FIT1045: Algorithms and Programming Fundamentals in Python

Lecture 14

Divide and Conquer

https://www.britannica.com/place/ancient-Greece/Alexander-the-Great

Objectives

Objectives of this lecture are to:

1. Understand design paradigm divide-and-conquer
2. Know how to sort efficiently (Mergesort)

This covers learning outcomes:

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

Overview

1. Divide-and-conquer paradigm
2. Mergesort
3. Linear-time Merge
4. *Quicksort*

Computational Complexity of Insertion Sort

```
def insert(i, lst):
    j = i
    while j > 0 and lst[j-1] > lst[j]:
        lst[j-1], lst[j] = lst[j], lst[j-1]
        j = j - 1
```

$n = \mathrm{i}$	
$O(1)$	$O(1)$
$O(1)$	$O(n)$
$O(0)$	$O(0)$
$O(0)$	$O(n)$
$O(1)$	$O(n)$

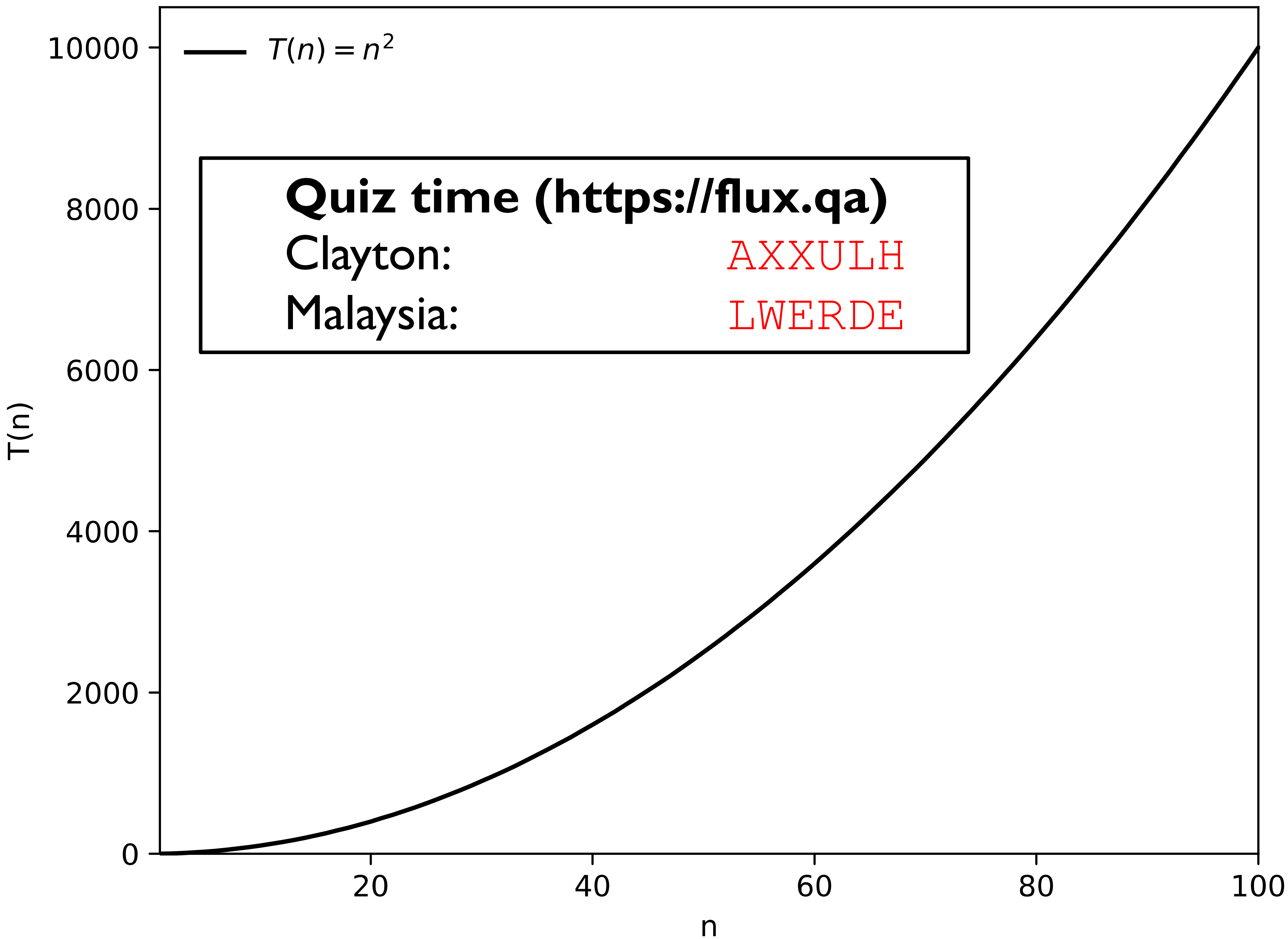
```
def insertion_sort(lst):
    for i in range(1, len(lst)):
        insert(i, lst)
```

$n = \mathrm{len(lst)}$	
$O(n)$	$O(n)$
$O(n)$	$O(n^2)$
$O(n)$	$O(n^2)$

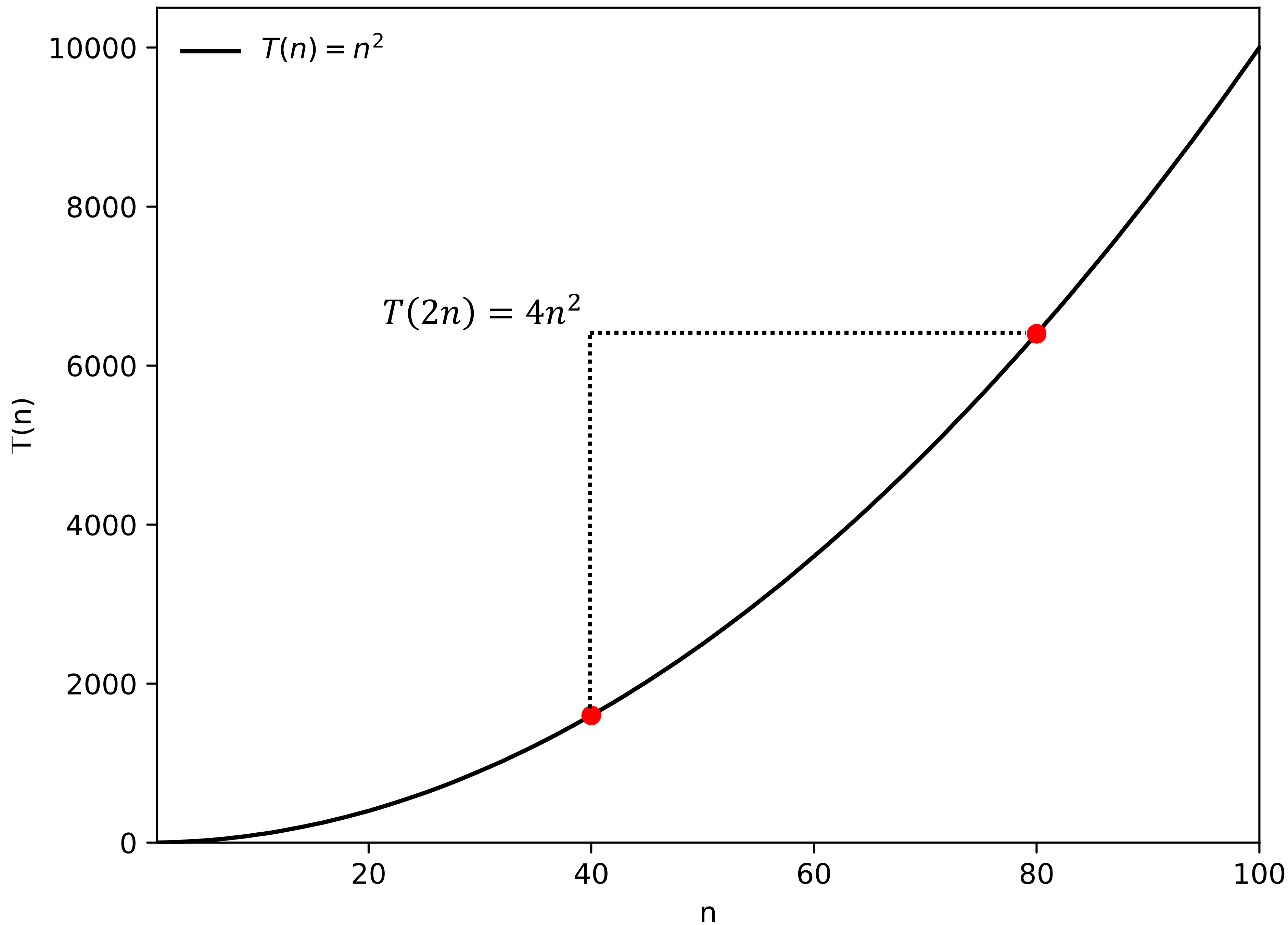
Total cost of `insert` calls

- best case (sorted input): $\overbrace{1 + 1 + 1 + \cdots + 1}^{n \text{ times}} = O(n)$
- worst case (inversely sorted input): $\underbrace{1 + 2 + \cdots + (n-1) + n}_{= (n+1)n/2 \text{ as previously}} = O(n^2)$

