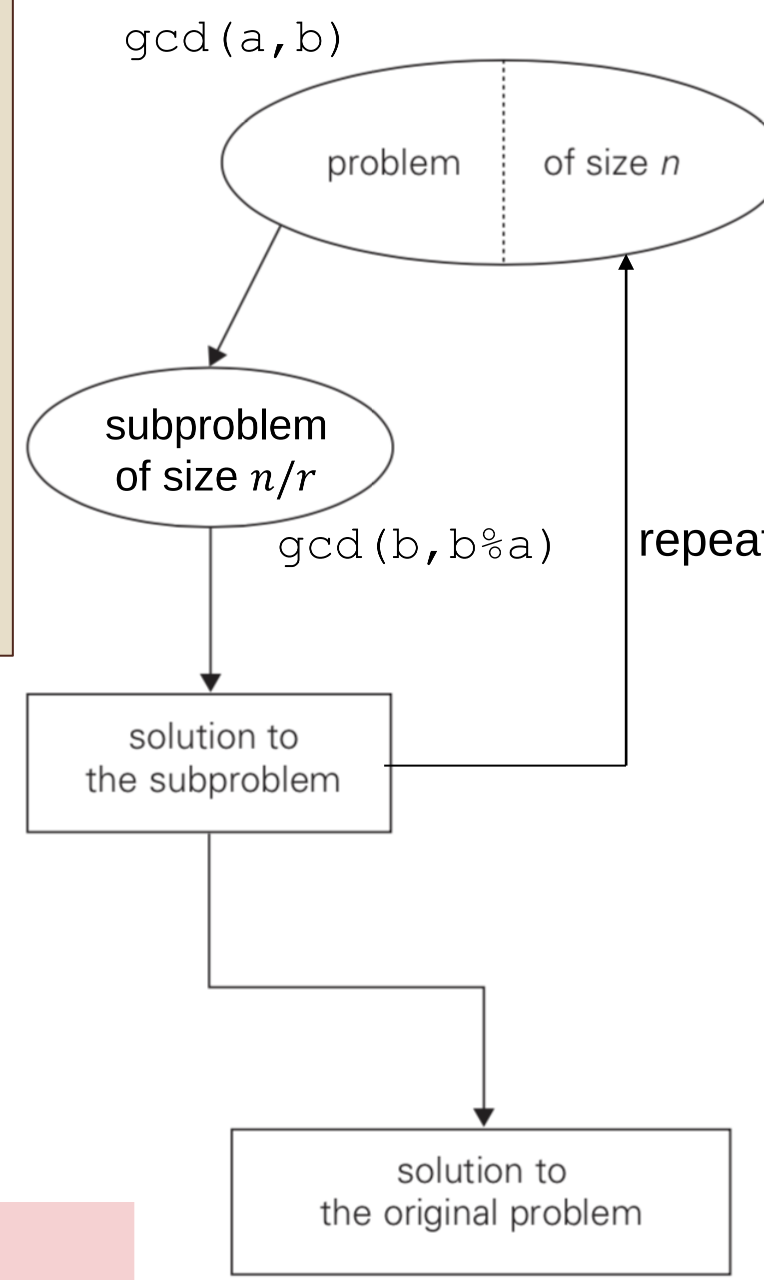
Eukleides of Alexandria
3xx BC – 2xx BC

```
def gcd_euclid(a, b):
    """
    Input : integers a and b such that not a==b==0
    Output: the greatest common divisor of a and b
    """
    while b != 0:
        a, b = b, a % b
    return a
```

Instance of decrease and conquer



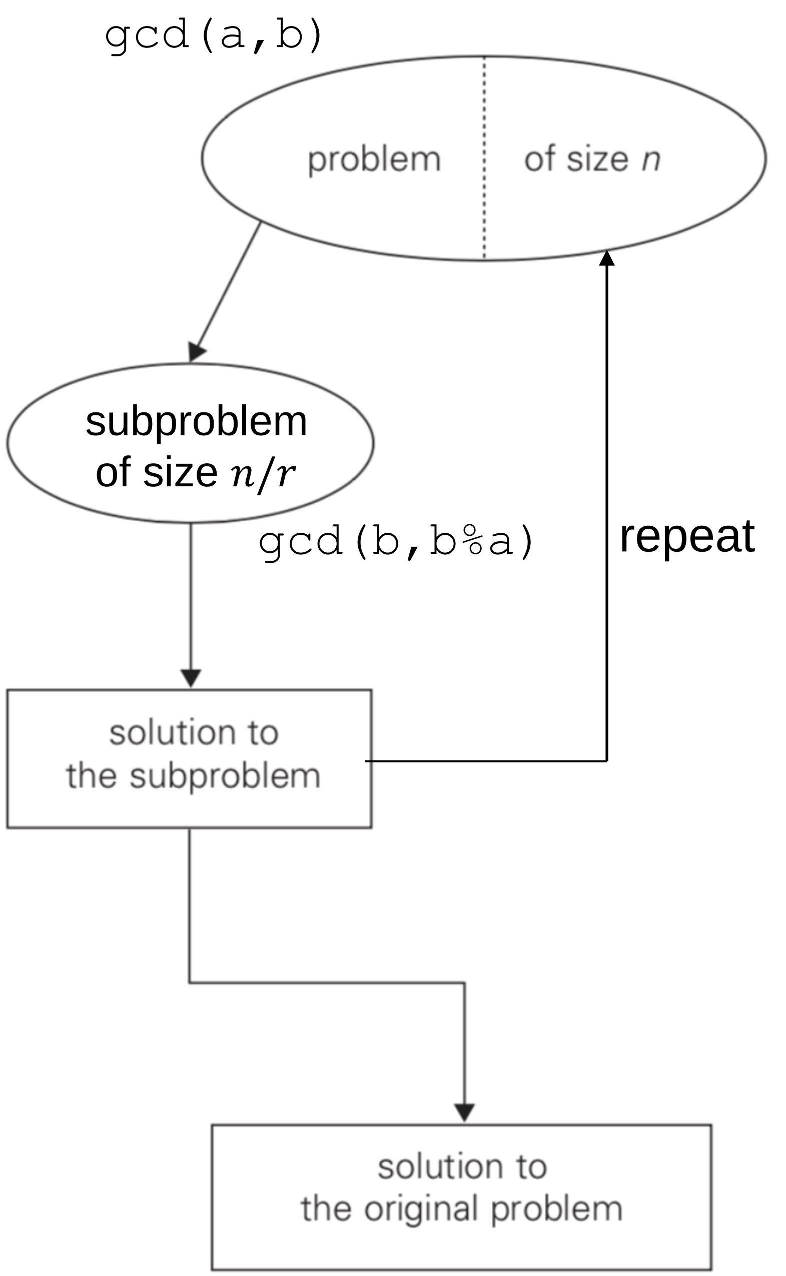





```python
def gcd_euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

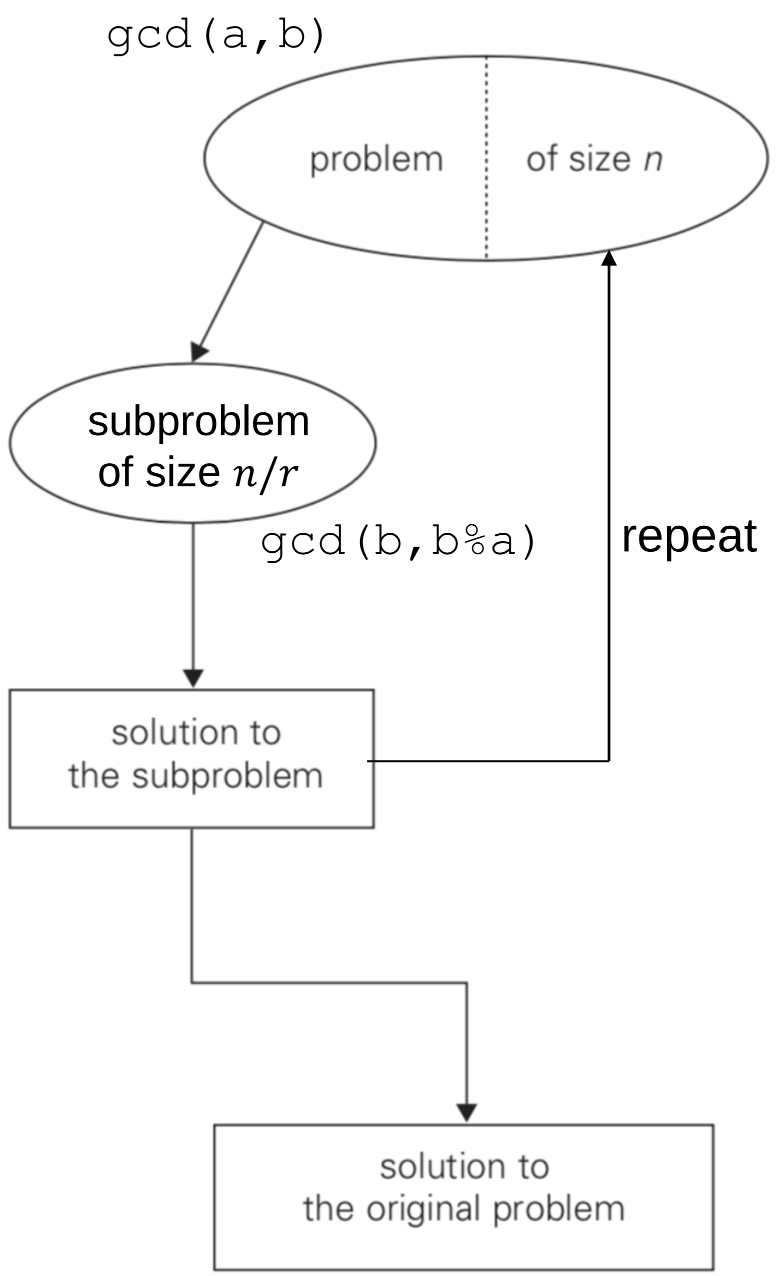
Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a0 and b0 such
       that not a0==a0==0
    O: gcd(a0,b0)
    """
    while b != 0:
        a, b = b, a % b
    return a
```



