Lecture 17: Analyzing Algorithms

CS 51P

November 2, 2022

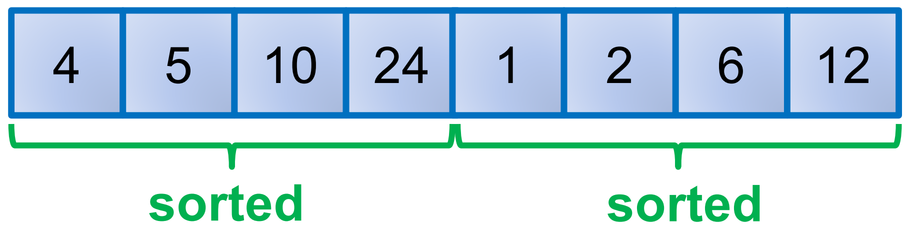
Tom Yeh
he/him/his

Three Possible Sorting Algorithms

- For each position in the list:
  - Find the object that should be there; put it in the right place
- For each object in the list:
  - If that object should be earlier in the list, put it in the right place
- Recursively:
  - Sort the first half of the list
  - Sort the second half of the list
  - Merge the two halves together

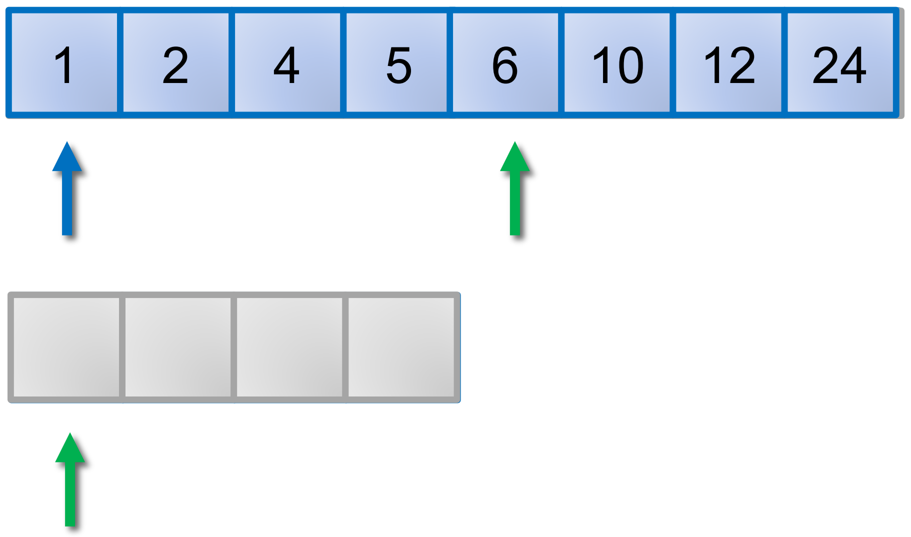
Merging

- What if our list looked like two sorted lists end to end?
- We could sort by merging the two lists!