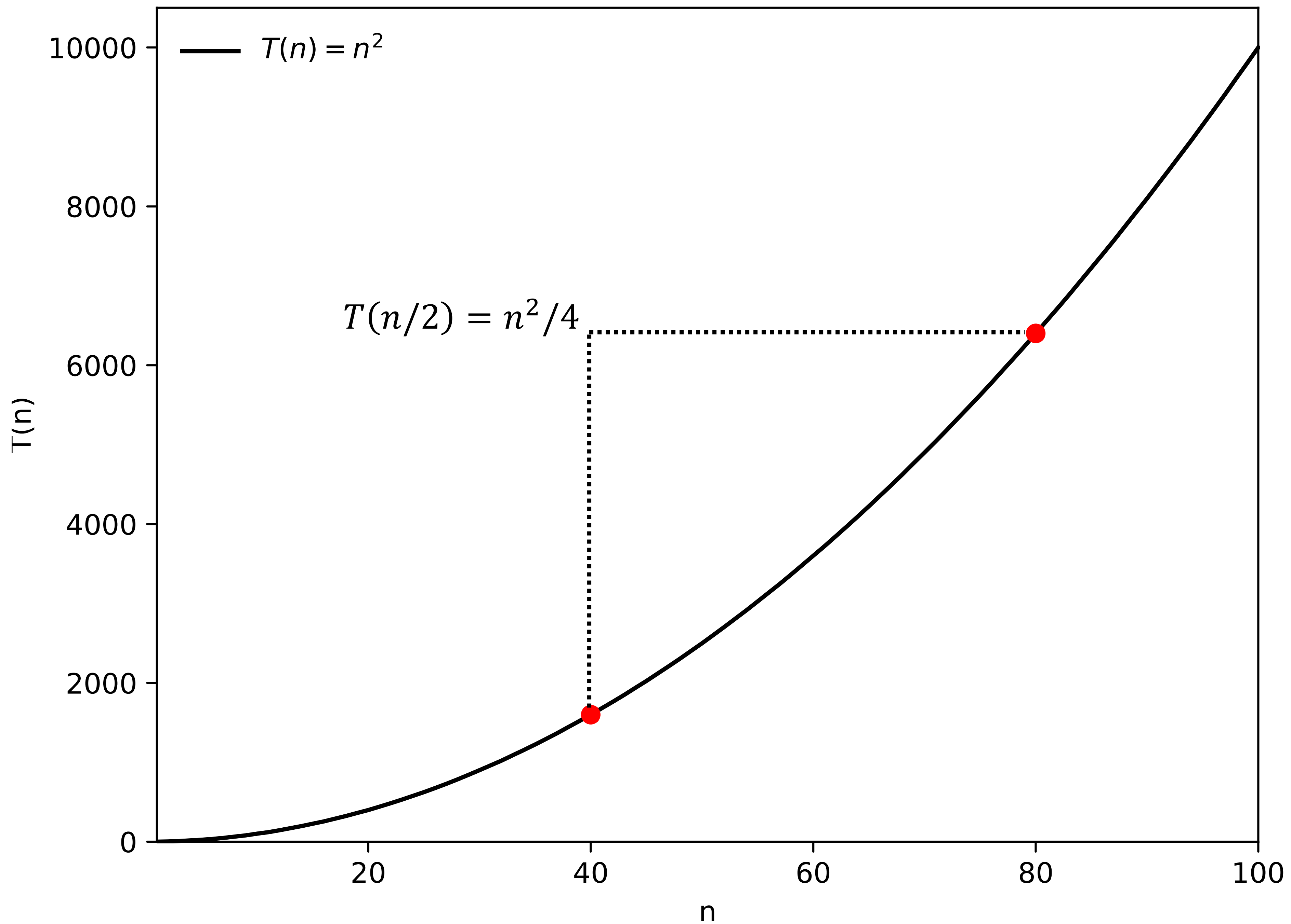
Quadratic complexity



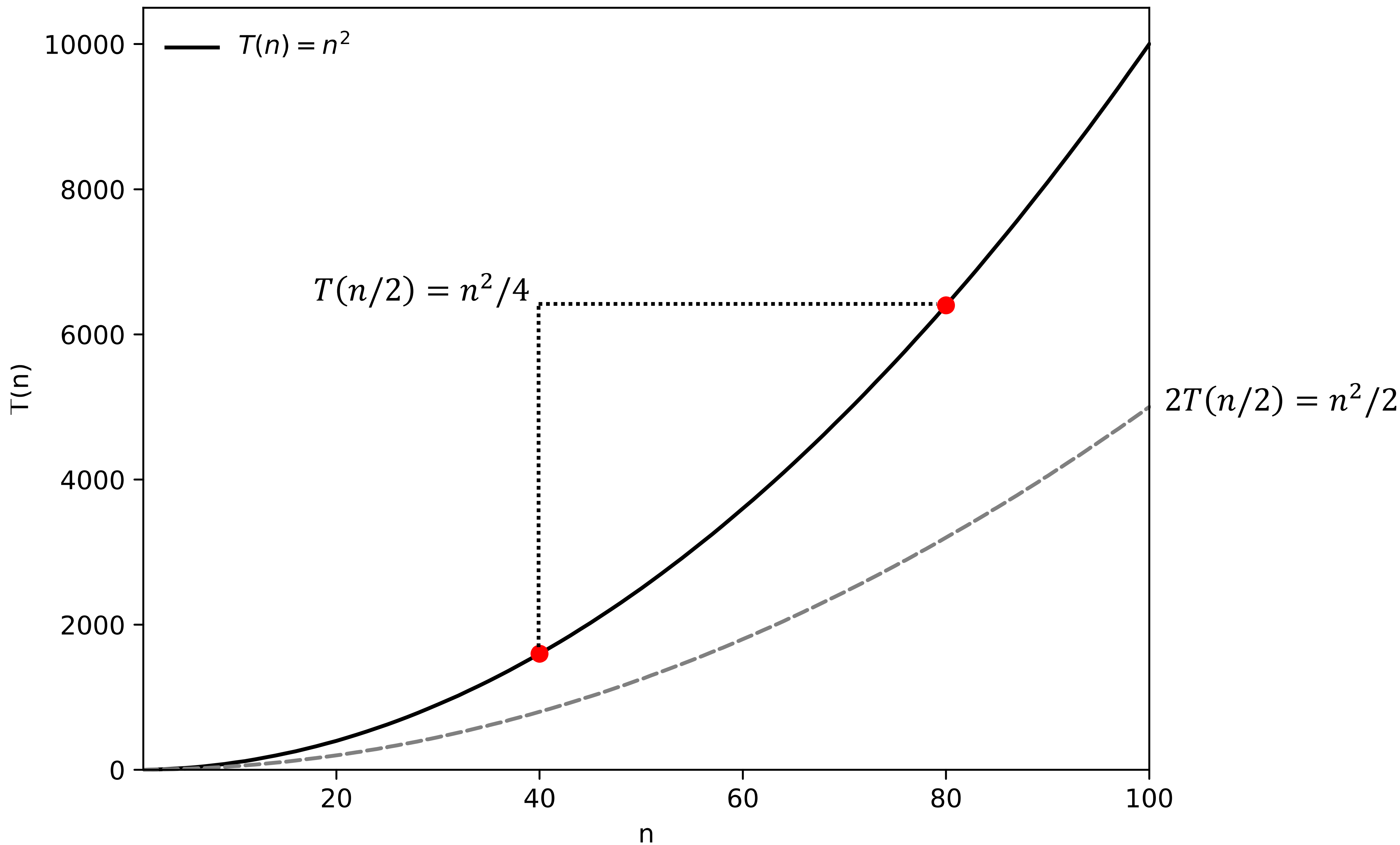
Quadratic complexity means that 2x input leads to 4x computation time



Conversely: half input means only quarter computation time

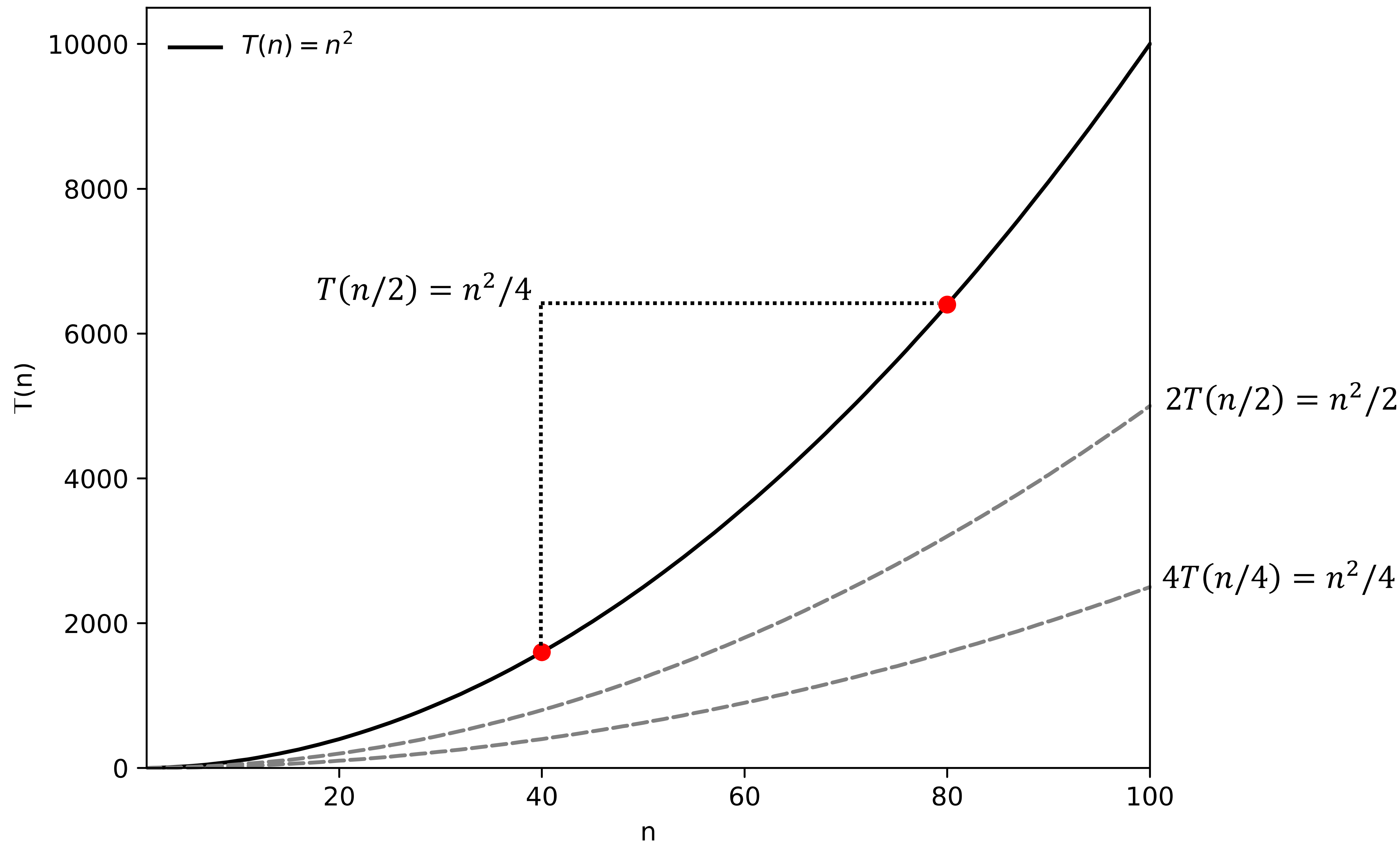


So solving two inputs of half size each can be done in half the time!

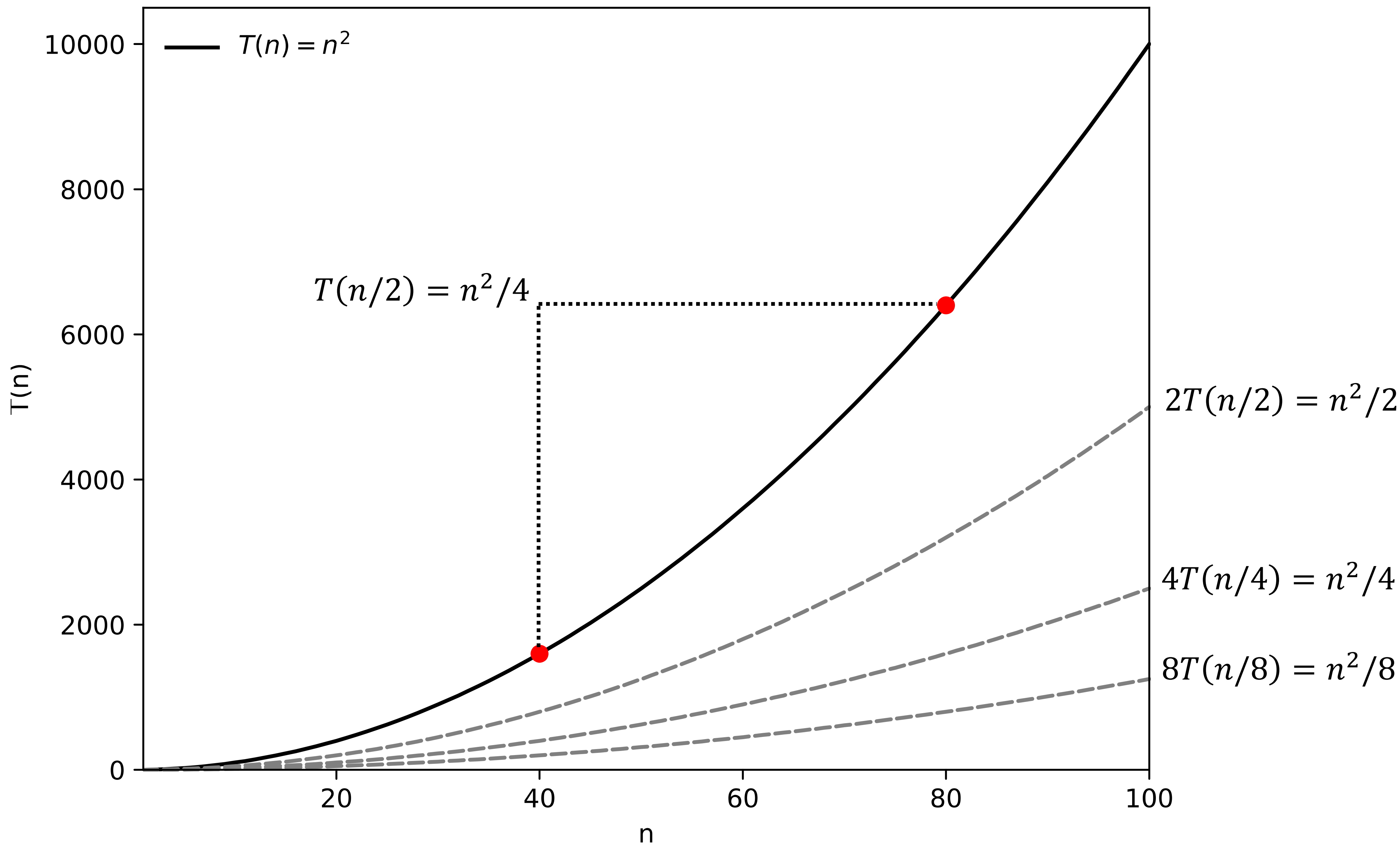


$$T(n/2) + T(n/2) = 2n^2/4 = n^2/2$$

Or four inputs of quarter size can be solved in a quarter of the time



...and so on



$T(n) = n^2$
$T(n/2) = n^2/4$
$2T(n/2) = n^2/2$
$4T(n/4) = n^2/4$
$8T(n/8) = n^2/8$
10000
8000
6000
4000
2000
0
20
40
60
80
100
n
T(n)