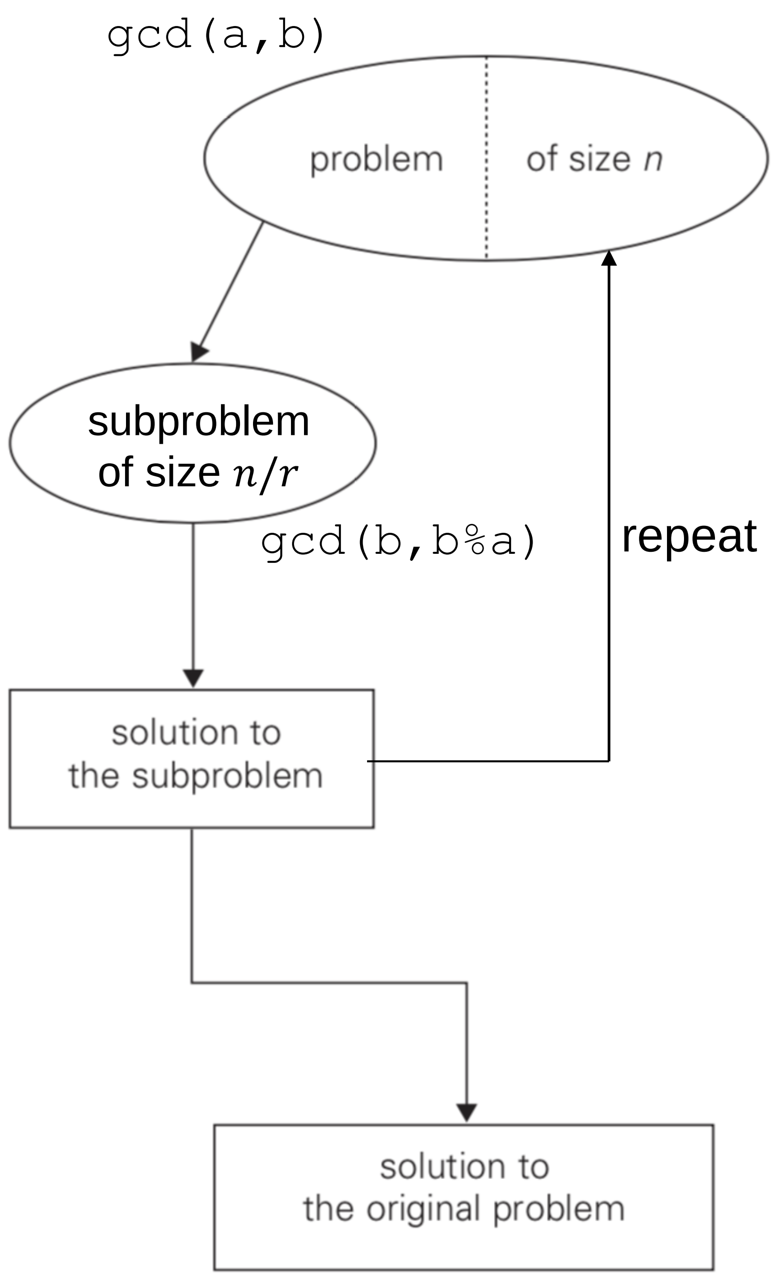
Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a0 and b0 such
       that not a0==a0==0
    O: gcd(a0,b0)
    """
    #PRC: a,b==a0,b0 (original input)
    while b != 0:
        a, b = b, a % b
    return a
```

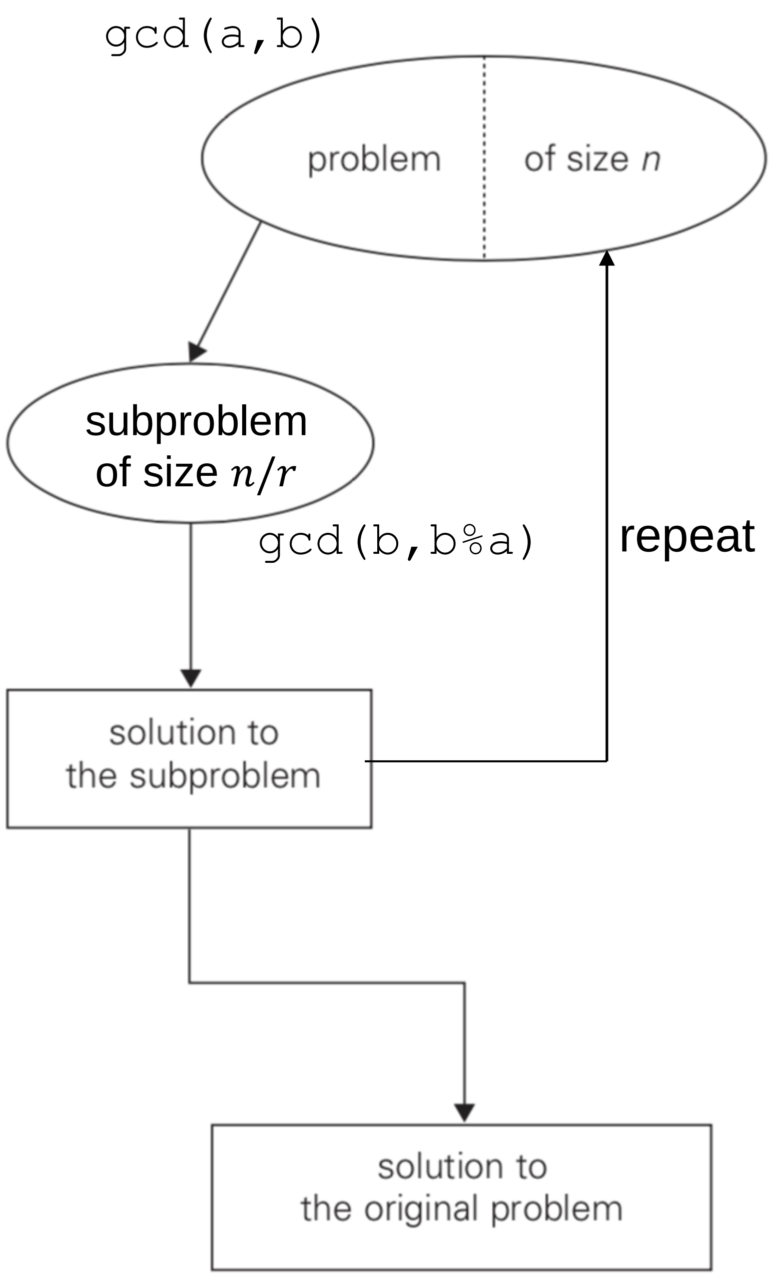


Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a0 and b0 such
       that not a0==a0==0
    O: gcd(a0,b0)
    """
    #PRC: a,b==a0,b0 (original input)
    while b != 0:
        #I: gcd(a,b)==gcd(a0,b0)
        a, b = b, a % b
        #I: gcd(a,b)==gcd(a0,b0)
    return a
```