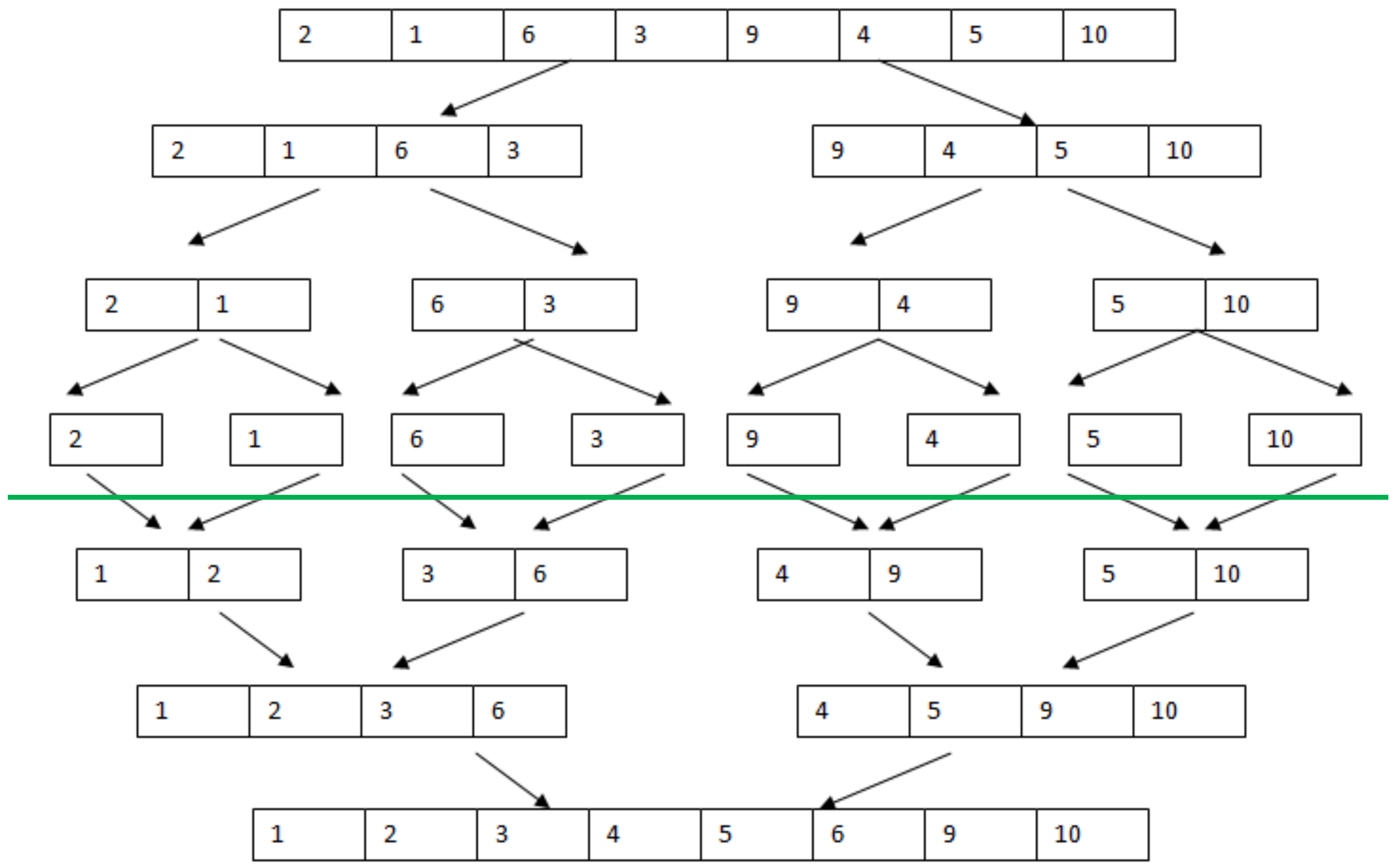


Merging

- What if our list looked like two sorted lists end to end?
- We could sort by merging the two lists!



Mergesort


2 1 6 3 9 4 5 10
2 1 6 3
9 4 5 10
2 1
6 3
9 4
5 10
2
1
6
3
9
4
5
10
1 2
3 6
4 9
5 10
1 2 3 6
4 5 9 10
1 2 3 4 5 6 9 10

Implement Mergesort

- Assume we have a function merge(list, start, end)

Sorting Algorithms

Selection Sort

```
def selection_sort(lst):

  # for each pos in list
  for pos in range(len(lst)):

    # find obj that should be there
    min_pos = pos
    for i in range(pos+1, len(lst)):
      if lst[i] < lst[min_pos]:
        min_pos = i

    # swap that obj into position pos
    swap(lst, pos, min_pos)
```

Insertion Sort

```
def insertion_sort(lst):

  # for each obj in list
  for pos in range(len(lst)):

    # move obj to correct position
    curr_pos = pos
    while curr_pos > 0 and
        lst[curr_pos]<lst[curr_pos-1]:
      swap(lst, curr_pos-1, curr_pos)
      curr_pos = curr_pos - 1
```

Which algorithm is better?

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*?

- Correct(er)?
- Faster?
- Less space?
- Less power consumption?
- Easier to code?
- Easier to maintain?
- Required for homework?

