Lecture 18 - March 20

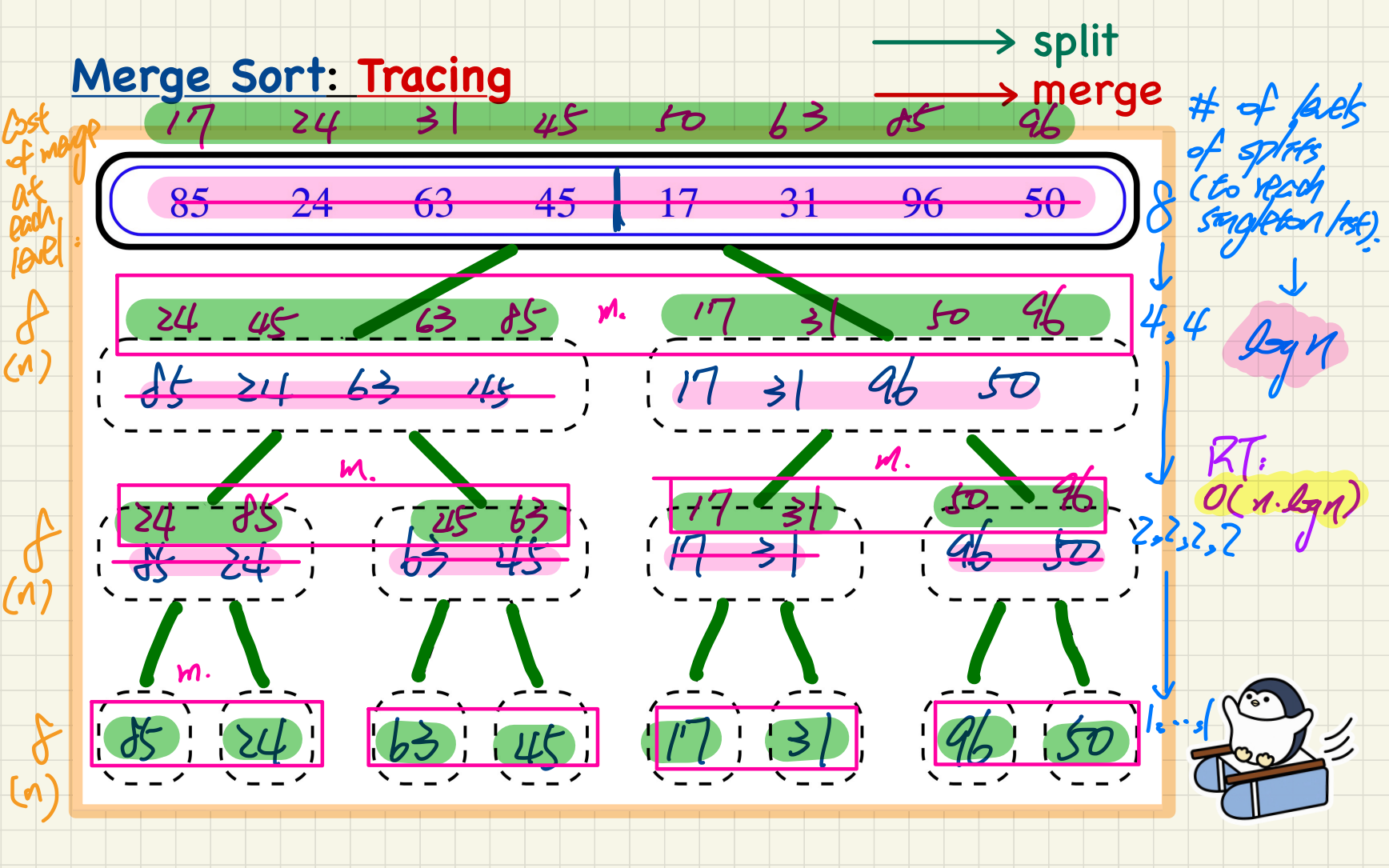
Merge Sort, Quick Sort, BST

MergeSort: Recurrence Relation
QuickSort: Ideas, Java, RT

Announcements/Reminders

- ProgTest2 info & example questions released
- Assignment 3 (on linked Trees) solution released
- WrittenTest and ProgTest1 results & feedback released
- Makeup Lecture (on Queues) posted
- Lecture notes template, Office Hours, TA Contact

Merge Sort: Tracing



Merge Sort: Running Time

```java
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>();
    sortedList.add(list.get(0));
  }
  else {
    int middle = list.size() / 2;
    List<Integer> left = list.subList(0, middle);
    List<Integer> right = list.subList(middle, list.size());
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = merge(sortedLeft, sortedRight);
  }
  return sortedList;
}
```