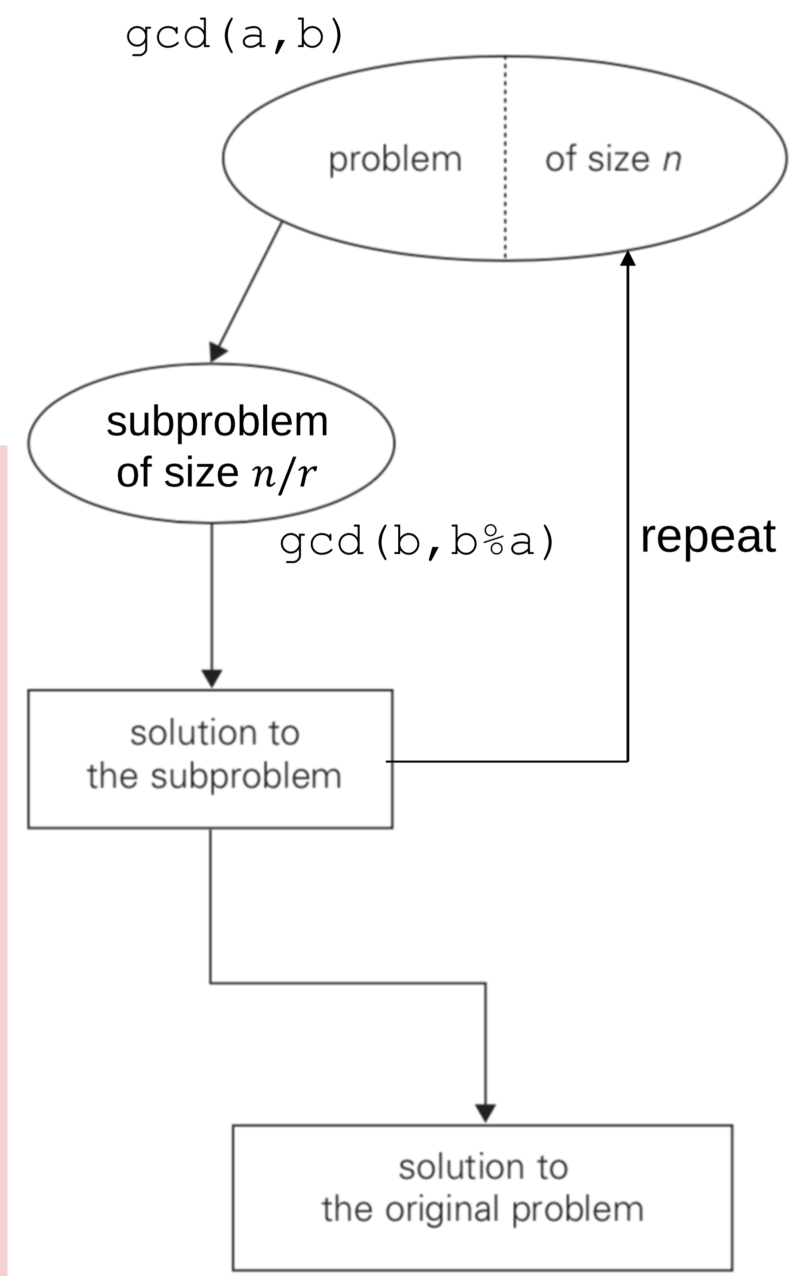
Exercise: correctness via loop invariant




```
def gcd_euclid(a, b):
    """
    I: integers a0 and b0 such
       that not a0==a0==0
    O: gcd(a0,b0)
    """
    #PRC: a,b==a0,b0 (original input)
    while b != 0:
        #I: gcd(a,b)==gcd(a0,b0)
        a, b = b, a % b
        #I: gcd(a,b)==gcd(a0,b0)
    #EXC: b==0
    return a
```