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

Example Basic steps:

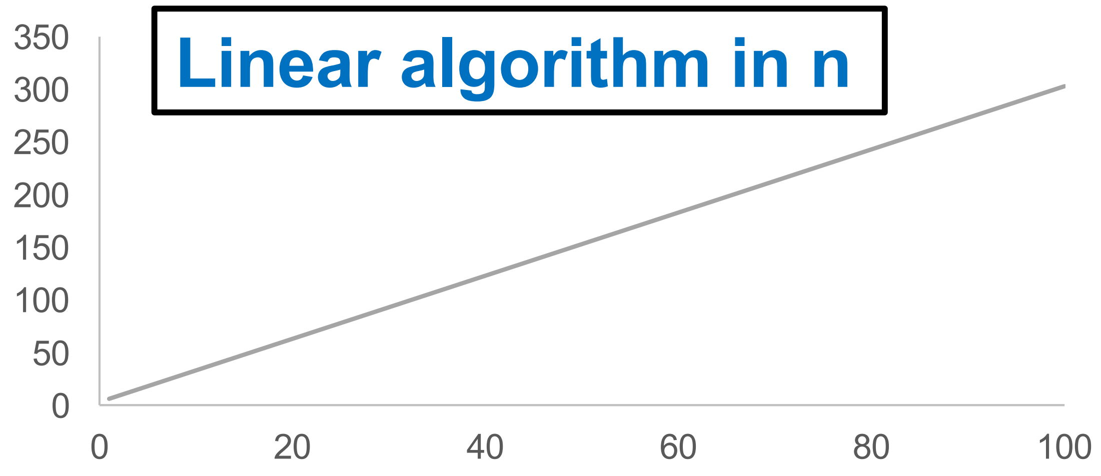
- Access value of a variable, list element, or object attr
- Assign to a variable, list element, or object attr
- Do one arithmetic or logical operation
- Call a function

Counting Steps

```
# Store sum of 0..n-1 in sum
sum = 0
for i in range(n):
    sum = sum + i
```

Statement:	# times done
sum = 0	1
i= v	n
sum = sum + i	n
Total steps:	2n + 1

All basic steps take time 1. There are n loop iterations. Therefore, takes time proportional to n.



Not all operations are basic steps

```
# Store n copies of ‘c’ in s
s = ""
for i in range(n):
    s = s + 'c'
```

Staten	mes done
s = ""	1
i = v	n
s = s + 'c'	n
Total ste	n + 1

Concatenation is not a basic step. Strings are immutable, but we can reassign. Each concatenate requires creating a string with more elements. Copying the values over, and then assigning the new elements to the new values.

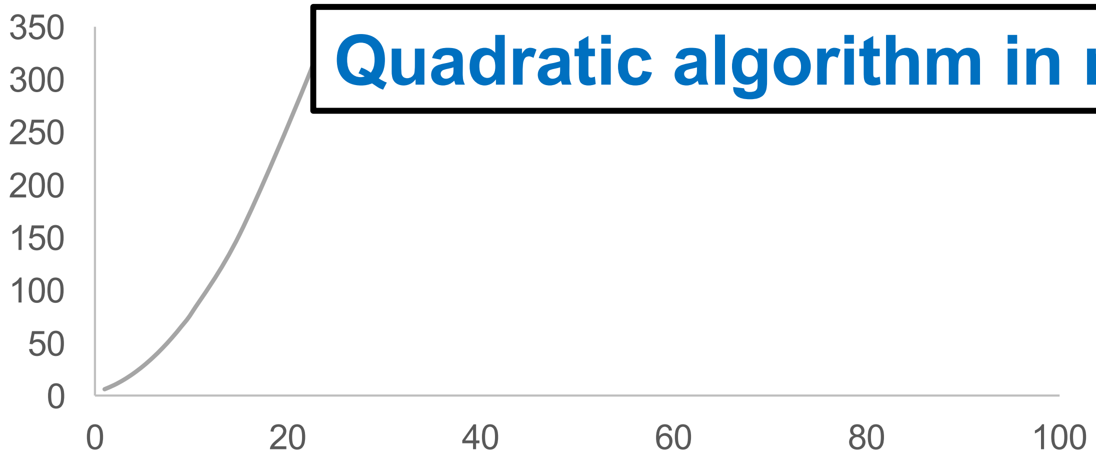
For each i, concatenation creates and fills i sequence

Not all operations are basic steps

```
# Store n copies of ‘c’ in s
s = ""
for i in range(n):
    s = s + 'c'
```

Statement:	# times	# steps
s = ""	1	1
i = v	n	1
s = s + 'c';	n	i
Total steps:	(n-1)*n/2 + n + 1	

Concatenation is not a basic step. For each i, concatenation creates and fills i sequence elements.