$T(0) = 1$

$T(1) = 1$

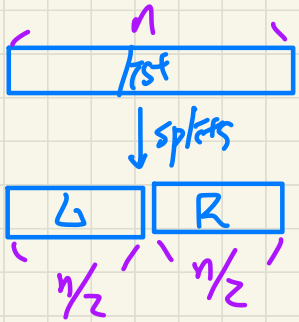
$O(1)$

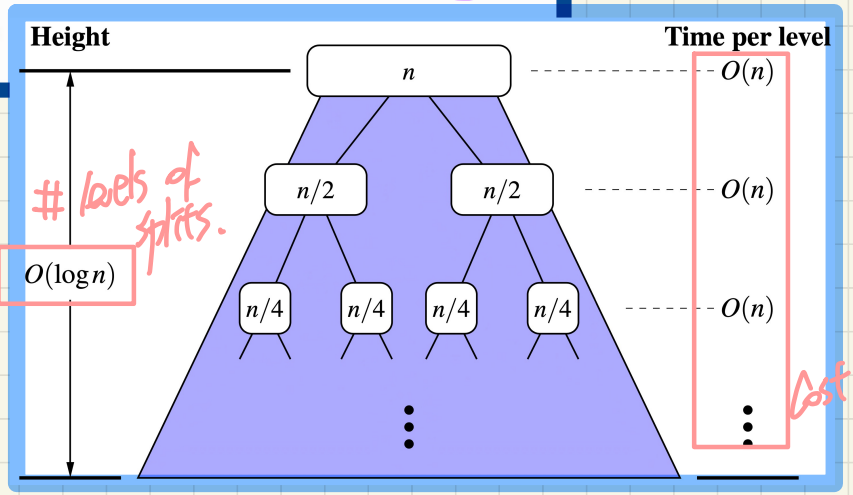
$T(n)$

assump: $T(\frac{n}{2})$

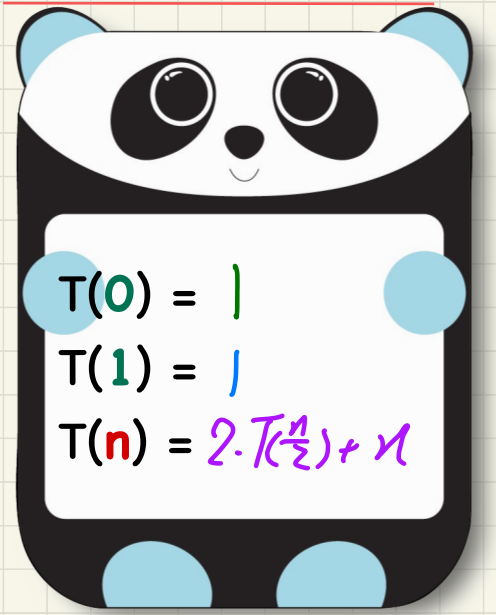
assump: $T(\frac{n}{2})$

$O(n)$





Running Time as a Recurrence Relation



Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

$T(n) = 2 \cdot T(n/2) + n$

Warm-up: $T(\frac{n}{2}) = 2 \cdot T(\frac{n/2}{2}) + \frac{n}{2}$

$T(n) = 2 \cdot T(\frac{n}{2}) + n$

$= 2 \cdot (2 \cdot T(\frac{n}{4}) + \frac{n}{2}) + n \quad [4 \cdot T(\frac{n}{4}) + 2n]$

$= 2 \cdot (2 \cdot (2 \cdot T(\frac{n}{8}) + \frac{n}{4}) + \frac{n}{2}) + n \quad [8 \cdot T(\frac{n}{8}) + 3n]$