This motivates *divide-and-conquer* strategy

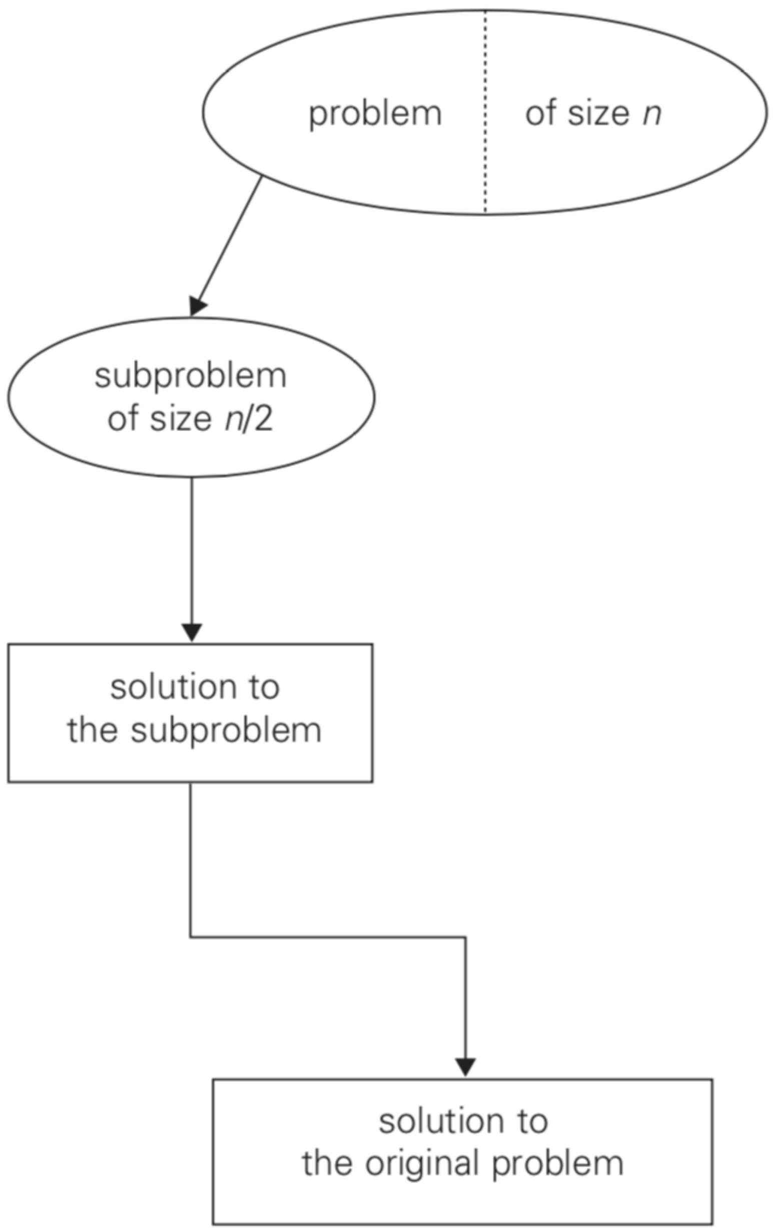


illustration: [Levitin, p. 133]

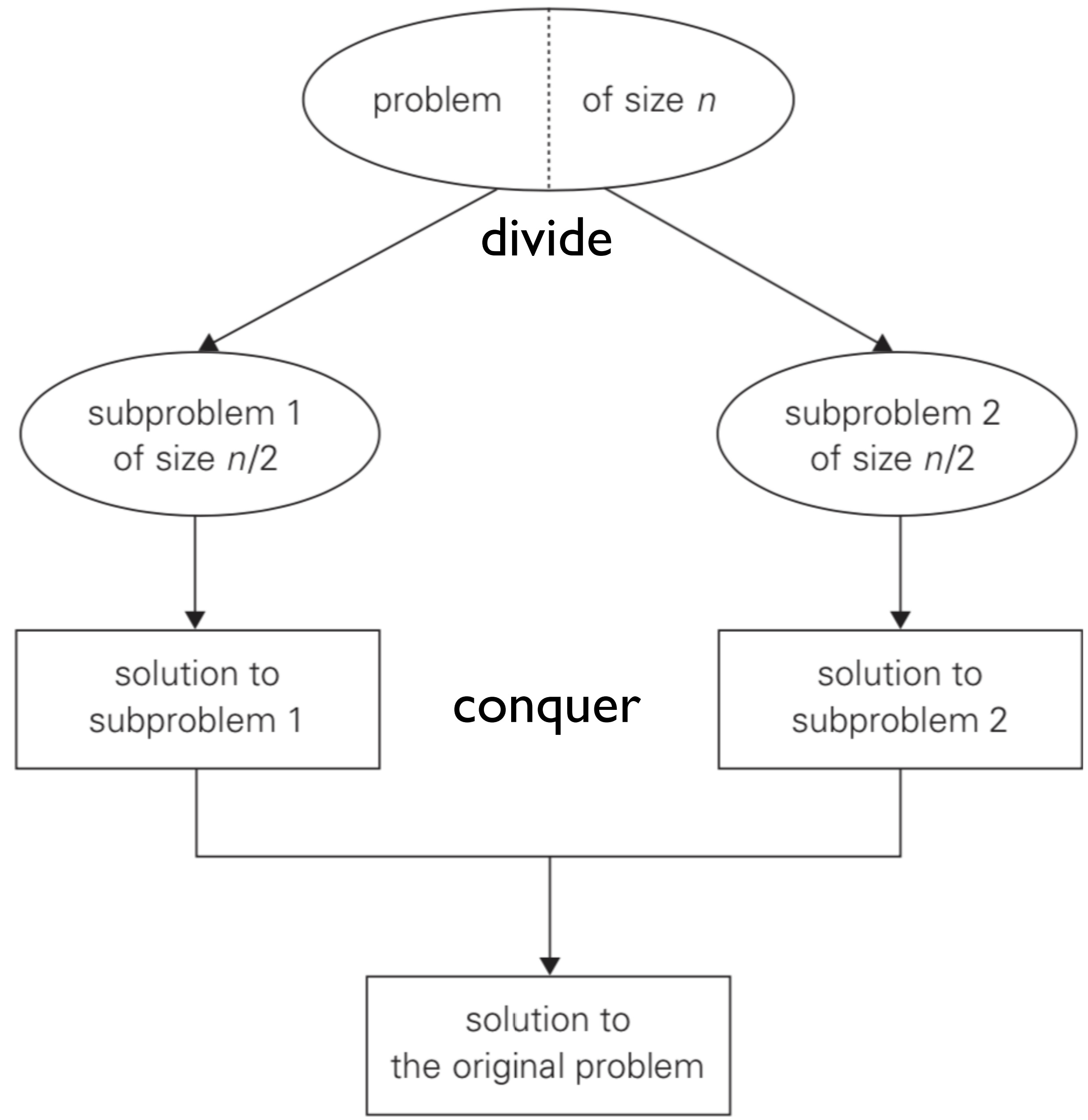
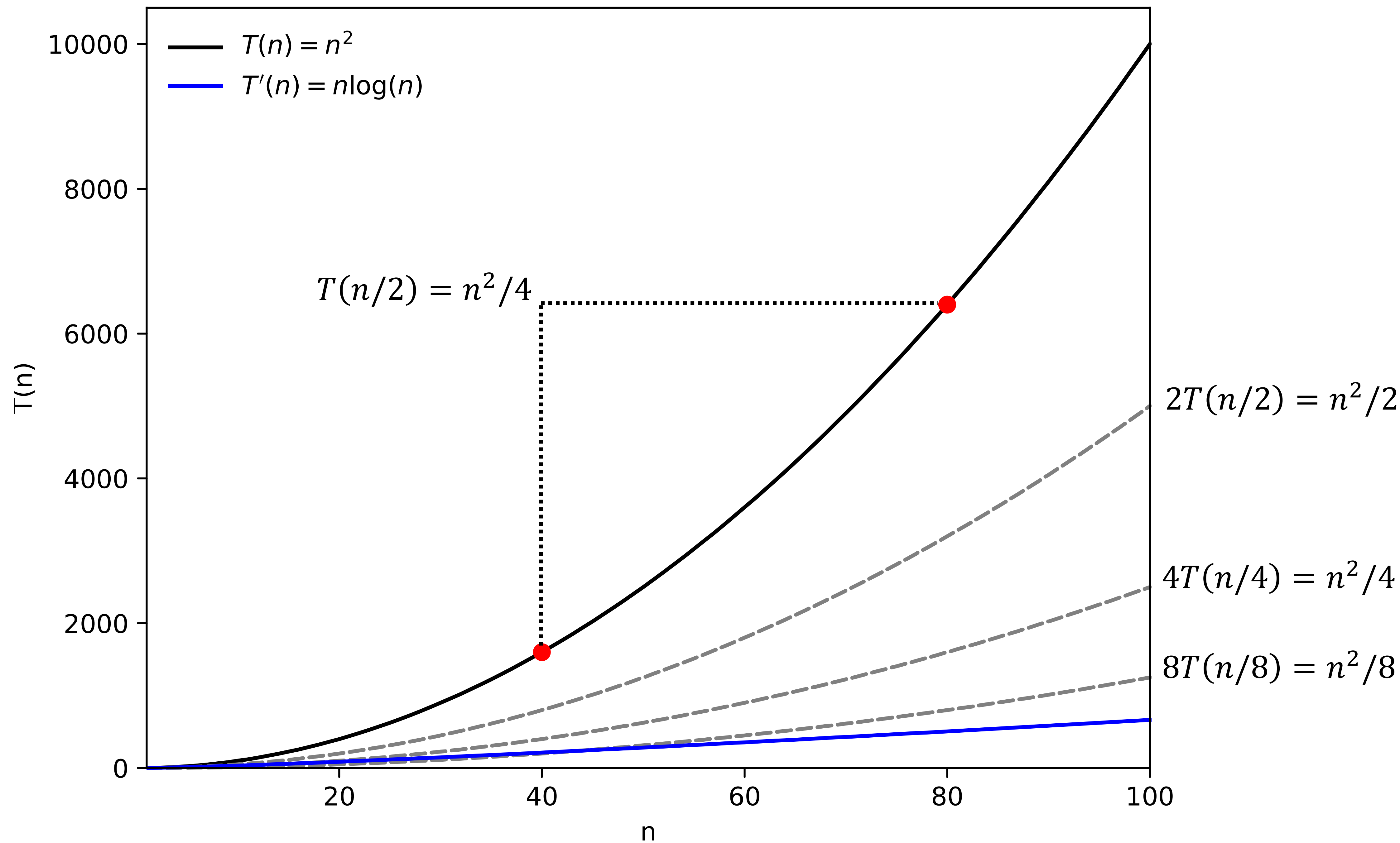


illustration: [Levitin, p. 170]

Goal: getting to "linearithmic" time by dividing log n times



$2^{\log n}T\left(n/2^{\log n}\right) = n^2/2^{\log n} = n$ plus cost of putting solutions together