Quadratic algorithm in n

Linear versus quadractic

```
# Store sum of 1..n in sum
sum = 0
for i in range(1, n+1):
    sum = sum + k;
```

```
# Store n copies of ‘c’ in s
s = ""
for i in range(n):
    s = s + 'c'
```

Linear algorithm

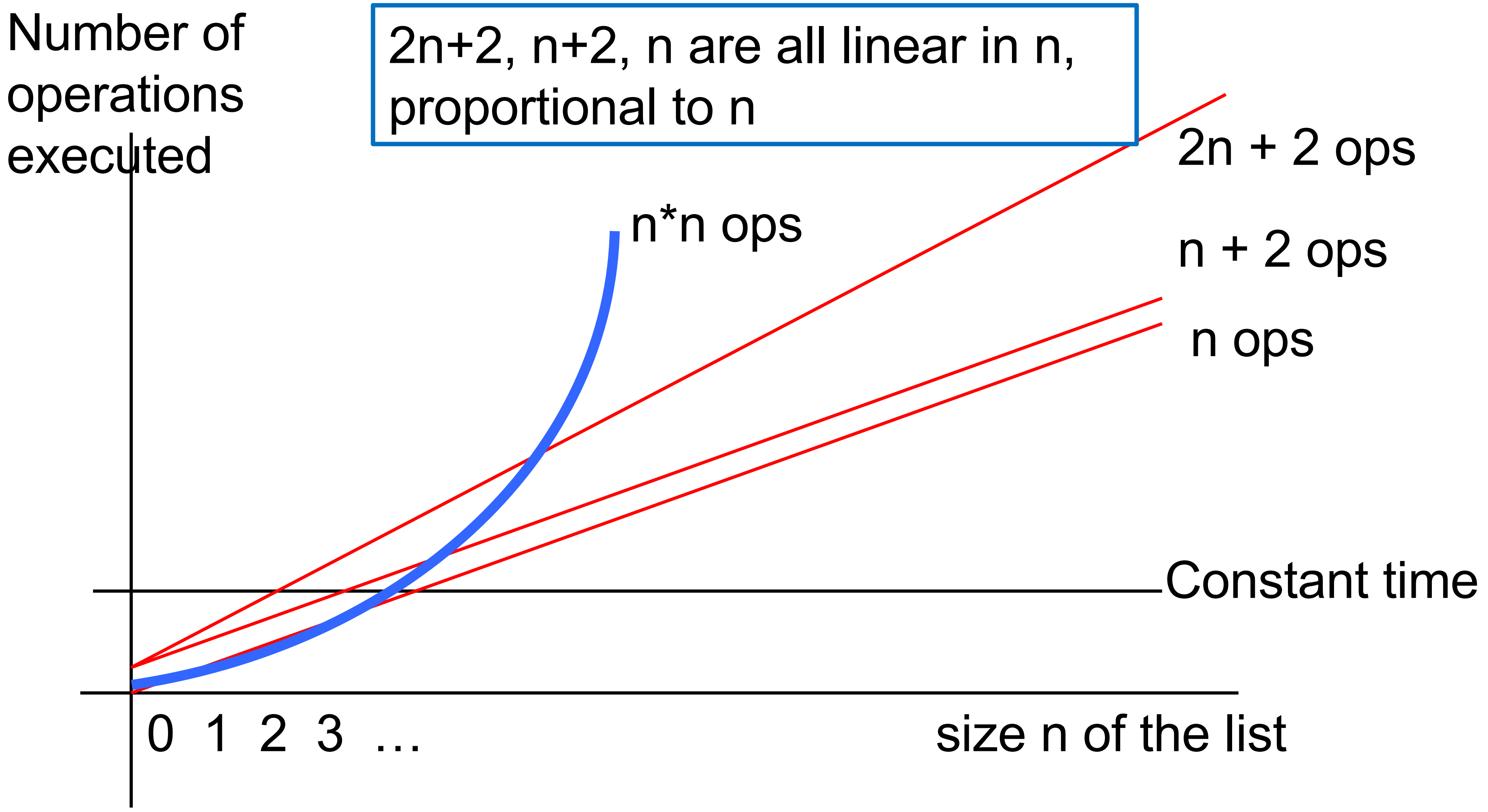
Quadratic algorithm

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

Looking at execution speed

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



"Big O" Notation

• $n^2 + 2n + 5$	$O(n^2)$
• $1000n + 25000$	$O(n)$
• $\frac{2^n}{15} + n^{100}$	$O(2^n)$
• $n \log n + 25n$	$O(n \log n)$

How Fast is Fast enough?

$O(1)$	constant	excellent
$O(\log n)$	logarithmic	excellent
$O(n)$	linear	good
$O(n \log n)$	n log n	pretty good
$O(n^2)$	quadratic	maybe OK
$O(n^3)$	cubic	not good
$O(2^n)$	exponential	too slow

Evaluating Speed of Selection Sort

```
def selection_sort(lst):
  for pos in range(len(lst)):
    # find obj that should be there
    min_pos = pos
    for i in range(pos+1, len(lst)):
      if lst[i] < lst[min_pos]:
        min_pos = i

    # swap that obj to position pos
    swap(lst, pos, min_pos)
