4.2

Sorting

Lesson Plan

- [10] Icebreaker
- [15] O(n^2) sorts
- [25] O(nlogn) sorts
- [5] lst.sort()
- [10] O(n) sorts???

- [30] Practice!

Icebreaker

n^2 sorts

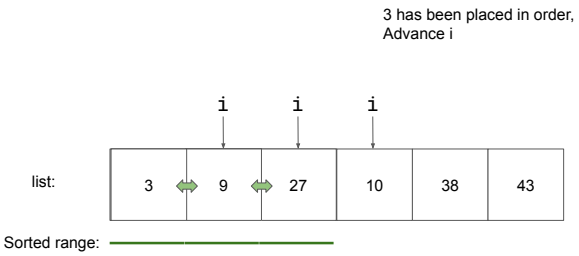
visualizer

Elements of n^2 sorts

- Outer and inner loops
- Sorted and unsorted portions of the list

Insertion Sort

Insertion Sort



Insertion Sort

One iteration of insert (for the number at index i):
While the value to its left is larger:
    Swap to the left

Full insertion sort:
For i from 1 to len(lst):
    insert starting at i

We pick values from the unsorted area (back part) and ***insert*** them into the correct spot in the sorted area (front part)

Insertion Sort

```
def insertion_sort(list):
  for i in range(1, len(arr)):
    element = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > element:
      arr[j+1] = arr[j]
      j -= 1
    arr[j+1] = element
```

Insertion Sort - Performance

Size of input is len(list), which we'll call n

```
def insertion_sort(list):
  for i in range(1, len(arr)):                  → n steps
    element = arr[i]                            → 2 elemental operations
    j = i - 1
    while j >= 0 and arr[j] > element:          → worst-case: n steps
      arr[j+1] = arr[j]                         → 2 elemental operations
      j -= 1
    arr[j+1] = element                          → 1 elemental operation
```

Total Operations (worst case): n * (2 + n * (2) + 1) = $2n^2 + 3n$ = **$O(n^2)$**

Insertion Sort - Performance

Worst-case input:
1. Every element needs to be moved all the way to the front by the while loop
2. Equivalently: a reverse-sorted list

$O(n^2)$

Best-case input:
1. Every element doesn't need to swapped at all by the while loop
2. Equivalently: a sorted list

O(n)

Bubble Sort

Bubble Sort

One iteration of bubble (starting at index i):

For j from 0 to len(lst) - i - 1:

Compare the values at j and j+1

Swap them (if needed) so the larger value is at j+1

Full bubble sort:

For i from 0 to len(lst) - 1:

bubble starting at i

We ***bubble*** larger numbers up starting at each index

```
def bubble_sort(lst):
  for i in range(len(lst) - 1):

    for j in range(len(lst) - i - 1):
      if lst[j] > lst[j + 1]:

        lst[j], lst[j + 1] = lst[j + 1], lst[j]


```

```
def bubble_sort(lst):
  for i in range(len(lst) - 1):
    swapped = False
    for j in range(len(lst) - i - 1):
      if lst[j] > lst[j + 1]:
        swapped = True
        lst[j], lst[j + 1] = lst[j + 1], lst[j]

    if not swapped:
      return
```

Bubble Sort - Performance

Worst-case input:

1. Every element needs bubble all the way to the end
2. A reverse-sorted list

$O(n^2)$

Best-case input:

1. swapped = False so the loop exits after one iteration
2. A sorted list

O(n)

Selection Sort

Selection Sort

One iteration of select starting at i:

Find the minimum element from i to the end of the lst

Swap the minimum element with the element at i

Full Selection Sort:

for i from 0 to len(lst) - 1:

select(i)

(this is a typo)

```
1  def selection_sort(lst):
2    for i in range(len(lst)) - 1:
3      min_index = i
4      for j in range(i + 1, len(lst)):
5        if lst[j] < lst[min_index]:
6          min_index = j
7      lst[i], lst[min_index] = lst[min_index], lst[i]
8
9
```

Selection Sort - Performance

Worst-case input:
1. Every element needs to be moved all the way to the front by the while loop
2. Equivalently: a reverse-sorted list

$O(n^2)$

Best-case input:
1. No matter what, we need to find the minimum (an O(n) operation) n-1 times

$O(n^2)$

nlogn sorts

Merge Sort

First off, let's talk merge

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

lst3 = lst1 + lst2
lst3.sort()

This is nlogn!