Overview

1. Divide-and-conquer paradigm
2. Mergesort
3. Linear-time Merge
4. *Quicksort*

Application to sorting

b	f	h	a	d	k	g	j	e	c	i
0	1	2	3	4	5	6	7	8	9	10

consider as two separate sorting problems

b	f	h	a	d	k
0	1	2	3	4	5

g	j	e	c	i
6	7	8	9	10

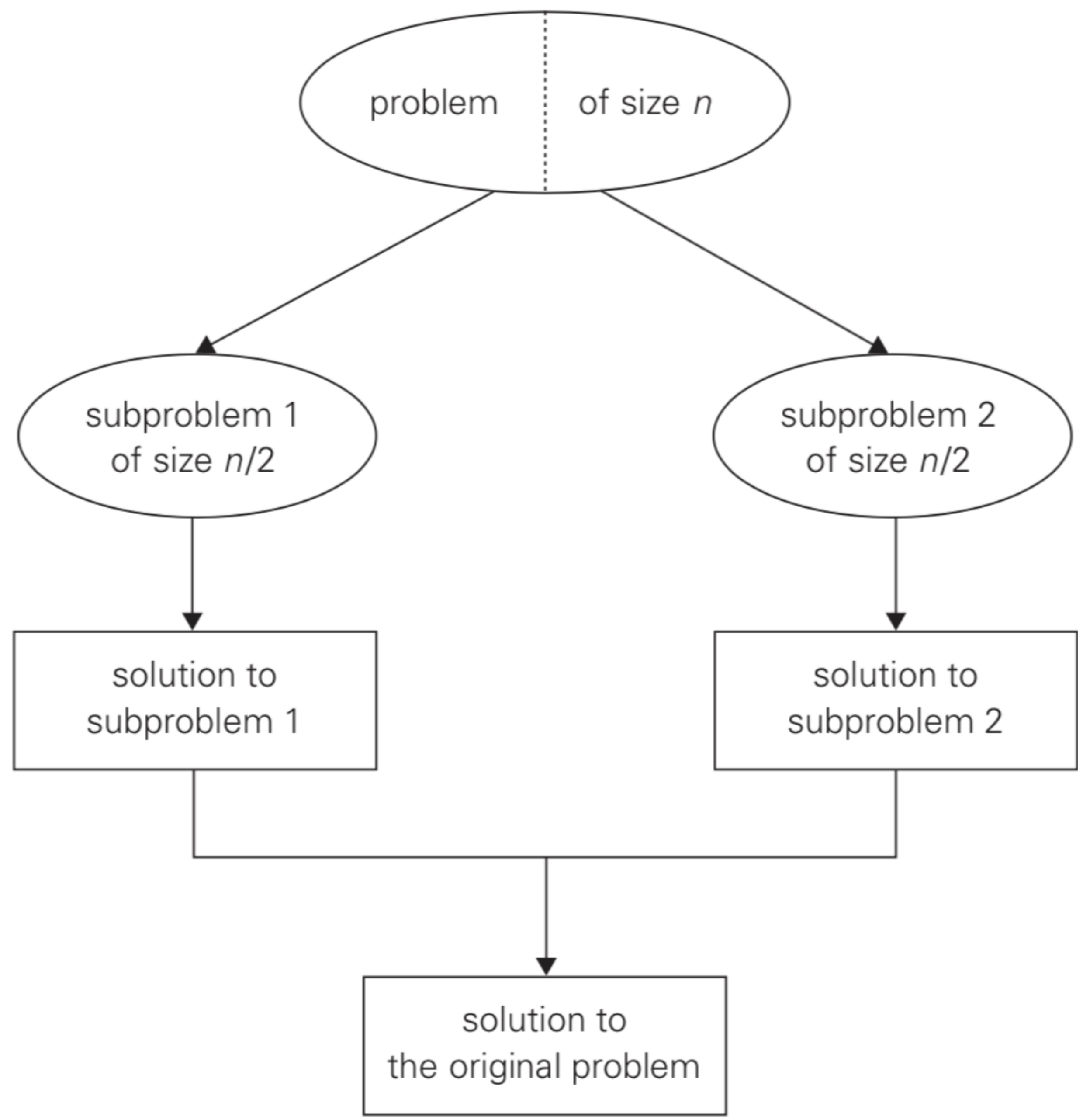
solve independently

a	b	d	f	h	k
0	1	2	3	4	5

c	e	g	j	i
6	7	8	9	10

merge solutions

a	b	c	d	e	f	g	h	i	j	k
0	1	2	3	4	5	6	7	8	9	10



How to solve subproblems

b	f	h	a	d	k	g	j	e	c	i
0	1	2	3	4	5	6	7	8	9	10

consider as two separate sorting problems

b	f	h	a	d	k
0	1	2	3	4	5

g	j	e	c	i
6	7	8	9	10

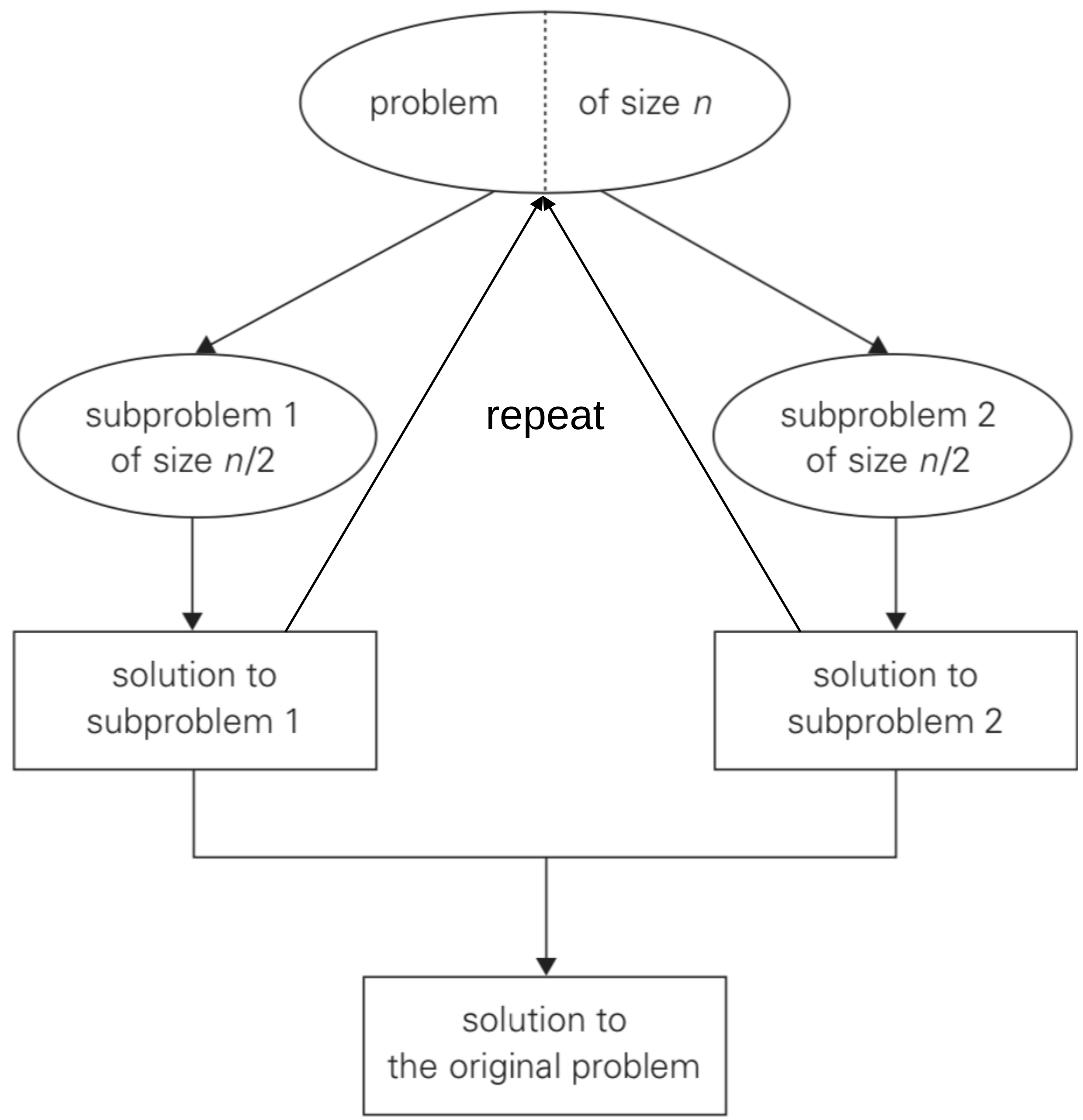
solve independently

a	b	d	f	h	k
0	1	2	3	4	5

c	e	g	j	i
6	7	8	9	10

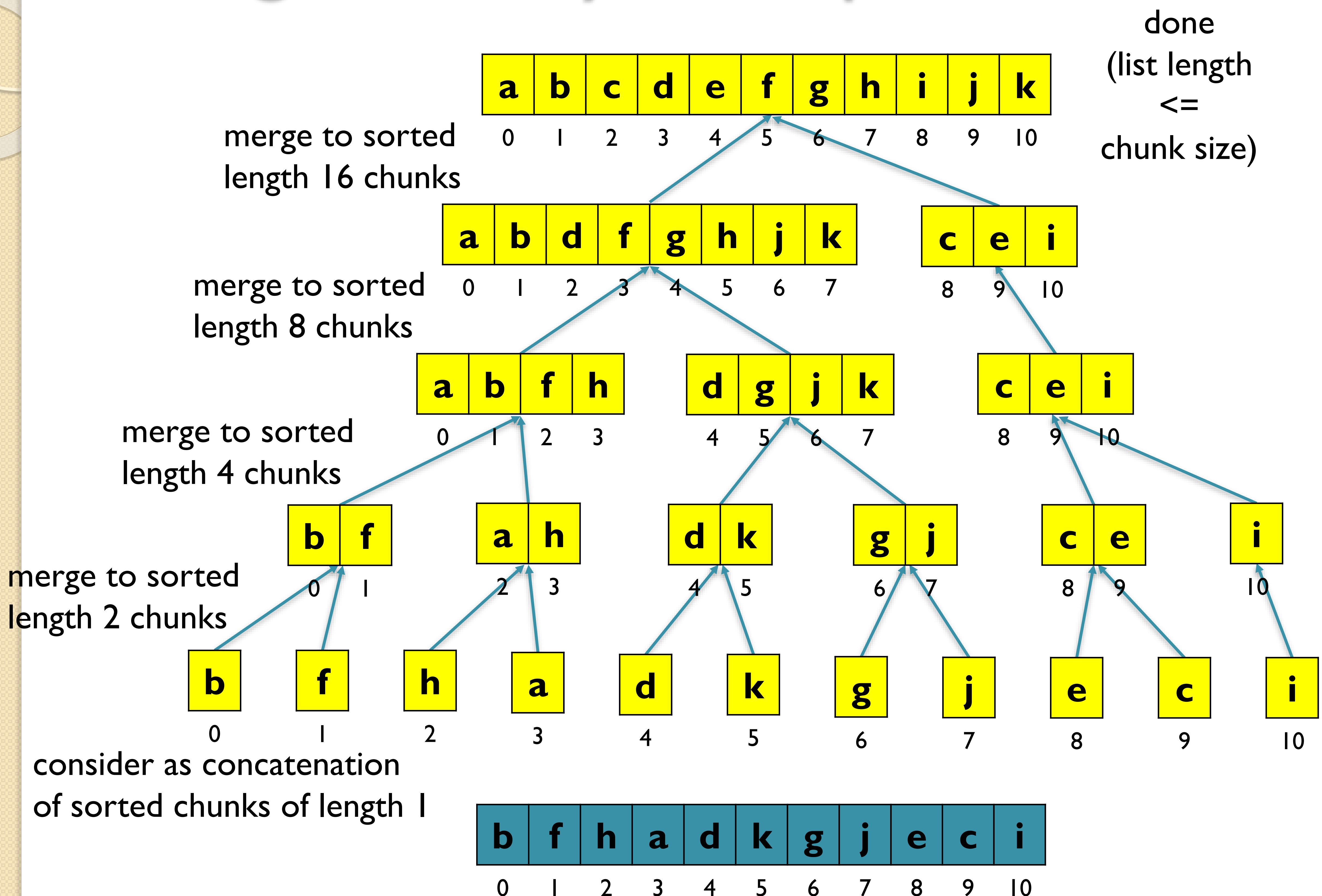
merge solutions

a	b	c	d	e	f	g	h	i	j	k
0	1	2	3	4	5	6	7	8	9	10



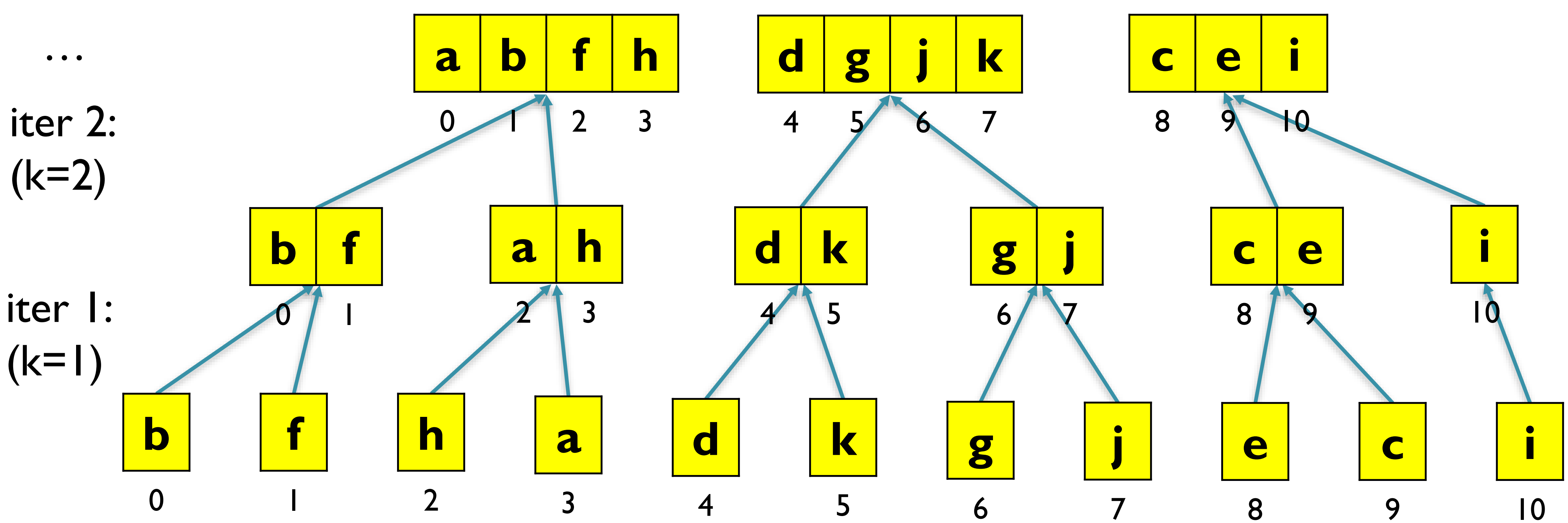
Insertion Sort? What would be resulting time complexity?