```

# Times	# Steps
n	O(1)
n	O(1)
n*O(n)	O(1)
n*O(n)	O(1)
<= n*O(n)	O(1)
n	O(1)

Selection Sort runs in time $O(n^2)$

Comparison

	selection sort
worst case	O(n^2)
best case	O(n^2)
avg case	O(n^2)
space	O(1)

Evaluating Speed of Insertion Sort

```
def insertion_sort(lst):
  for pos in range(len(lst)):
    # swap that obj to right place
    curr_pos = pos
    while curr_pos > 0 and
        lst[curr_pos]<lst[curr_pos-1]:
      swap(lst, curr_pos-1, curr_pos)
      curr_pos = curr_pos - 1
```

# Times	# Steps
n	O(1)
n	O(1)
<=n*O(n)	O(1)
<= n*O(n)	O(1)
<= n*O(n)	O(1)

Insertion Sort runs in time $O(n^2)$

Comparison

	selection sort	insertion sort
worst case	$O(n^2)$	$O(n^2)$
best case	$O(n^2)$	$O(n)$
avg case	$O(n^2)$	$O(n^2)$
space	$O(1)$	$O(1)$

Evaluating Speed of Merge Sort

```
def merge_sort_helper(lst, start, end):
  # Base Case
  if (end-start) < 2:
    return

  # Recursive Case
  middle = start + int((end-start)/2)
  merge_sort_helper(lst, start, middle)
  merge_sort_helper(lst, middle, end)
  merge(lst, start, end)

def merge_sort(lst):
  merge_sort_helper(lst, 0, len(lst))
```