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = []

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = [6]

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = [6, 12]

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = [6, 12, 15]

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = [6, 12, 15, 18]

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = [6, 12, 15, 18, 19]

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = [6, 12, 15, 18, 19, 25]

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = [6, 12, 15, 18, 19, 25, 26]

Can you combine 2 sorted arrays into 1 sorted array?

lst1 = [12, 18, 19, 30]
lst2 = [6, 15, 25, 26]

res = [6, 12, 15, 18, 19, 25, 26, 30]

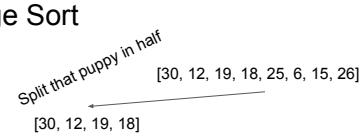
This is linear! O(n+m)

```python
def merge(lst1, lst2):
  i = 0
  j = 0

  res = []

  while i < len(lst1) or j < len(lst2):
    [redacted]
    elif lst1[i] < lst2[j]:
      res.append(lst1[i])
      i += 1
    else:
      res.append(lst2[j])
      j+= 1

  return res
```

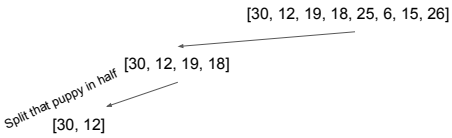
Merge Sort

[30, 12, 19, 18, 25, 6, 15, 26]

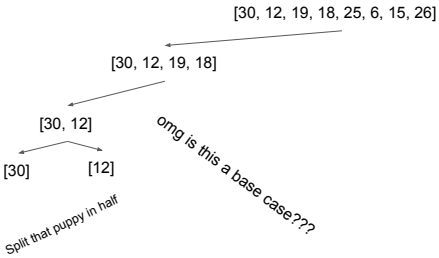
Merge Sort



Merge Sort

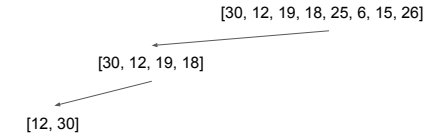


Merge Sort



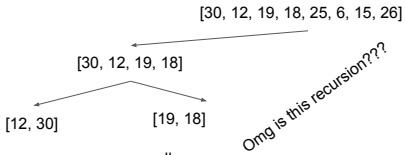
it's mergin' time

Merge Sort



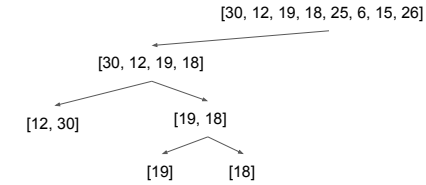
CALL ME
A
HIGHWAY
ON-RAMP
BECAUSE
I'M
MERGED

Merge Sort



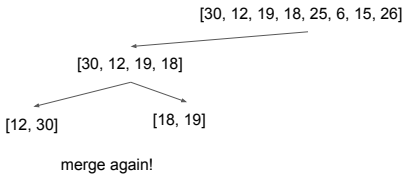
call me a
banana
because
i'm split

Merge Sort



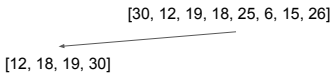
merge!

Merge Sort



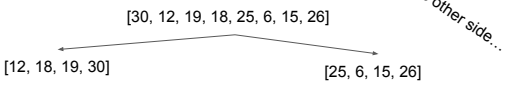
merge again!

Merge Sort

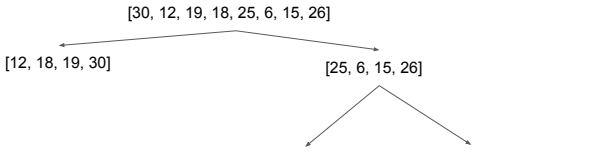


Merge Sort

Now for the other side…



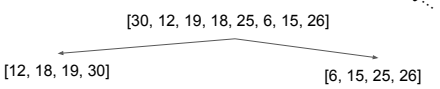
Merge Sort



A whole buncha splittin' & mergin' you get the idea

Merge Sort

Eventually…



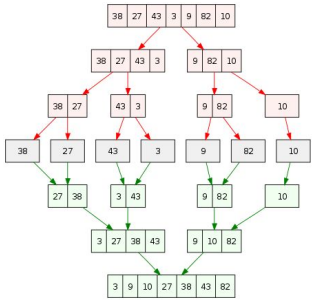
Merge Sort

[6, 12, 15, 18, 19, 25, 26, 30]

voila

Mergesort

- When n == 1, return immediately
- Split array in half and recursively sort each half
- Merge the sorted halves into a final sorted sequence