Merge Sort by Example



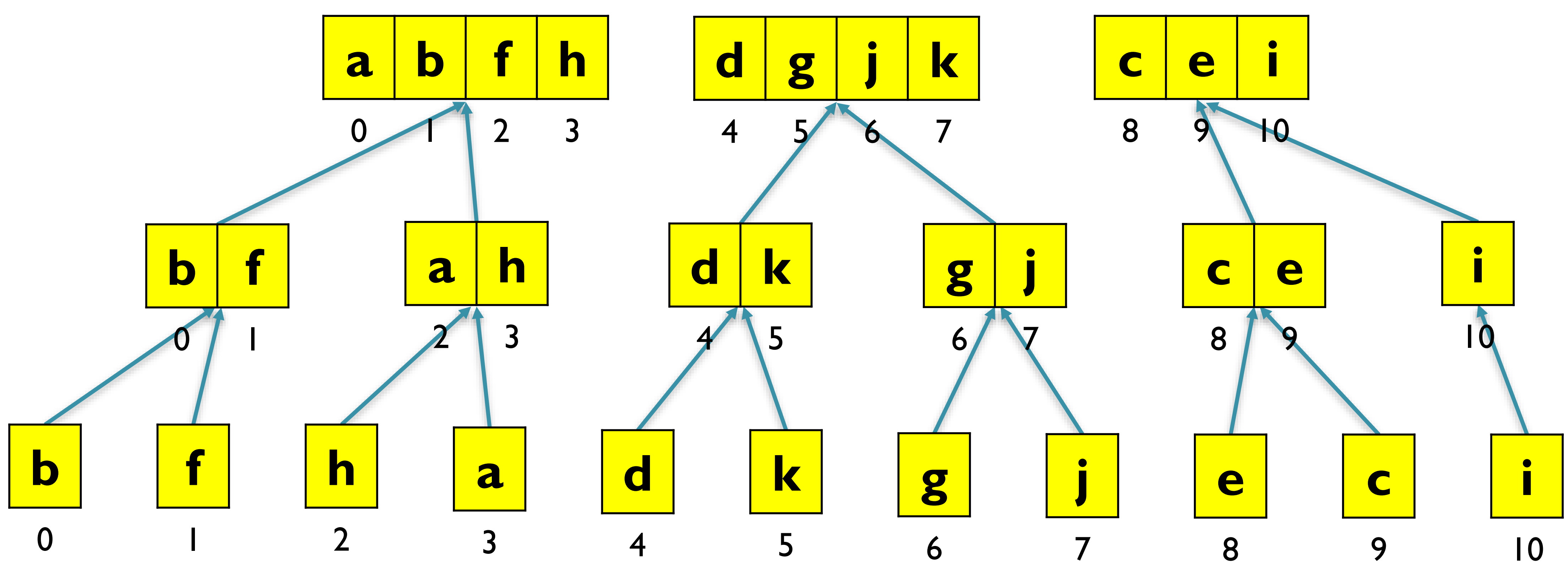
Mergesort in Python

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:

        # set lst to concatenation of merged chunks of
        # size k


        k = 2 * k
    return ls
```



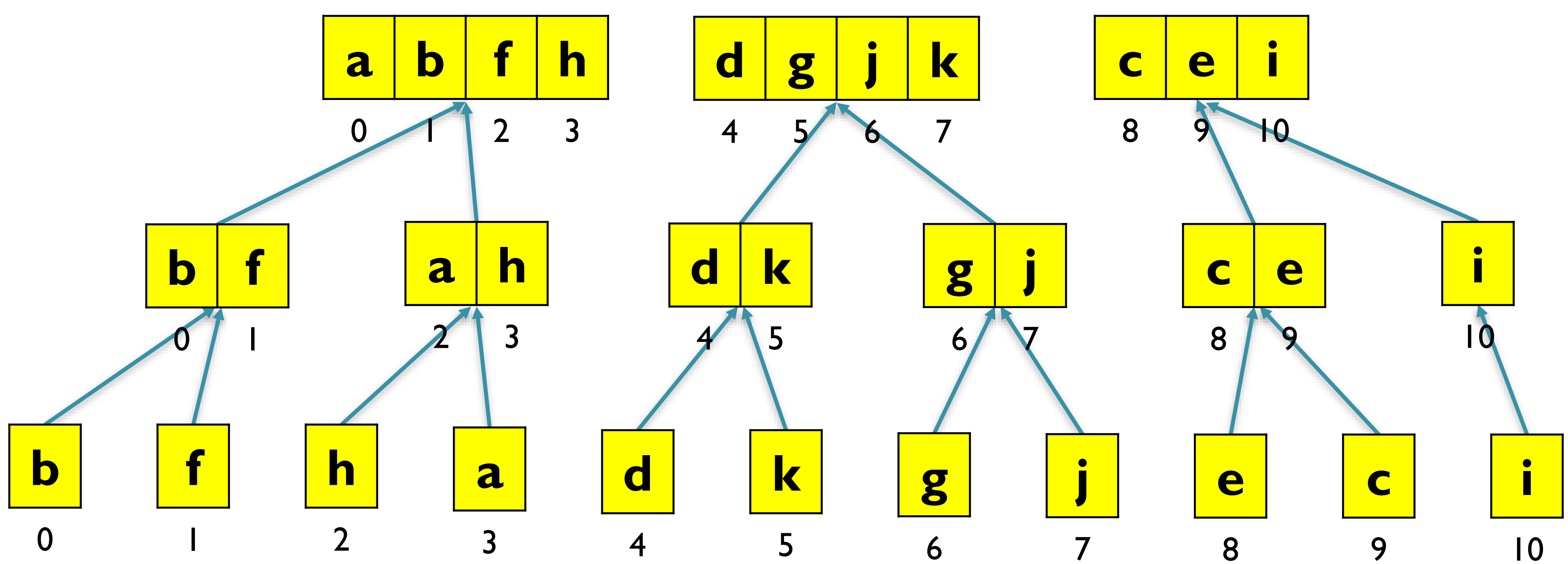
Mergesort in Python

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        # for all pairs of consecutive chunks
        # ls[a:b], ls[b:c]:
        #       merge and concatenate to nxt
        ls = nxt
        k = 2 * k
    return ls
```



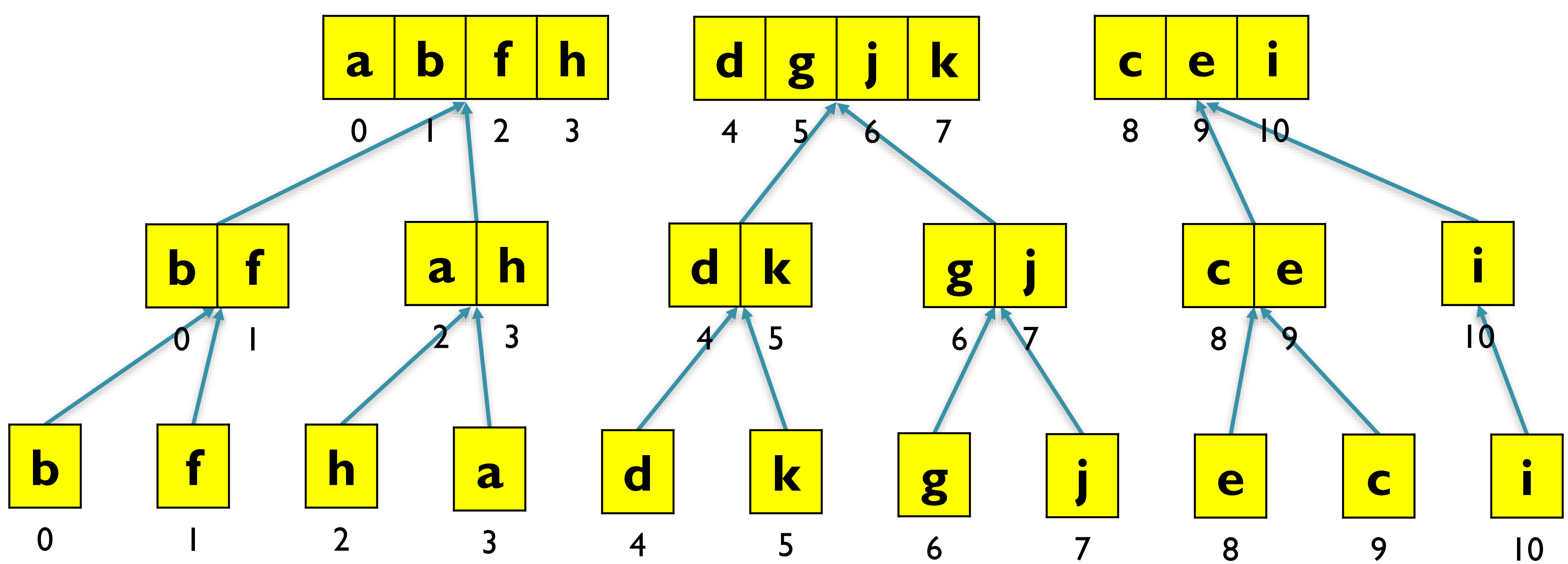
Mergesort in Python

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        # for all pairs of consecutive chunks
        # ls[a:b], ls[b:c]:
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```



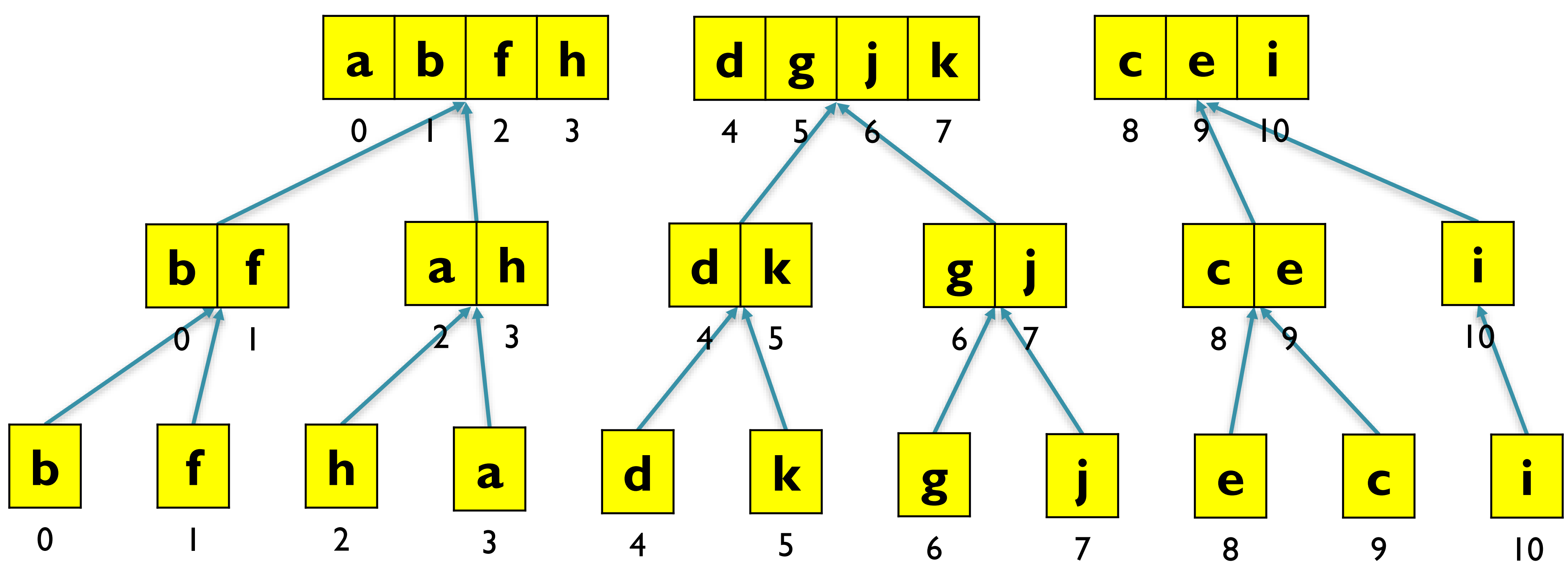
Mergesort in Python

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
        # ls[a:b], ls[b:c]:
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```