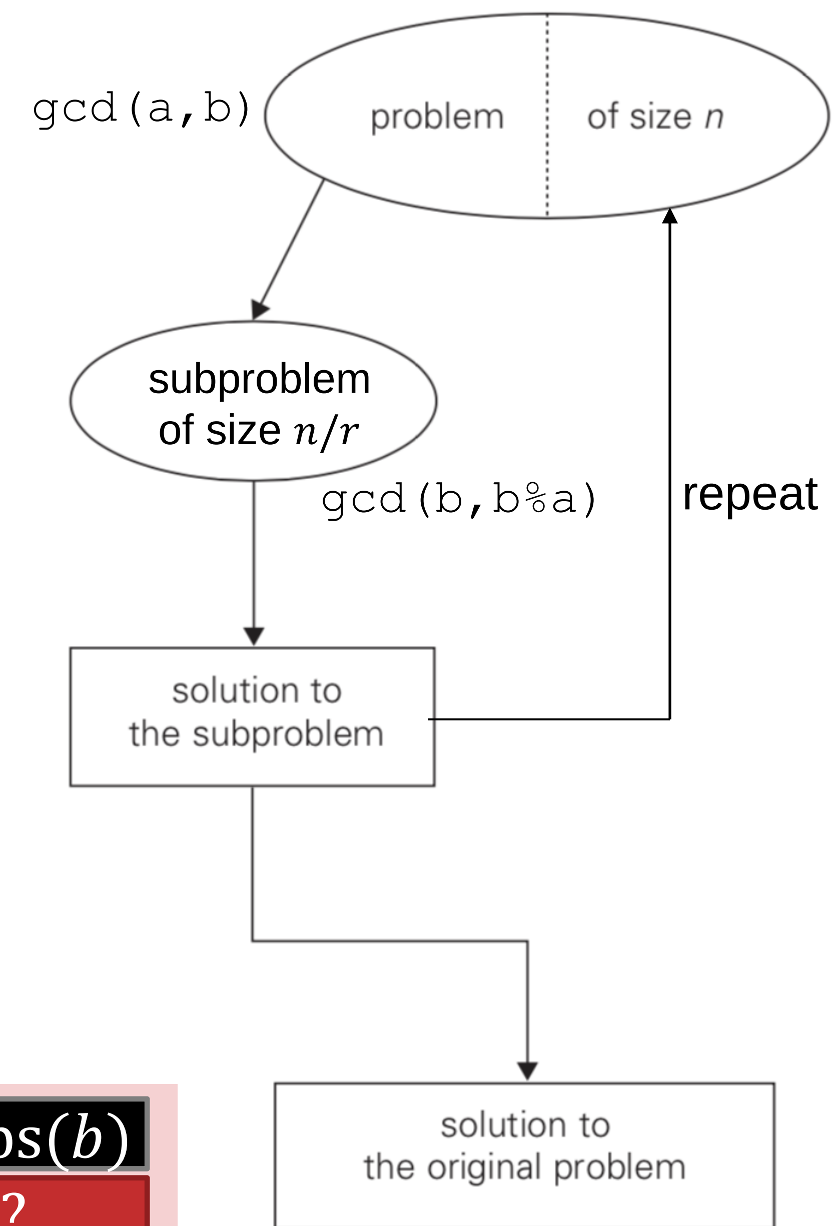
Exercise: correctness via loop invariant

```
def gcd_euclid(a, b):
    """
    I: integers a0 and b0 such
       that not a0==a0==0
    O: gcd(a0,b0)
    """
    #PRC: a,b==a0,b0 (original input)
    while b != 0:
        #I: gcd(a,b)==gcd(a0,b0)
        a, b = b, a % b
        #I: gcd(a,b)==gcd(a0,b0)
    #EXC: b==0
    #POC: a==gcd(a,b)==gcd(a0,b0)
    return a
```



Can we analyse computational complexity as for binary search?

Need to determine how many iterations we can have in worst case!