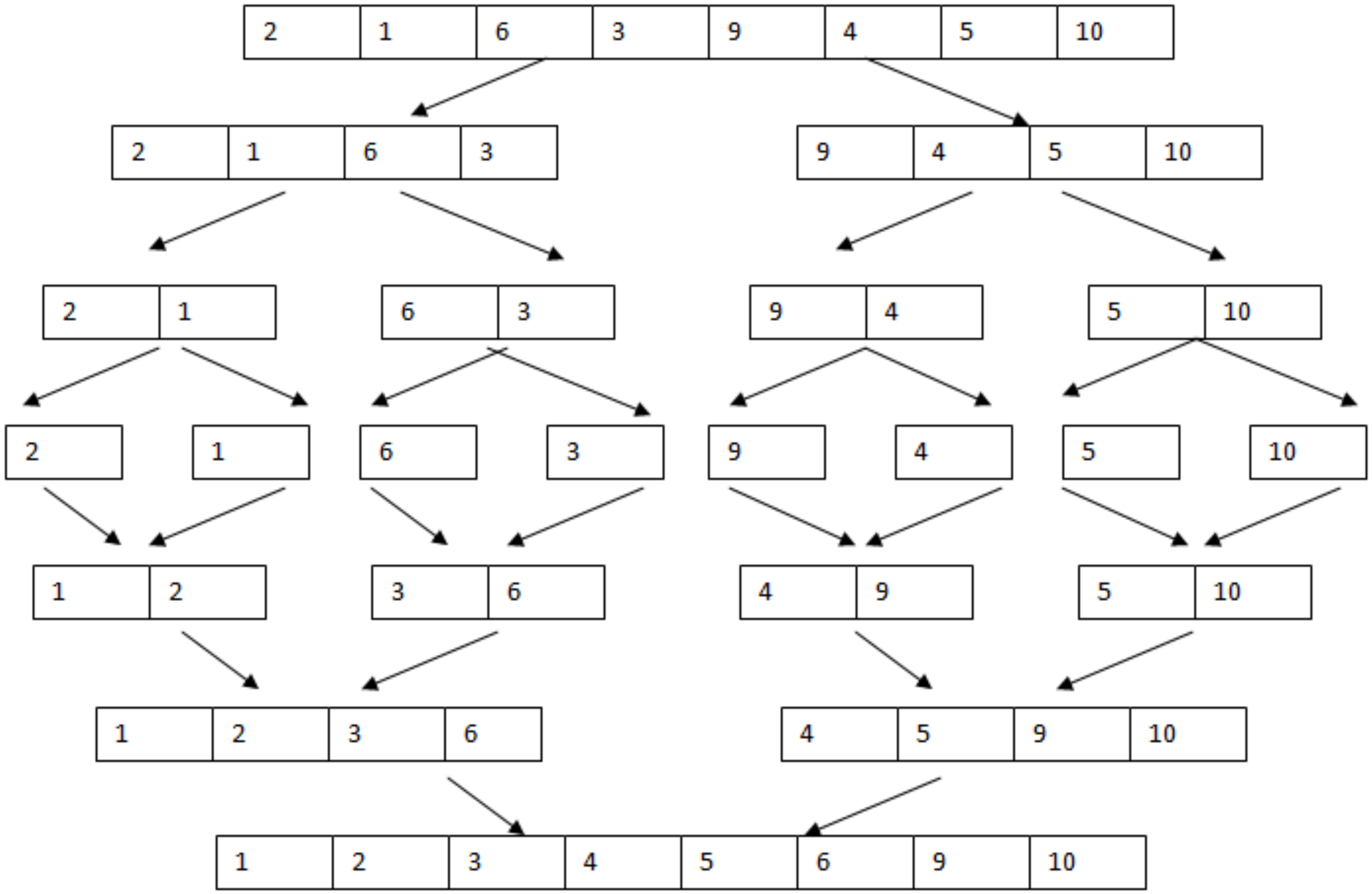
# Times	# Steps
1	O(1)
<=1	O(1)
1	O(1)
	?
	?
	?

Evaluating Speed of Merge Sort

```
def merge(lst, start, end):
  middle = (end-start)//2
  olist = lst[start:middle].copy()
  pos = start
  i = start
  j = middle
  length = len(lst)
  while i < middle :
    if j == length or olist[i] < lst[j
      lst[pos] = olist[i]
      i += 1
    else:
      lst[pos] = lst[j]
      j += 1
    pos += 1
```

# Times	# Steps
1	2
1	O(end-start)
1	1
1	1
1	1
1	O(1)
(end-start)/2	O(1)
(end-start)/2	O(1)
<=(end-start)/2	O(1)
<=(end-start)/2	O(1)
<=(end-start)/2	O(1)
<=(end-start)/2	O(1)
(end-start)/2	O(1)

Evaluating Speed of Merge Sort



Comparison

	selection sort	insertion sort	merge sort
worst case	$O(n^2)$	$O(n^2)$	$O(n \log n)$
best case	$O(n^2)$	$O(n)$	$O(n \log n)$
avg case	$O(n^2)$	$O(n^2)$	$O(n \log n)$
space	$O(1)$	$O(1)$	$O(n)$

Sorting in Python

- List.sort()
  - Sorts list in place
  - Optional argument reverse=True to reverse order (greatest->least)
  - Optional argument key defines expression to sort

- sorted(lst)
  - Creates sorted copy of list
  - Optional arguments reverse and key