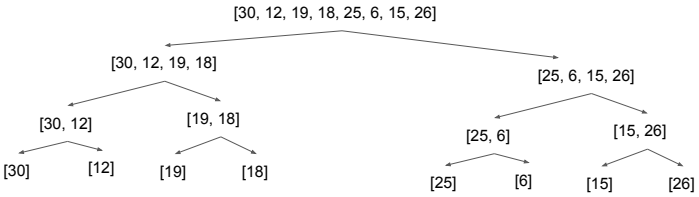


Merge Sort

```
def merge_sort(lst):
  if len(lst) <= 1:
    return lst

  mid = len(lst) // 2

  left = merge_sort(lst[:mid])
  right = merge_sort(lst[mid:])
  return merge(left, right)
```

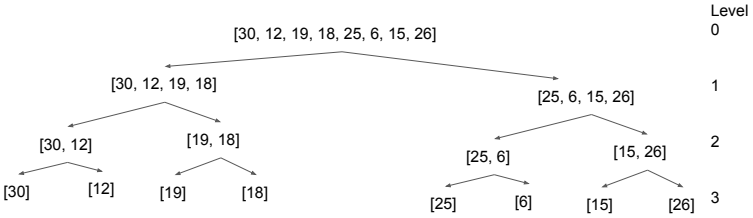
```
1  def merge(lst1, lst2):
2    i = 0
3    j = 0
4
5    res = []
6
7    while i < len(lst1) or j < len(lst2):
8      if i >= len(lst1):
9        res += lst2[j:]
10       break
11     elif j >= len(lst2):
12       res += lst1[i:]
13       break
14     elif lst1[i] < lst2[j]:
15       res.append(lst1[i])
16       i += 1
17     else:
18       res.append(lst2[j])
19       j+= 1
20
21   return res
```

Merge Sort Runtime



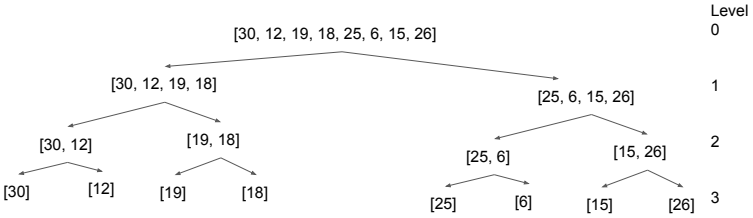
How many nodes are there?
How much work is done at each node?

Merge Sort Runtime



There are log(n) levels
Each level i has 2^i nodes
Each node at level i has $n/2^i$ elements

Merge Sort Runtime



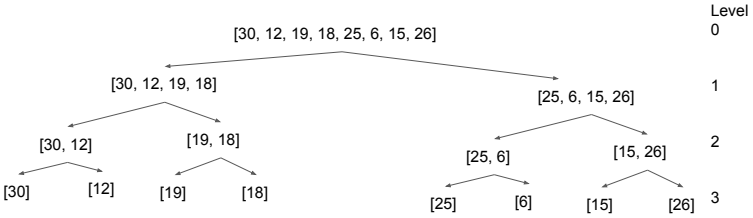
There are log(n) levels
Each level i has 2^i nodes
Each node at level i has $n/2^i$ elements

So there are just n elements at every level…

So linear merge makes every level take O(n) work!