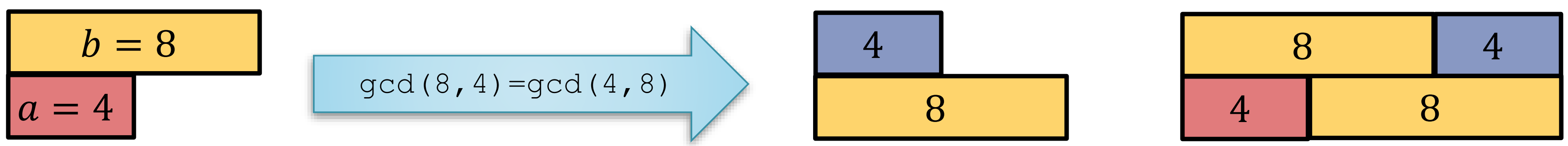

```
def gcd_euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Line		
`def gcd_euclid(a, b):`	$n = \text{abs}(a) + \text{abs}(b)$	
`while b != 0:`	$O(1)$	?
`a, b = b, a % b`	0	?
`return a`	$O(1)$	$O(1)$
	$O(1)$	?

By what factor is problem decreased per iteration of Euclid's Algorithm?

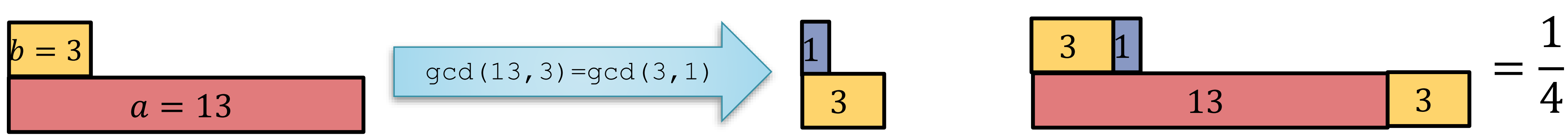
Example 1

No decrease in problem size!

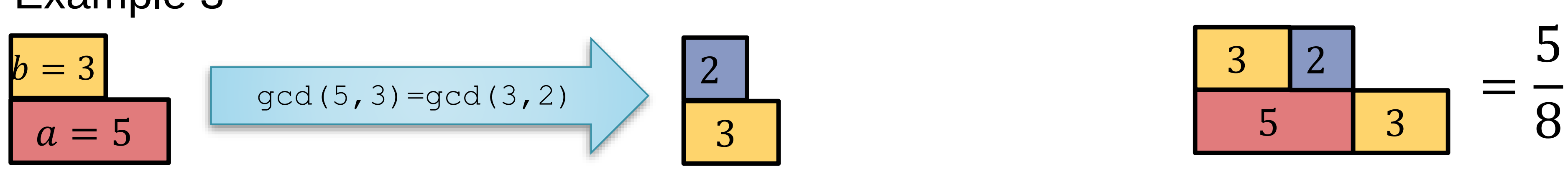


But this can only happen once in the beginning (afterwards $b > a\%b$ guarantees $a > b$)

Example 2



Example 3



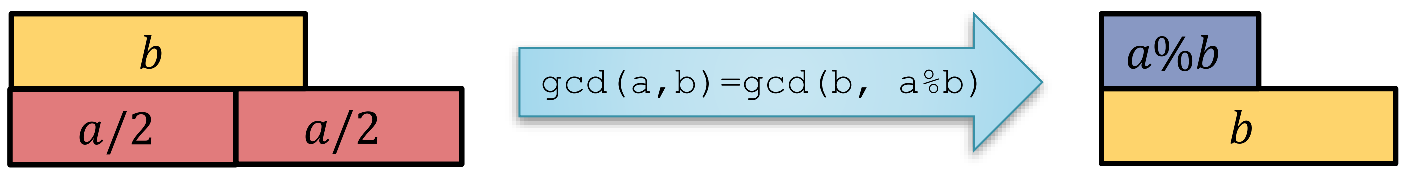
When $a > b$ there is decrease but at varying rate!

Do we need a fixed rate of decrease for logarithmic complexity?

No, just guarantee that reduction factor is always at least some $\alpha > 1$