Mergesort in Python

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
            b, c = a + k, a + 2*k
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```



Complexity of Mergesort

```python
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
            b, c = a + k, a + 2*k
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```

`k, n = 1, len(ls)`	$O(0)$	$O(0)$
`while k < n:`	$O(\log n)$	$O(\log n)$
`nxt = []`	$O(\log n)$	$O(\log n)$
`for a in range(0, n, 2*k):`	?	?
`ls = nxt`	$O(\log n)$	$O(\log n)$
`k = 2 * k`	$O(\log n)$	$O(\log n)$
`return ls`	$O(0)$	$O(0)$

- Let k_i be chunk size after i iterations of loop
- In the beginning: $k_0 = 1$
- In every iteration size is doubled: $k_i = 2k_{i-1}$, i.e., $k_i = 2^i$
- After $l = \lceil \log_2 n \rceil$ iterations: $k_l = 2^{\lceil \log_2 n \rceil} \geq n$
- So at most $O(\log_2 n)$ outer loop iterations

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

Complexity of Mergesort

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
            b, c = a + k, a + 2*k
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```

`k, n = 1, len(ls)`	$O(0)$	$O(0)$
`while k < n:`	$O(\log n)$	$O(\log n)$
`nxt = []`	$O(\log n)$	$O(\log n)$
`for a in range(0, n, 2*k):`	?	?
`ls = nxt`	$O(\log n)$	$O(\log n)$
`k = 2 * k`	$O(\log n)$	$O(\log n)$
`return ls`	$O(0)$	$O(0)$

- Let k_i be chunk size after i iterations of loop
- In the beginning: $k_0 = 1$
- In every iteration size is doubled: $k_i = 2k_{i-1}$, i.e., $k_i = 2^i$
- After $l = \lceil \log_2 n \rceil$ iterations: $k_l = 2^{\lceil \log_2 n \rceil} \geq n$
- So at most $O(\log_2 n)$ outer loop iterations

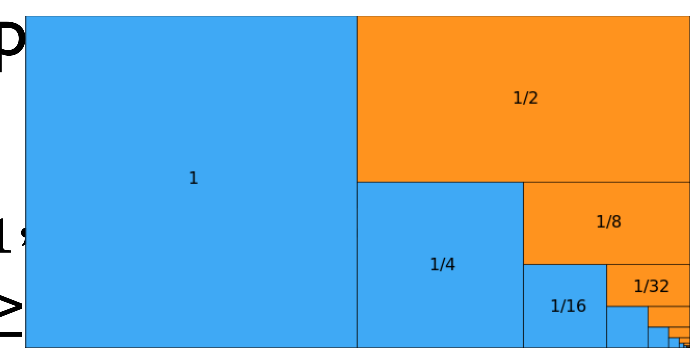
- Inner loop iterations: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots$

Complexity of Mergesort

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
            b, c = a + k, a + 2*k
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```

`k, n = 1, len(ls)`	$O(0)$	$O(0)$
`while k < n:`	$O(\log n)$	$O(\log n)$
`nxt = []`	$O(\log n)$	$O(\log n)$
`for a in range(0, n, 2*k):`	$O(n)$	$O(n)$
`b, c = a + k, a + 2*k`	$O(n)$	$O(n)$
`nxt += merge(ls[a:b], ls[b:c])`	?	?
`ls = nxt`	$O(\log n)$	$O(\log n)$
`k = 2 * k`	$O(\log n)$	$O(\log n)$
`return ls`	$O(0)$	$O(0)$

- Let k_i be chunk size after i iterations of loop
- In the beginning: $k_0 = 1$
- In every iteration size is doubled: $k_i = 2k_{i-1}$
- After $l = \lceil \log_2 n \rceil$ iterations: $k_l = 2^{\lceil \log_2 n \rceil} \geq$
- So at most $O(\log_2 n)$ outer loop iterations



- Inner loop iterations: $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots < n \overbrace{\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)}^{= 1 \text{ (geometric series)}} = n$

Overview

1. Divide-and-conquer paradigm
2. Mergesort
3. Linear-time Merge
4. *Quicksort*

Complexity of Mergesort

```python
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
            b, c = a + k, a + 2*k
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```

Line		
`k, n = 1, len(ls)`	$O(0)$	$O(0)$
`while k < n:`	$O(\log n)$	$O(\log n)$
`nxt = []`	$O(\log n)$	$O(\log n)$
`for a in range(0, n, 2*k):`	$O(n)$	$O(n)$
`b, c = a + k, a + 2*k`	$O(n)$	$O(n)$
`nxt += merge(ls[a:b], ls[b:c])`	?	?
`ls = nxt`	$O(\log n)$	$O(\log n)$
`k = 2 * k`	$O(\log n)$	$O(\log n)$
`return ls`	$O(0)$	$O(0)$

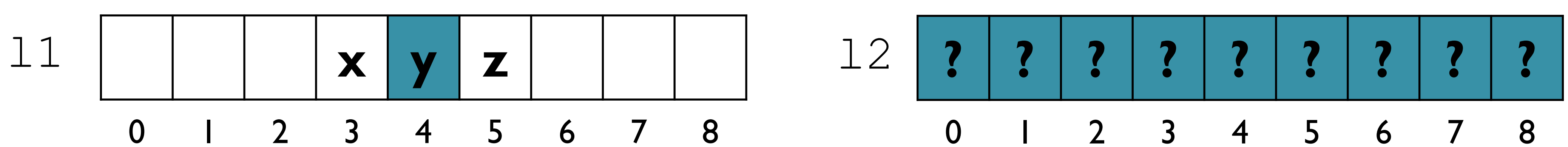
- What is total merging cost?
- Define complexity of `merge` in terms of `len(ls1)+len(ls2)`
- Need to aim for linear time complexity

Merging by repeated insertion

```python
def insertion_merge(l1, l2):
    res = l1 + l2
    n1, n2 = len(l1), len(l2)
    for i in range(n1, n1+n2):
        insert(i, res)
    return res
```

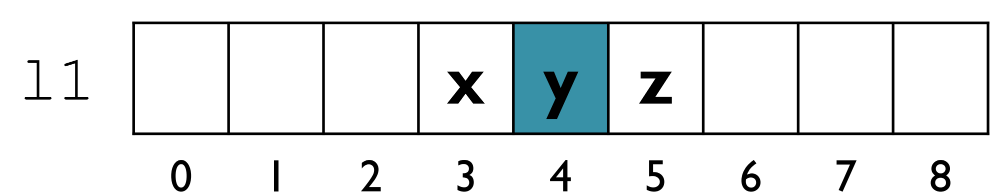
$n = \mathrm{n1} + \mathrm{n2}$

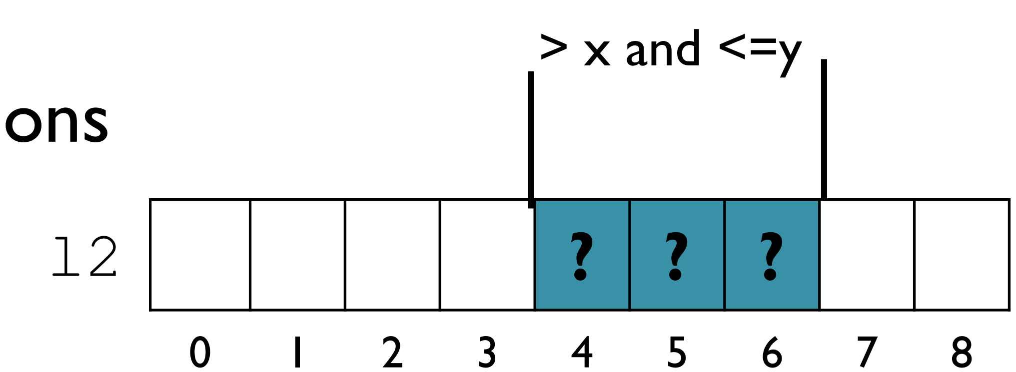
Worst case: `l2[n-1] < l1[0]`



n1*n2 comparisons

Idea again: reduce explicit comparisons

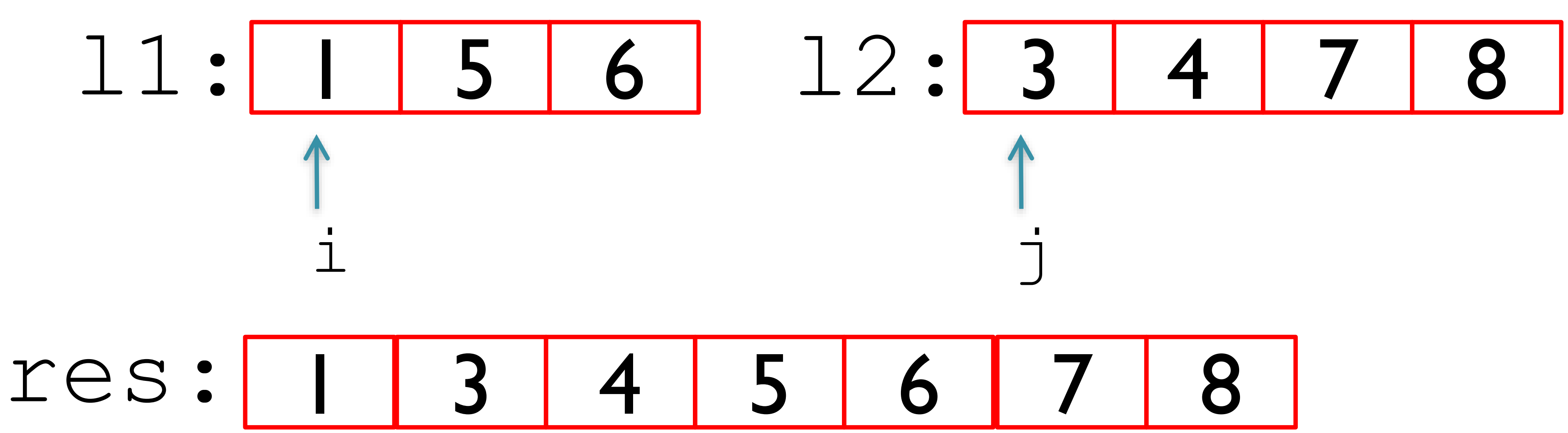




O(n1+n2) comparisons

Smart Merging Algorithm

```
def merge(l1, l2):
    res = []
    n1, n2 = len(l1), len(l2)
    i, j = 0, 0
    while i < n1 and j < n2:
        if l1[i] <= l2[j]:
            res += [l1[i]]
            i += 1
        else:
            res += [l2[j]]
            j += 1
    return res + l1[i:] + l2[j:]
```



Computational Complexity of Smart Merging Algorithm

```
def merge(l1, l2):
    res = []
    n1, n2 = len(l1), len(l2)
    i, j = 0, 0
    while i < n1 and j < n2:
        if l1[i] <= l2[j]:
            res += [l1[i]]
            i += 1
        else:
            res += [l2[j]]
            j += 1
    return res + l1[i:] + l2[j:]
```

`def merge(l1, l2):`	$n = \mathrm{n1} + \mathrm{n2}$	
`res = []`	$O(0)$	$O(0)$
`n1, n2 = len(l1), len(l2)`	$O(0)$	$O(0)$
`i, j = 0, 0`	$O(1)$	$O(1)$
`while i < n1 and j < n2:`	$O(n)$	$O(n)$
`if ... else ...` block	$O(n)$	$O(n)$
`return res + l1[i:] + l2[j:]`	$O(n)$	$O(n)$
	$O(n)$	$O(n)$