O(n) work on log(n) levels = **O(nlogn)**

Merge Sort Space Complexity



There are log(n) levels
Each level i has 2^i nodes
Each node at level i has $n/2^i$ elements

At most one branch is on the stack at a time.
n + n/2 + n/4 + … + 1 < 2n = O(n)

Quick Sort

First, let's talk partition

[30, 12, 19, 26, 25, 6, 15, 18]

Pick a pivot (we commonly use the last element of the list)

Rearrange the list so that all of the elements smaller than the pivot come before it, all the of the larger elements come after

Partition

[30, 12, 19, 26, 25, 6, 15, 18]

[12, 6, 15, 18, 30, 19, 26, 25]

```
lst = [30, 12, 19, 26, 25, 6, 15, 18]
pivot = lst[-1]
small_lst = []
big_lst = []

for i in range(len(lst) - 1):
  if lst[i] <= pivot:
    small_lst.append(lst[i])
  else:
    big_lst.append(lst[i])

result = []
result.extend(small_lst)
result.append(pivot)
result.extend(big_lst)
```

Partition

This method uses O(n) extra space

It's possible (and more common) to do it in O(1) space, but this is a fine implementation

The idea behind Quick Sort…

[30, 12, 19, 26, 25, 6, 15, 18]

Pick a pivot

[12, 6, 15, 18, 25, 30, 19, 26]

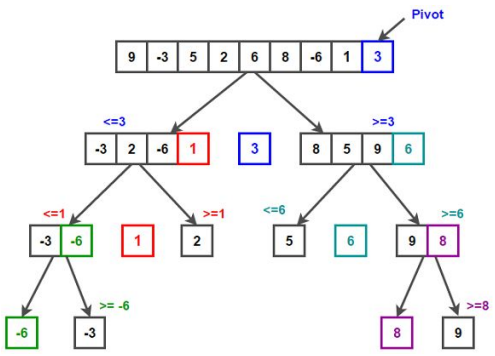
Partition

Note: The pivot's in its correct place now! And everything else is on its correct side!

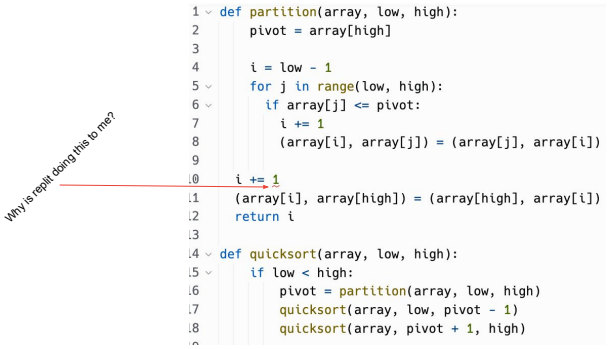
Repeat with the left side and repeat with the right side…

Recursion!

Recursion!



Quick Sort Implementation



Quicksort Runtime

Partitioning is O(n)

If we partition correctly, we split it perfectly in half each time, meaning we have log(n) levels

Just like in Merge Sort, this means we have *expected* O(nlogn) runtime

However, in the worst case (when the list is already sorted…), Quicksort has $O(n^2)$ runtime!

Runtime Comparisons

Effect of Doubling Input Size

n	nlog(n)	n^2
2	2	4
4	8	16
8	24	64
16	64	256
32	160	1024

Effect of 10x'ing Input Size

n	nlog(n)	n^2
10	~30	100
100	~700	10,000
1000	~10,000	1,000,000
10000	~130,000	100,000,000
100000	~1,700,000	10,000,000,000

lst.sort()

lst.sort()

sorts lst in place in O(nlogn) time!

```
lst = [42, 10, 36, 5]
lst.sort()
# now lst is equal to [5, 10, 36, 42]

x = lst.sort()
# x is equal to None!!
```

lst.sort() - reverse!

```
lst = [42, 10, 36, 5]
lst.sort(reverse = True)
# now lst is equal to [42, 36, 10, 5]
```

lst.sort() - key!

```
def mod_10(x):
    return x % 10

lst = [42, 10, 36, 5]
lst.sort(key = mod_10)
# now lst is equal to [10, 42, 5, 36]
```

Alternatively:

```
lst.sort(key = lambda x : x % 10)
```

lst.sort() - key!

You can even use built-in functions!

```
lst = [‘a’, ‘quick’, ‘purple’, ‘fox’]
lst.sort(key = len)

# lst is now ['a', 'fox', 'quick', 'purple']
```

lst.sort()

```
lst = [42, 10, 36, 5]
lst.sort(key = lambda x : x * -1, reverse = True)
```

What is lst now?

[5, 10, 36, 42]

Is an O(n) sort possible?

sort of.