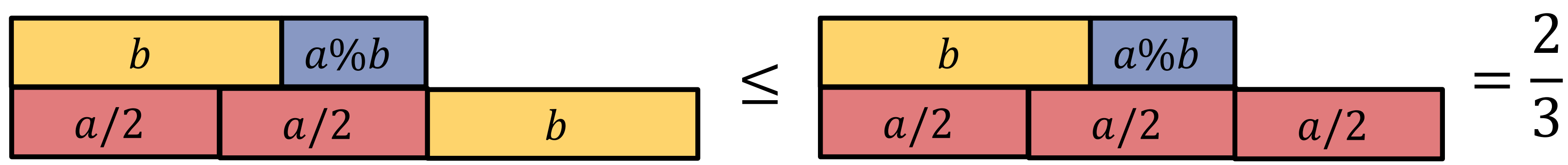
First case: "large b"

Case $b \geq a/2$



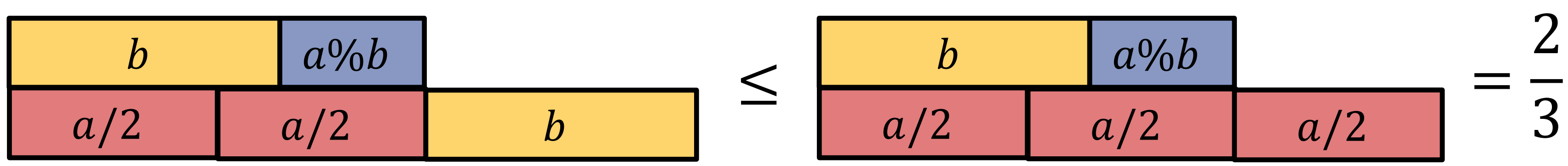
Relative size of decreased problem with large b

Case $b \geq a/2$

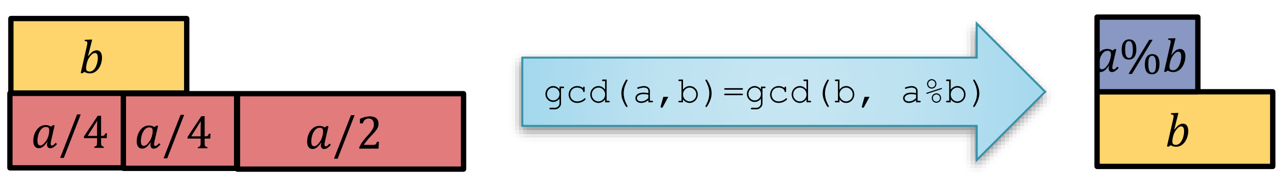


Second case: "small b"

Case $b \geq a/2$

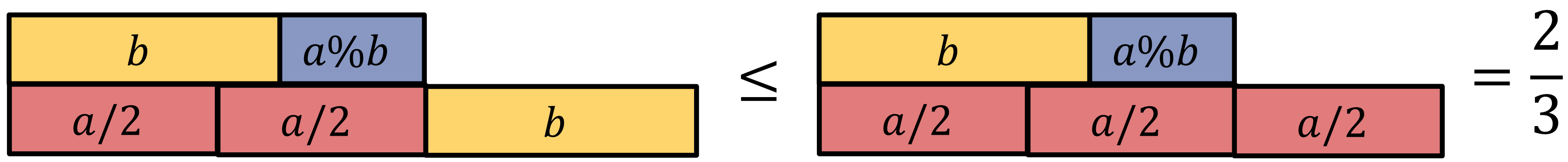


Case $a/2 > b \geq a/4$

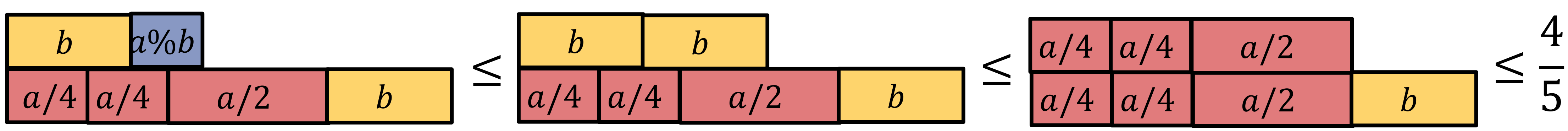


Relative size of decreased problem in second case

Case $b \geq a/2$

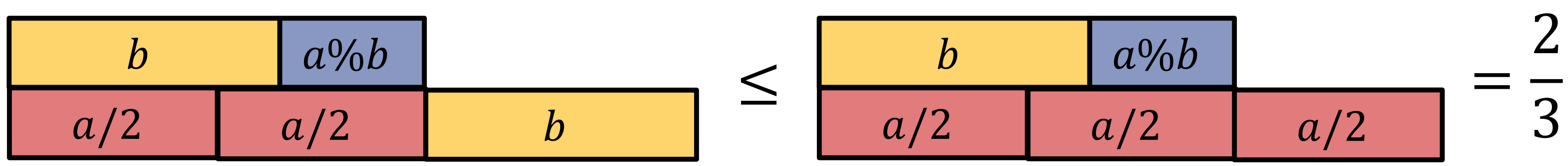


Case $a/2 > b \geq a/4$

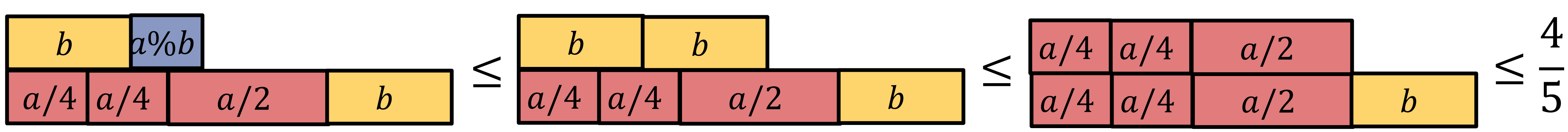


Final case: "tiny b"

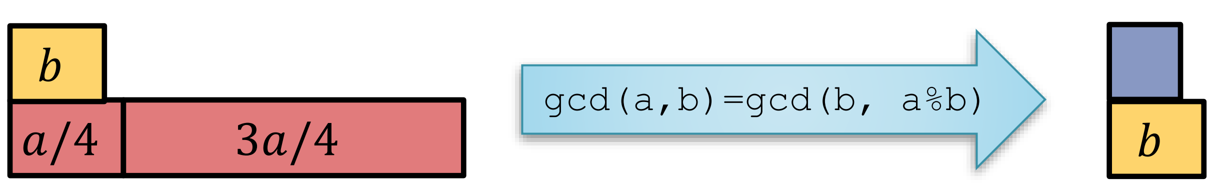
Case $b \geq a/2$



Case $a/2 > b \geq a/4$

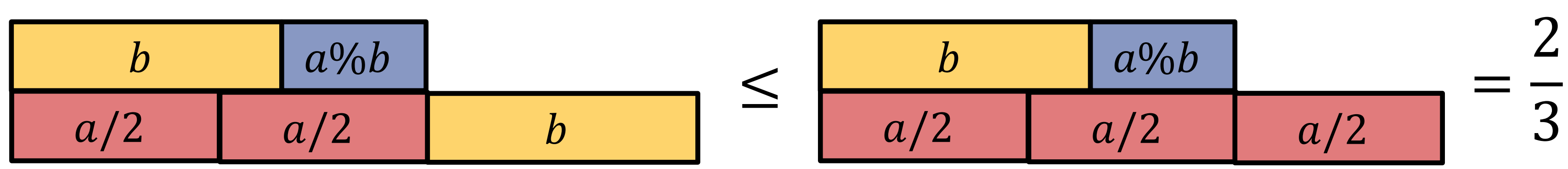


Case $a/4 > b$

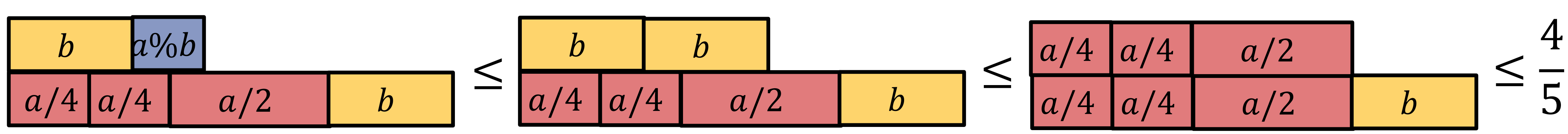