- Loop terminates after n1 increments of i or n2 increments of j
- In every iteration either i or j are incremented
- So loop terminates after at most n1+n2 = n iterations

- Final concatenation in O(n+n1+n2) = O(n)

Complexity of Mergesort

```
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
            b, c = a + k, a + 2*k
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```

`k, n = 1, len(ls)`	$O(0)$	$O(0)$
`while k < n:`	$O(\log n)$	$O(\log n)$
`nxt = []`	$O(\log n)$	$O(\log n)$
`for a in range(0, n, 2*k):`	$O(n)$	$O(n)$
`b, c = a + k, a + 2*k`	$O(n)$	$O(n)$
`nxt += merge(ls[a:b], ls[b:c])`	?	?
`ls = nxt`	$O(\log n)$	$O(\log n)$
`k = 2 * k`	$O(\log n)$	$O(\log n)$
`return ls`	$O(0)$	$O(0)$

- Let k_i be chunk size after i iterations of loop
- Merging cost in i-th iteration: $\frac{n}{2k_i} T_{\text{merge}}(2k_i)$

pairs of chunks to merge

cost of merging

Complexity of Mergesort

```python
def mergesort(ls):
    k, n = 1, len(ls)
    while k < n:
        nxt = []
        for a in range(0, n, 2*k):
            b, c = a + k, a + 2*k
            nxt += merge(ls[a:b], ls[b:c])
        ls = nxt
        k = 2 * k
    return ls
```

`k, n = 1, len(ls)`	$O(0)$	$O(0)$
`while k < n:`	$O(\log n)$	$O(\log n)$
`nxt = []`	$O(\log n)$	$O(\log n)$
`for a in range(0, n, 2*k):`	$O(n)$	$O(n)$
`b, c = a + k, a + 2*k`	$O(n)$	$O(n)$
`nxt += merge(ls[a:b], ls[b:c])`	$O(n \log n)$	$O(n \log n)$
`ls = nxt`	$O(\log n)$	$O(\log n)$
`k = 2 * k`	$O(\log n)$	$O(\log n)$
`return ls`	$O(0)$	$O(0)$
	$O(n \log n)$	$O(n \log n)$

- Let k_i be chunk size after i iterations of loop
- Merging cost in i-th iteration: $\frac{n}{2k_i} T_{\text{merge}}(2k_i) = O\left(n \frac{2k_i}{2k_i}\right) = O(n)$
- Resulting total merge cost: $O(n \log n)$

Overview

1. Divide-and-conquer paradigm
2. Mergesort
3. Linear-time Merge
4. *Quicksort*

Summary of Mergesort

With Mergesort, we split our list until it was single elements, then merged sorted lists

Splitting the list was simple; just split in half

Merging took work; needed to keep the lists sorted

Going the other way

Quicksort splits the lists intelligently

5	**2**	**14**	**8**	**3**	**1**	**4**	**6**	**11**	**9**	**7**
0	1	2	3	4	5	6	7	8	9	10

Choose an item to be the "pivot" (say, the first item)

Partition into two parts based on the pivot

Quicksort - Partitioning

5	**2**	**14**	**8**	**3**	**1**	**4**	**6**	**11**	**9**	**7**
0	1	2	3	4	5	6	7	8	9	10

<=p				**p**	**>p**					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

- Pivot is in final position
- Items on the left and right do not need to change sides
- Sort each side independently

Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

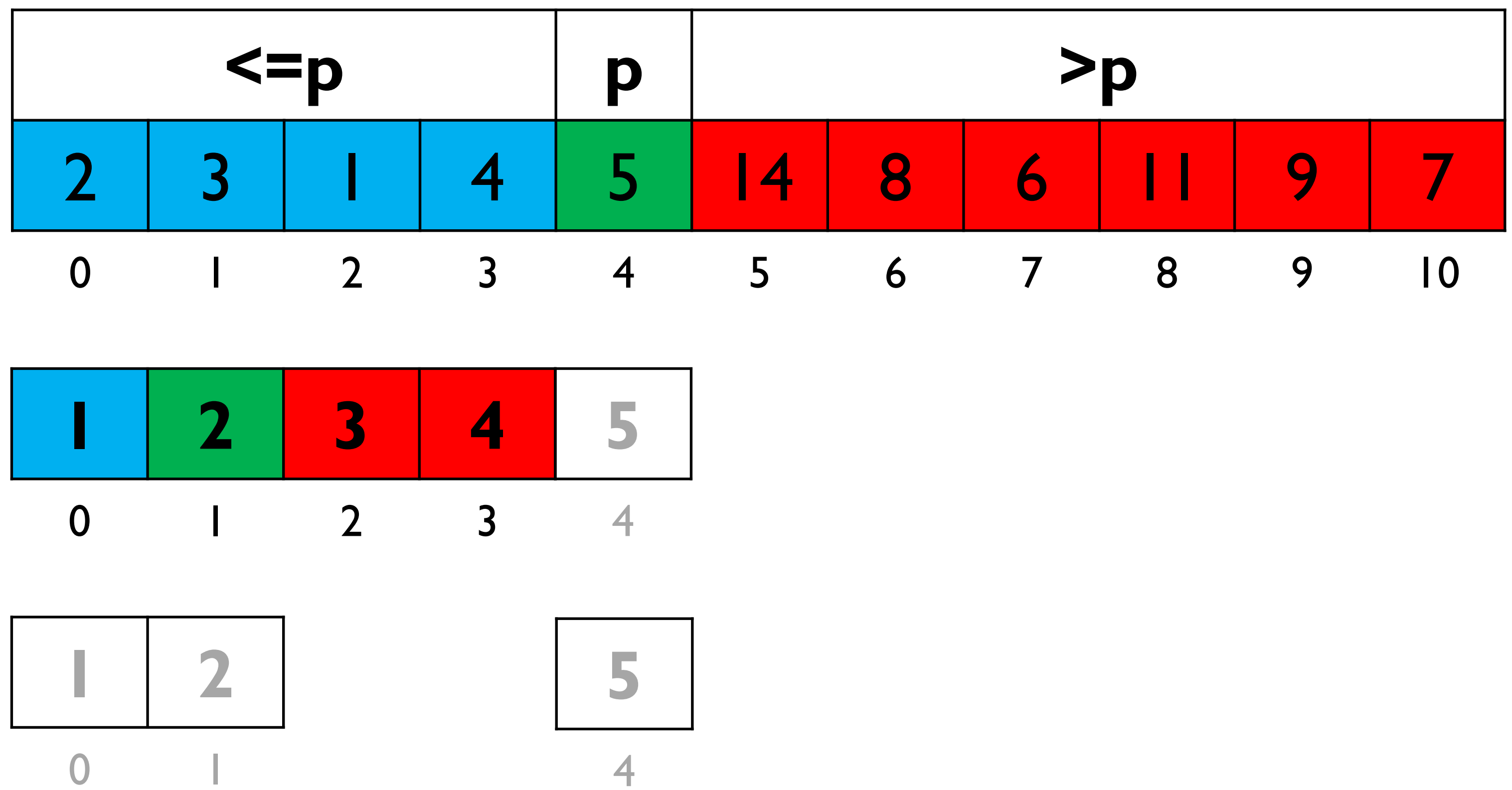
5
4

Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5
0	1	2	3	4

Quicksort - Algorithm



Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