(no pun intended)

Claim: No ***comparison sort*** can be better than O(nlogn)

Okay, so we wouldn't be allowed to *compare* values to each other.

If you know the range of possible values

The other sorts we learned are called 'comparison sorts', and the best we can do is nlogn.

If we don't have to compare values, we can do better.

Counting Sort

Here is a list of hypothetical student grades on a problem graded out of 20:
[7, 12, 3, 5, 3, 12, 19, 16, 20, 18, 20]

I know some facts about this data:
Minimum: 0
Maximum: 20

Counting Sort

[7, 12, 3, 5, 3, 12, 19, 16, 20, 18, 20]

Create a list of length 21 - one 'bucket' for each possible number, initialized to all 0s:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

You can do this in O(k) time where k is the maximum possible value.

Counting Sort

[7, 12, 3, 5, 3, 12, 19, 16, 20, 18, 20]

Now iterate through the list. Whenever you see a value x, increment the counter at index x of your buckets list. We're using this like a dictionary to count frequencies!

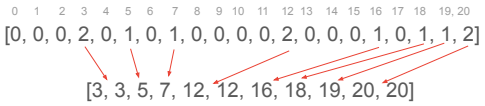
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19, 20
[0, 0, 0, 2, 0, 1, 0, 1, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 1, 1, 2]

You can do this in O(n) time where n is the number of elements.

Counting Sort

[7, 12, 3, 5, 3, 12, 19, 16, 20, 18, 20]

Finally, create an empty result list. Iterate through your buckets – for each bucket i with value x, add x i's to your result list.



You can do this in O(k + n) time where n is the number of elements.

Counting Sort

Creating empty buckets list: O(k)
Counting frequencies: O(n)
Creating result list: O(n + k)

Total runtime: O(n + k)

k is some constant (and hopefully small) number, so we can reduce this to O(n)!

Counting Sort Implementation

```
def counting_sort(nums, max_val):
  buckets = [0 for _ in range(max_val + 1)]

  for num in nums:
    buckets[num] += 1

  result = []
  for i in range(len(buckets)):
    i_count = buckets[i]
    for count in range(i_count):
      result += [i]

  return result


lst = [7, 12, 3, 5, 3, 12, 19, 16, 20, 18, 20]
max_val = 20
print(counting_sort(lst, max_val))
```

Counting Sort With Non-Integers

As long as items can be placed into integer buckets, Counting Sort works!

Instead of initializing buckets list with 0s, initialize it with all empty lists.

Instead of incrementing a count in each bucket, just add each item to its bucket's list

```
def counting_sort_players(players, max_val):
  buckets = [[] for _ in range(max_val + 1)]

  for player in players:
    player_num = player[1]
    buckets[player_num].append(player)

  result = []
  for players_per_num in buckets:
    for player_per_num in players_per_num:
      result.append(player_per_num)

  return result
```

```
def counting_sort(nums, max_val):
  buckets = [0 for _ in range(max_val + 1)]

  for num in nums:
    buckets[num] += 1

  result = []
  for i in range(len(buckets)):
    i_count = buckets[i]
    for count in range(i_count):
      result += [i]

  return result


lst = [7, 12, 3, 5, 3, 12, 19, 16, 20, 18, 20]
max_val = 20
print(counting_sort(lst, max_val))
```

```
players = [('Michael Jordan', 23), ('Sue Bird', 10), ('LeBron James', 6), ('Kareem Abdul-Jabar', 33), ('Maya Moore', 23)]
max_val = 55
print(counting_sort_players(players, max_val))
```

Practice Problems

You will be working in teams of 3 or 4. Use the table below to figure out what your role is. [replit]

Role	Responsibilities	Assignment Criteria
Captain	Share screen, write code, keep track of time, ensure all team members participate	Person who has been assigned this role the least number of times
Tester	Plays devil's advocate, design test cases, determine algorithm complexity (time and space)	Person who has been assigned this role the least number of times
Presenter	Explain solution to the class, present the team's algorithm design decisions, state solution's complexity (time and space), share one thing the team learned from the problem	Person who has been assigned this role the least number of times

If there are ties, get creative and come up with a way to break them (i.e., sort yourselves by last name, distance to Google Austin, etc.)
If there are 4 members in your team, you should have two Testers