Relative size of decreased problem in final case

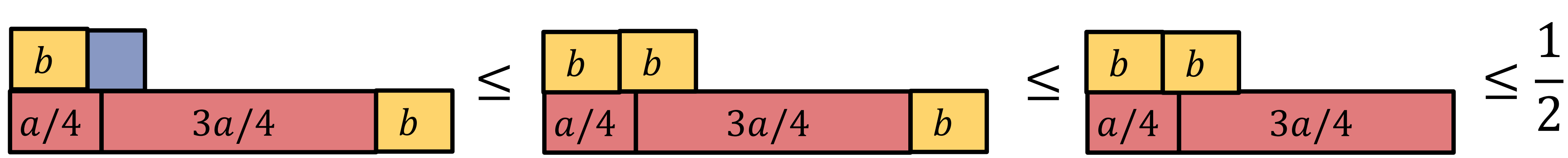
Case $b \geq a/2$



Case $a/2 > b \geq a/4$



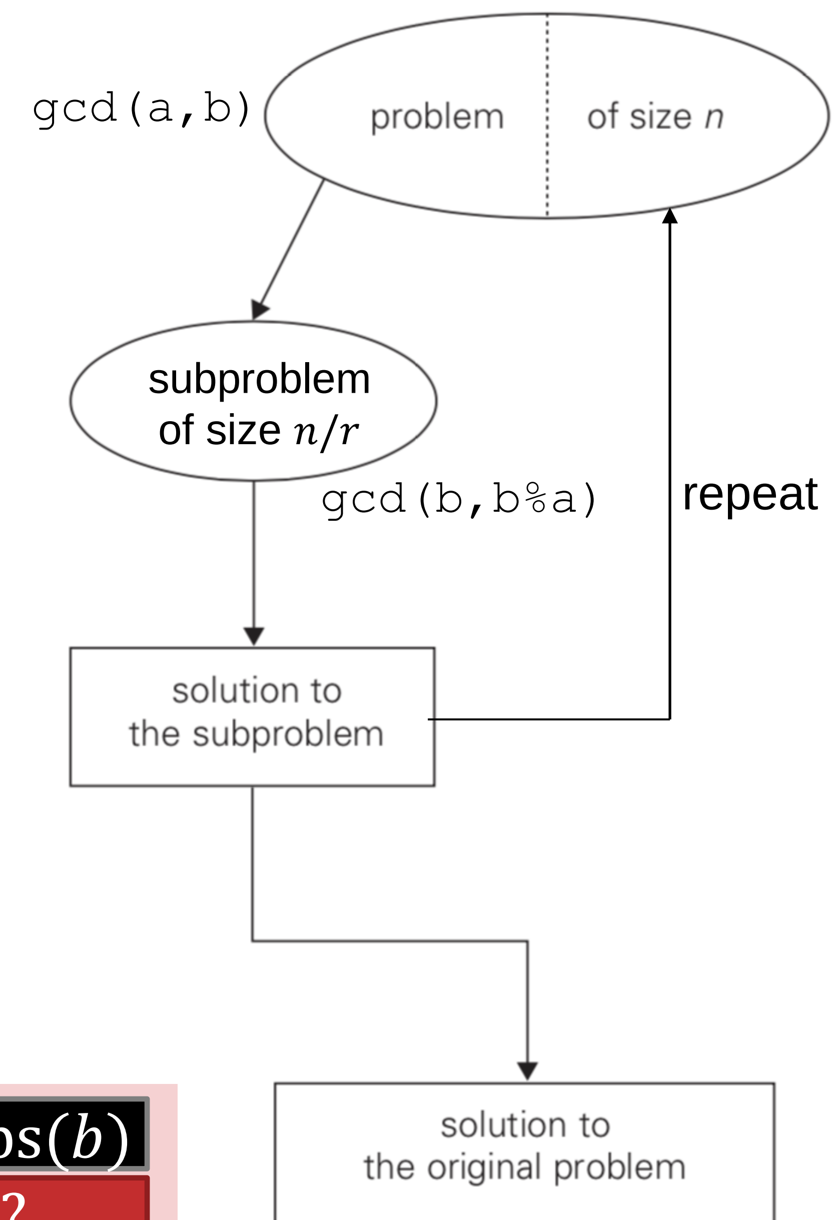
Case $a/4 > b$



In all cases: problem is at least decreased by a rate r of 5/4

Almost identical analysis as for binary search

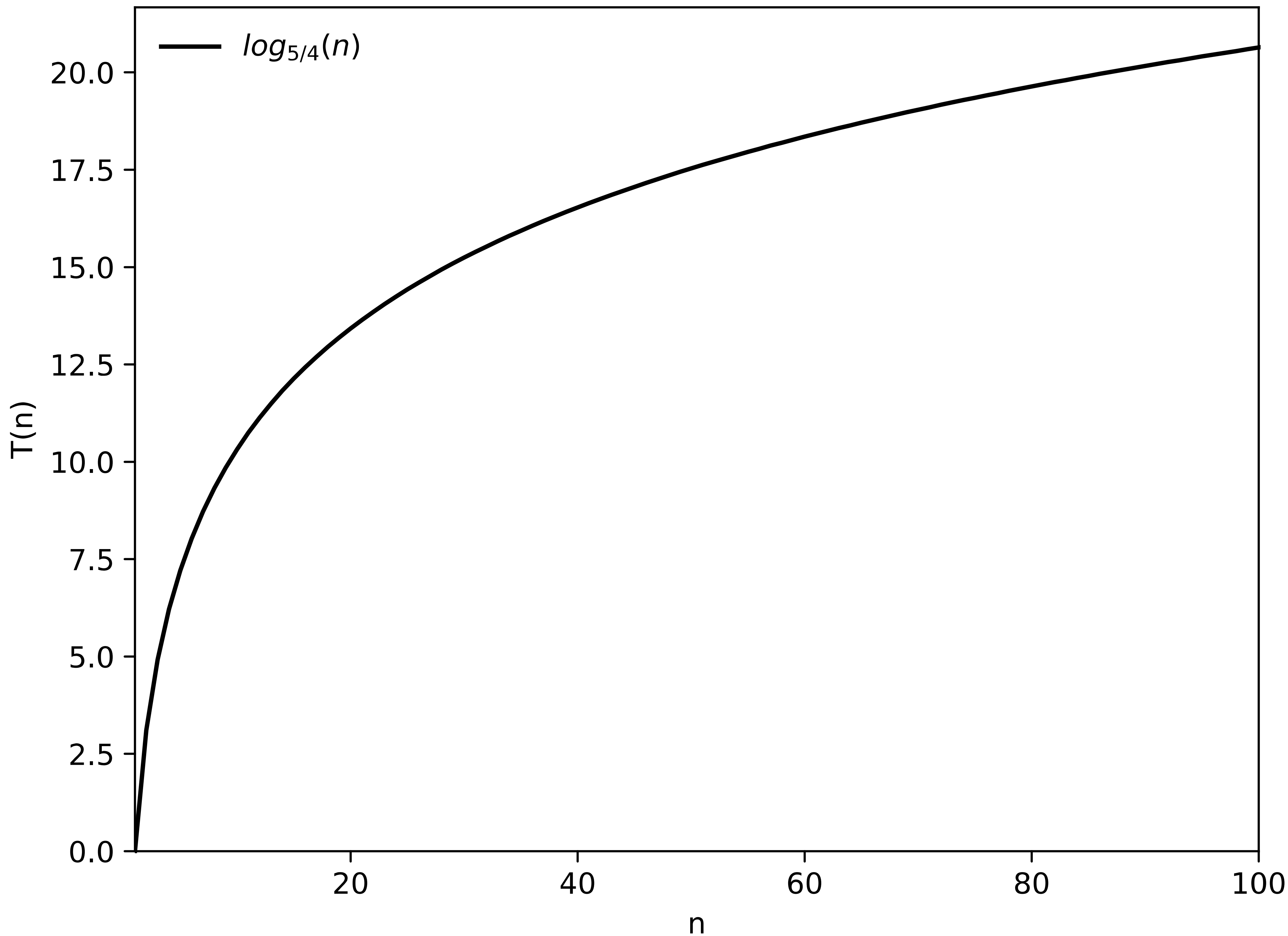
- Let $n_i = a_i + b_i$ be problem size after i iterations of loop
- In the beginning: $n_0 = n$
- Per iteration size is reduced by at least: $n_i = \lceil n_{i-1}/r \rceil$, i.e., $n_i = \lceil n/r^i \rceil$
- After at most $k = \lceil \log_r n \rceil$ iterations: $n_k = \lceil n/r^{\log_r n} \rceil = 1$, i.e., $b_k = 0$ and $a_k = 1$
- So at most $O(\log_r n)$ loop iterations



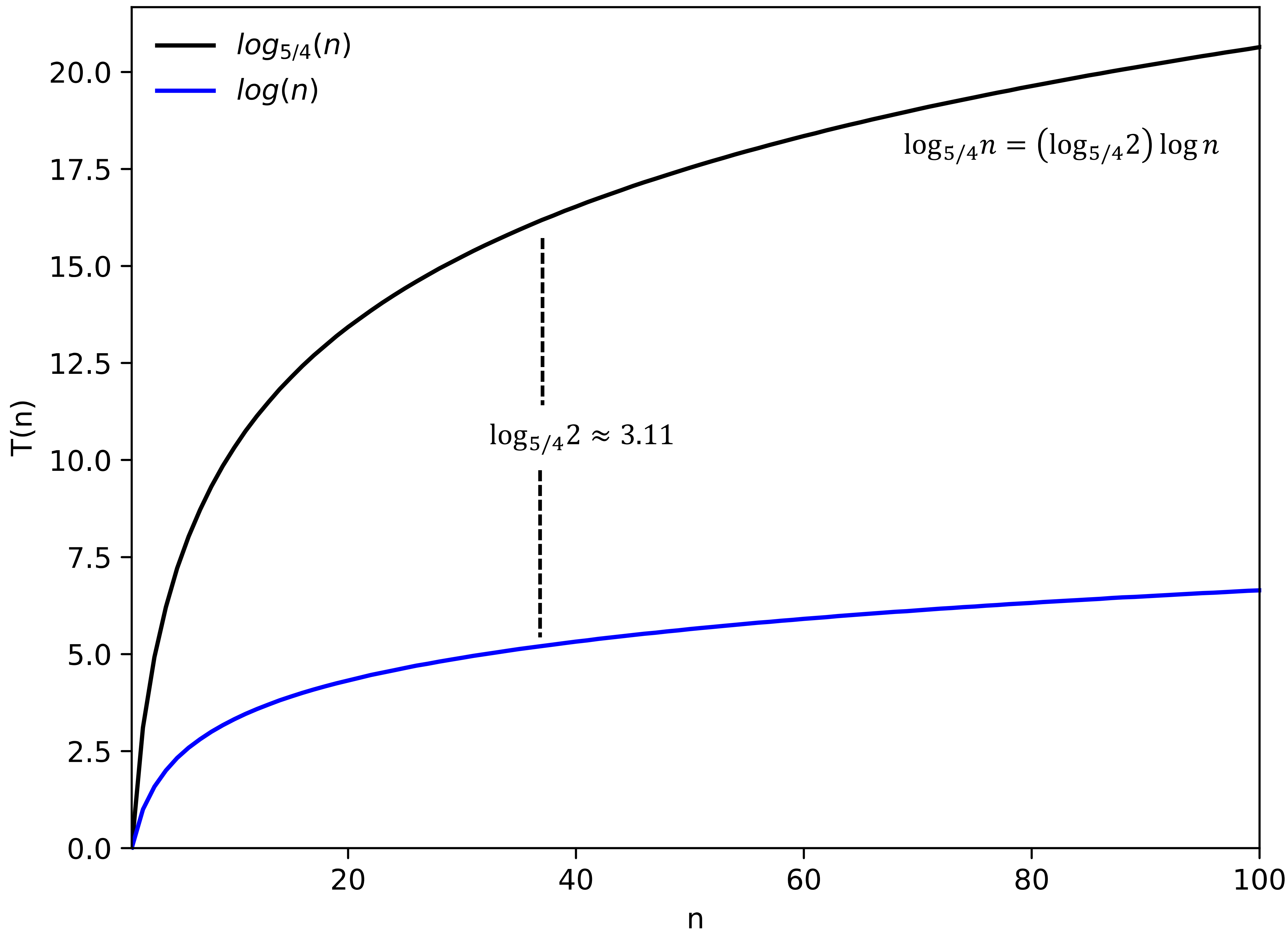