1	**2**	**3**	**4**	**5**
0	1	2	3	4

1	**2**	**3**	**4**	**5**
0	1	2	3	4

Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5
0	1	2	3	4

1	2	3	4	5
0	1	2	3	4

1	2	3	4	5
0	1	2	3	4

Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	8	6	11	9	7	14
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5
0	1	2	3	4

1	2	3	4	5
0	1	2	3	4

Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	8	6	11	9	7	14
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5						14
0	1	2	3	4						10

1	2	3	4	5
0	1	2	3	4

Quicksort - Algorithm

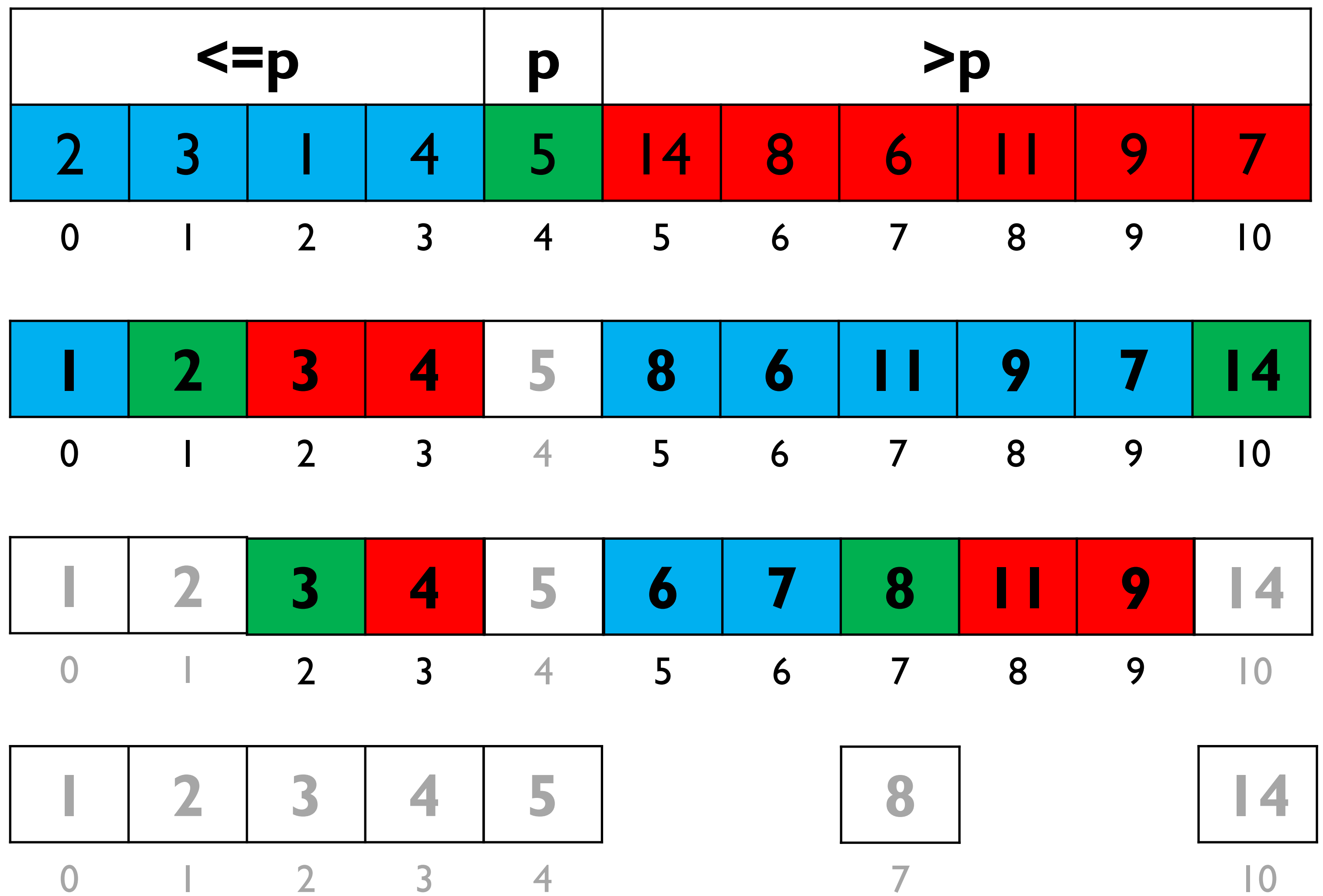
<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

1	**2**	**3**	**4**	5	**8**	**6**	**11**	**9**	**7**	**14**
0	1	2	3	4	5	6	7	8	9	10

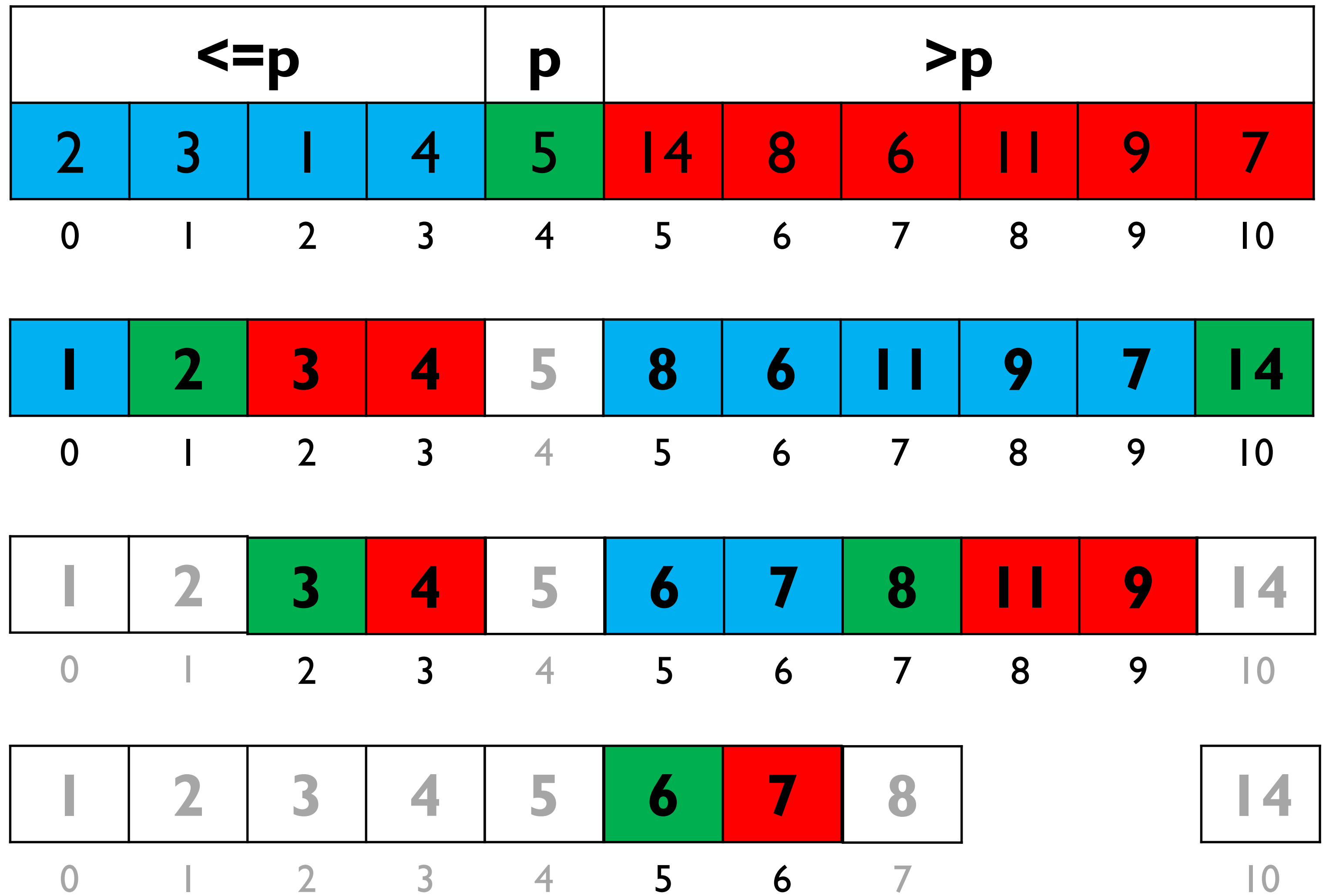
1	2	**3**	**4**	5	**6**	**7**	**8**	**11**	**9**	14
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5
0	1	2	3	4

Quicksort - Algorithm



Quicksort - Algorithm



Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	8	6	11	9	7	14
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8	11	9	14
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8			14
0	1	2	3	4	5	6	7			10

1	2	3	4	5	6	7	8			14
0	1	2	3	4	5	6	7			10

Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

0	1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	8	6	11	9	7	14

0	1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	11	9	14

0	1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	11	9	14

0	1	2	3	4	5	6	7			10
1	2	3	4	5	6	7	8			14

Quicksort - Algorithm

<=p				p	>p					
2	3	1	4	5	14	8	6	11	9	7
0	1	2	3	4	5	6	7	8	9	10

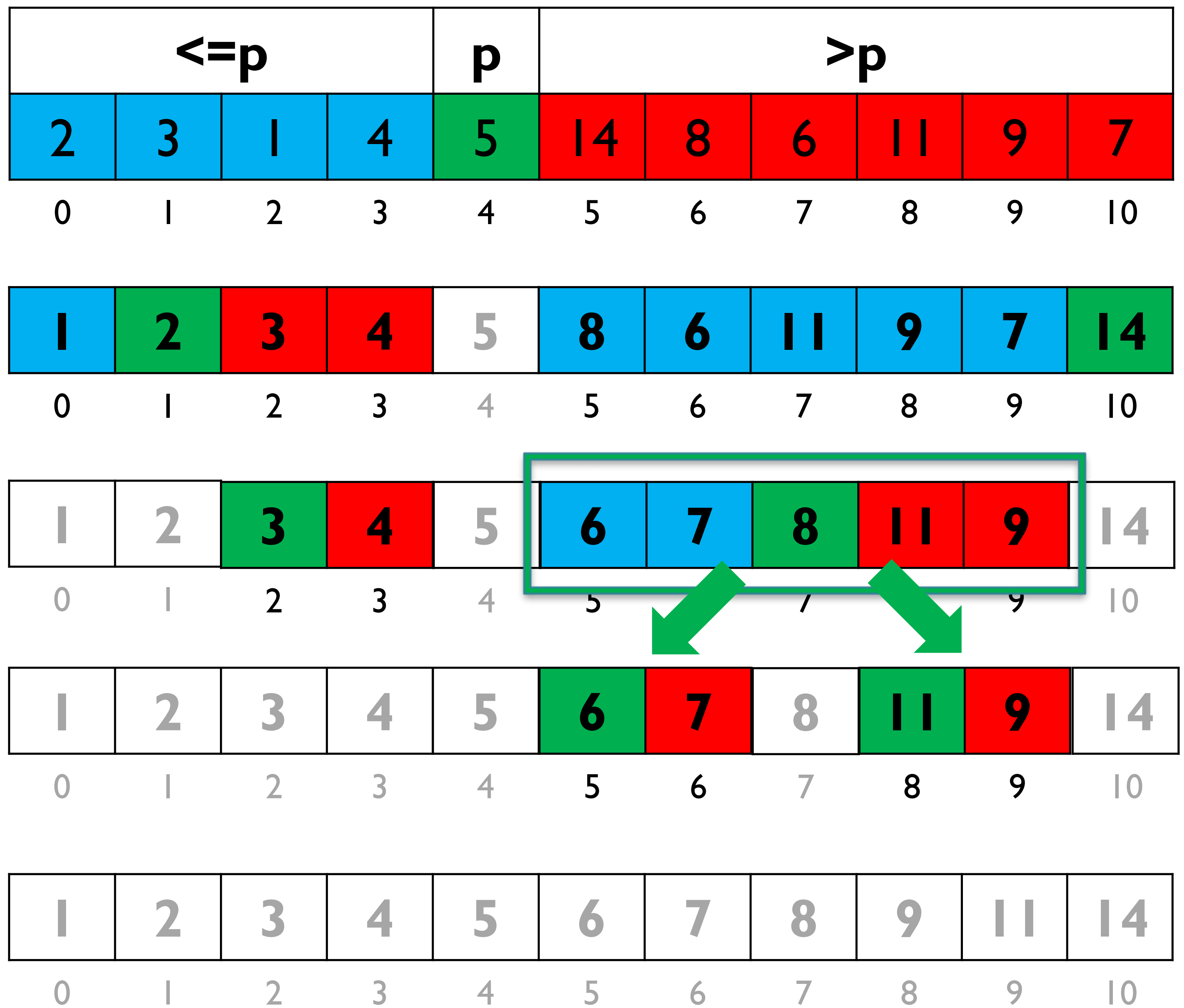
1	2	3	4	5	8	6	11	9	7	14
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8	11	9	14
0	1	2	3	4	5	6	7	8	9	10

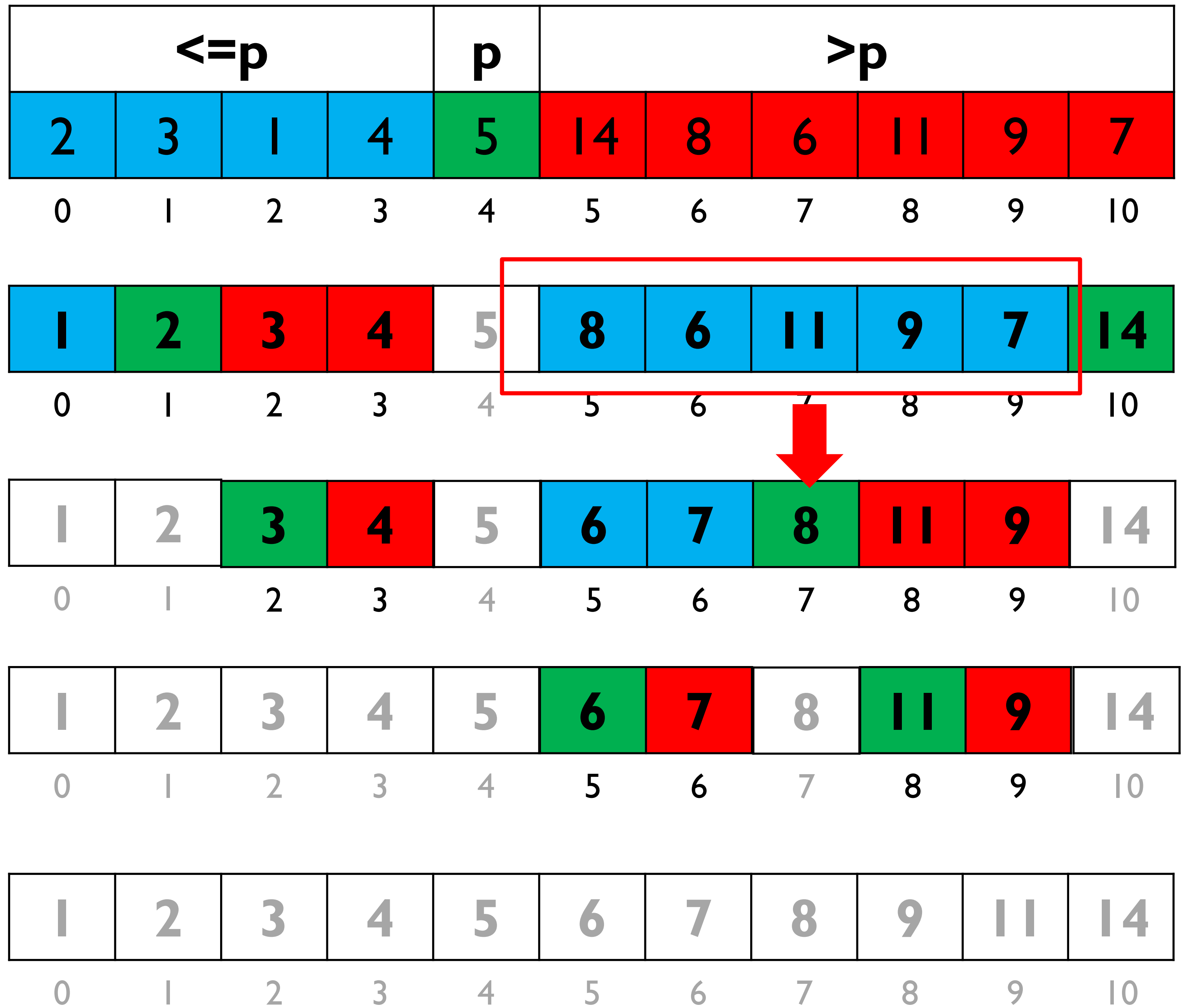
1	2	3	4	5	6	7	8	11	9	14
0	1	2	3	4	5	6	7	8	9	10

1	2	3	4	5	6	7	8	9	11	14
0	1	2	3	4	5	6	7	8	9	10

Good splits


<=p
p
>p
2 3 1 4 5 14 8 6 11 9 7
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 8 6 11 9 7 14
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 11 9 14
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 11 9 14
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 11 14
0 1 2 3 4 5 6 7 8 9 10

Bad splits



<=p
p
>p
2 3 1 4 5 14 8 6 11 9 7
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 8 6 11 9 7 14
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 11 9 14
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 11 9 14
0 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 11 14
0 1 2 3 4 5 6 7 8 9 10

Summary

Algorithmic paradigm: divide-and-conquer

- halving problem sizes leads to trivial subproblems after logarithmically many reductions
- if not too much overhead: allows to replace linear complexity term by logarithmic term

Merge Sort allows worst-case "linearithmic" sorting

Next lecture

- Recursive functions