```
def gcd_euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

`def gcd_euclid(a, b):`	$n = \text{abs}(a) + \text{abs}(b)$	
`while b != 0:`	$O(1)$	?
`a, b = b, a % b`	0	?
`return a`	$O(1)$	$O(1)$
	$O(1)$	?

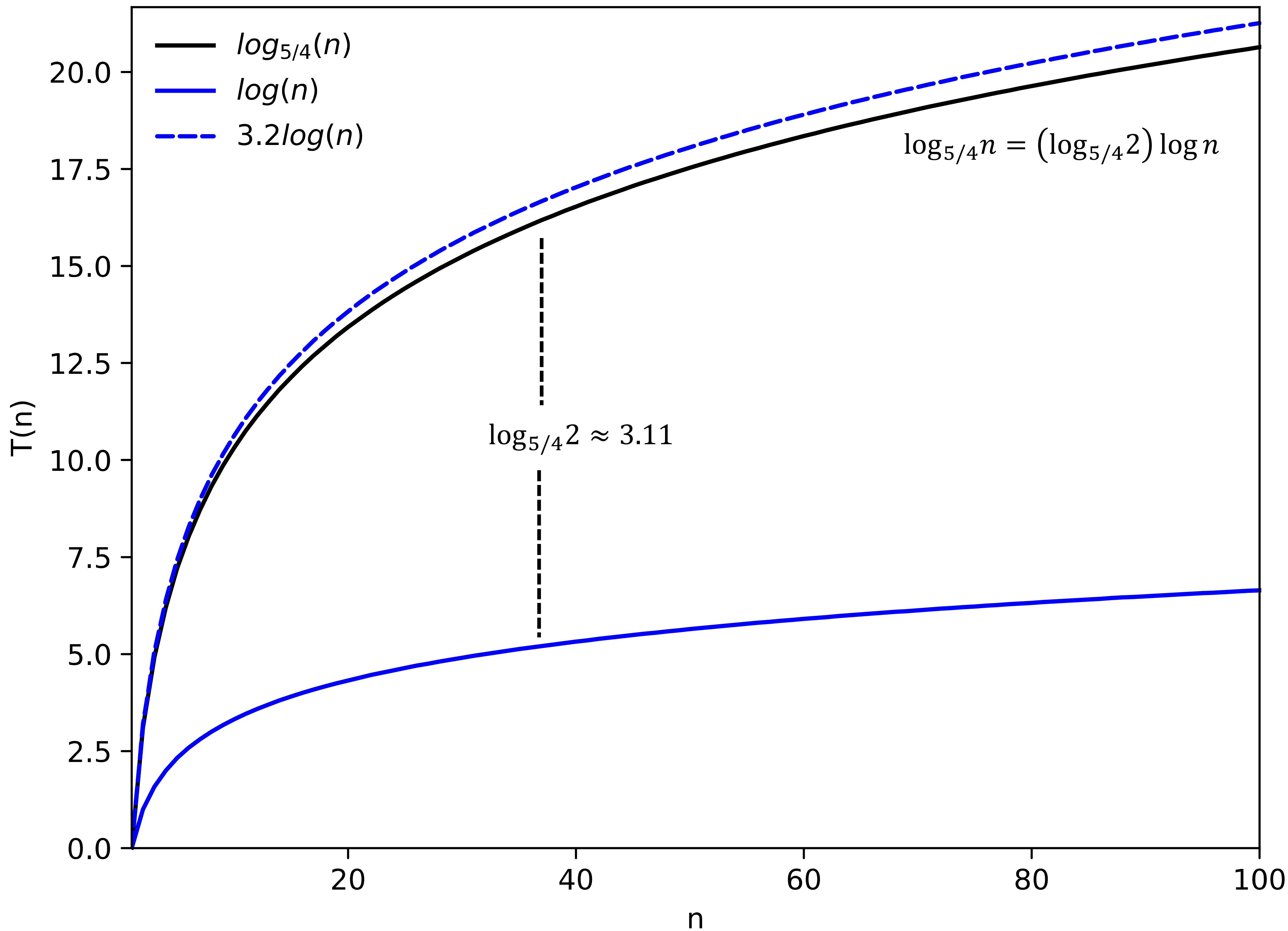
What does this mean in terms of order of growth?



Is order of growth log base 5/4 higher than log base 2?

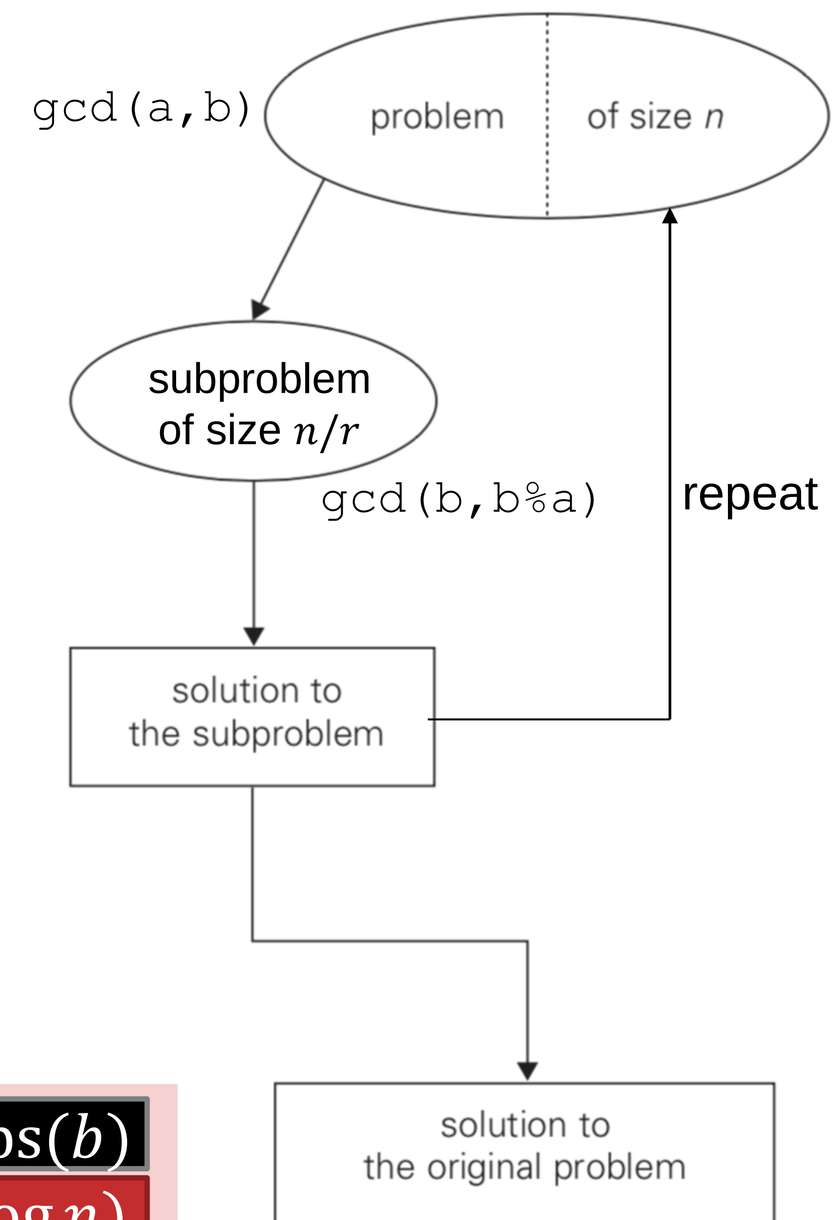


No: $O(\log n) = O(\log_r n)$



Almost identical analysis as for binary search

- Let $n_i = a_i + b_i$ be problem size after i iterations of loop
- In the beginning: $n_0 = n$
- Per iteration size is reduced by at least: $n_i = \lceil n_{i-1}/r \rceil$, i.e., $n_i = \lceil n/r^i \rceil$
- After at most $k = \lceil \log_r n \rceil$ iterations: $n_k = \lceil n/r^{\log_r n} \rceil = 1$, i.e., $b_k = 0$ and $a_k = 1$
- So at most $O(\log_r n)$ loop iterations




```
def gcd_euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Line		
`def gcd_euclid(a, b):`	$n = \mathrm{abs}(a) + \mathrm{abs}(b)$	
`while b != 0:`	$O(1)$	$O(\log n)$
`a, b = b, a % b`	0	$O(\log n)$
`return a`	$O(1)$	$O(1)$
	$O(1)$	$O(\log n)$

Summary

Algorithmic paradigm: decrease-and-conquer

- decreasing problem size by at least some rate r>1 leads to trivial subproblems after logarithmically many reductions
- if not too much overhead: allows to replace linear complexity term by logarithmic term

Binary Search allows logarithmic time look-up of value in sorted sequence

Euclid's Algorithm finds gcd in time logarithmically in sum of input abs(a) + abs(b)

Coming Up

- More examples for algorithm analysis
- Divide and conquer