$\vdots$

$= 2^{?} \cdot T(1) + ?n = 2^{\log n} \cdot 1 + \log n \cdot n =$

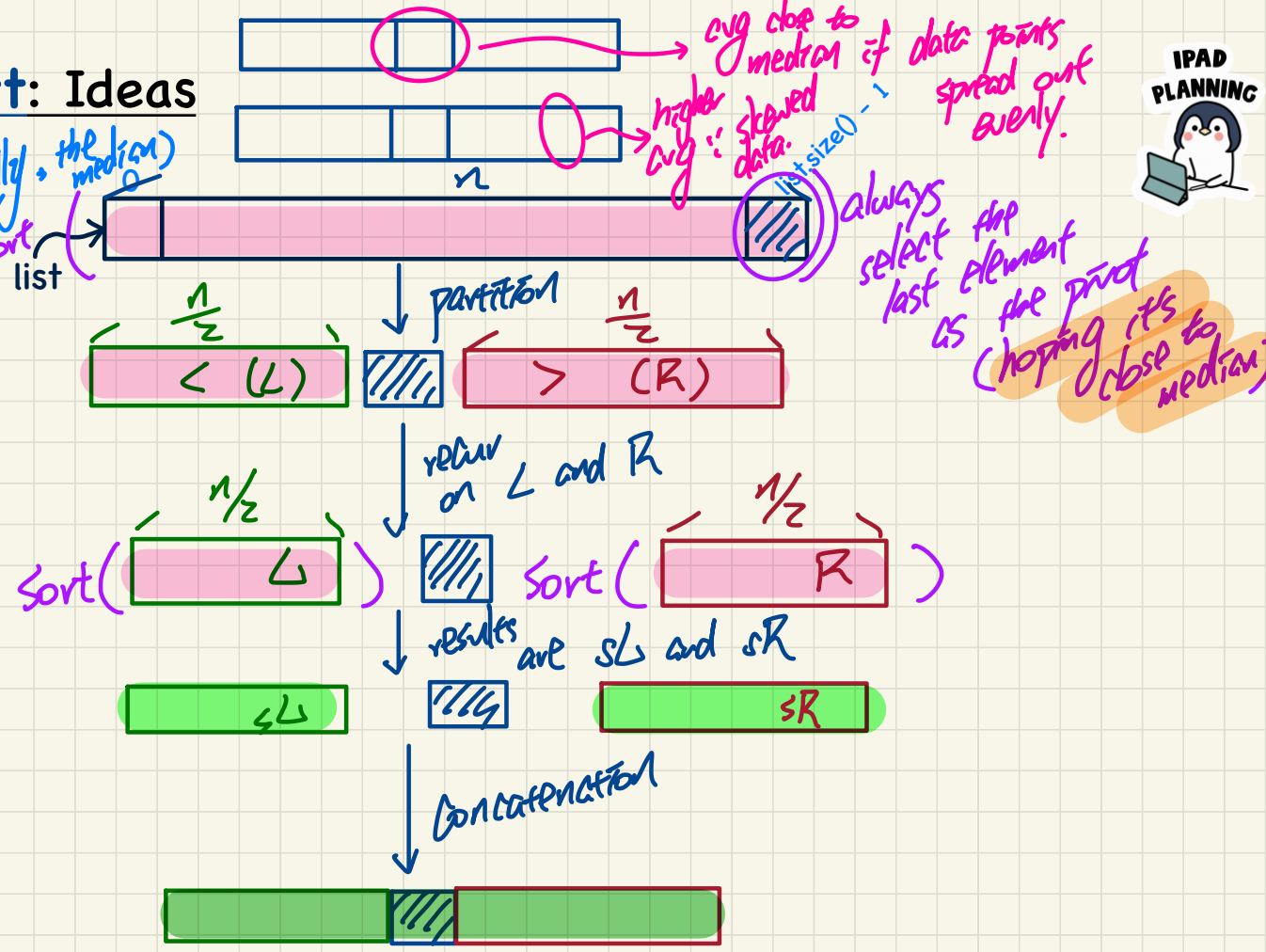
$I = \frac{n}{n} = \frac{n}{2^{\log n}}$

$n + \log n \cdot n$ is $O(n \cdot \log n)$

Quick Sort: Ideas

- pivot selection (ideally, the median)
- partition
- concatenation

median of medians algorithm → find median in O(n)



IPAD PLANNING

is merging sL and sR necessary? No!

Quick Sort in Java

Median of medians algo: O(n)
↳ as opposed to sort and select middle

```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue = list.get(pivotIndex);
    List<Integer> left = allLessThanOrEqualTo(pivotIndex, list);
    List<Integer> right = allLargerThan(pivotIndex, list);
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = new ArrayList<>();
    sortedList.addAll(sortedLeft);
    sortedList.add(pivotValue);
    sortedList.addAll(sortedRight);
  }
  return sortedList;
}
```

T(n)

O(1)

→ linear scan of the input list is necessary: O(n)

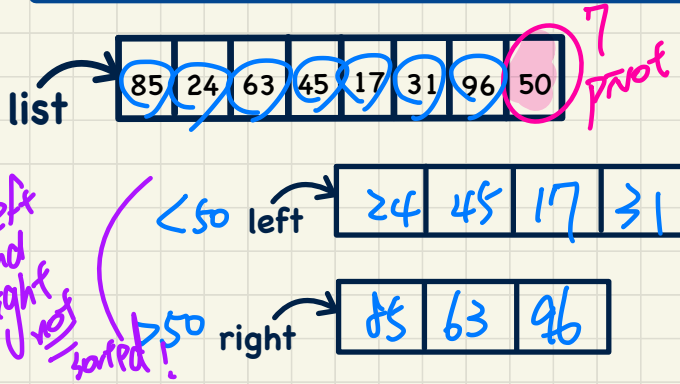
T(n/2) → only if pivot ≈ median

T(n/2)

Concat. O(1)

partition

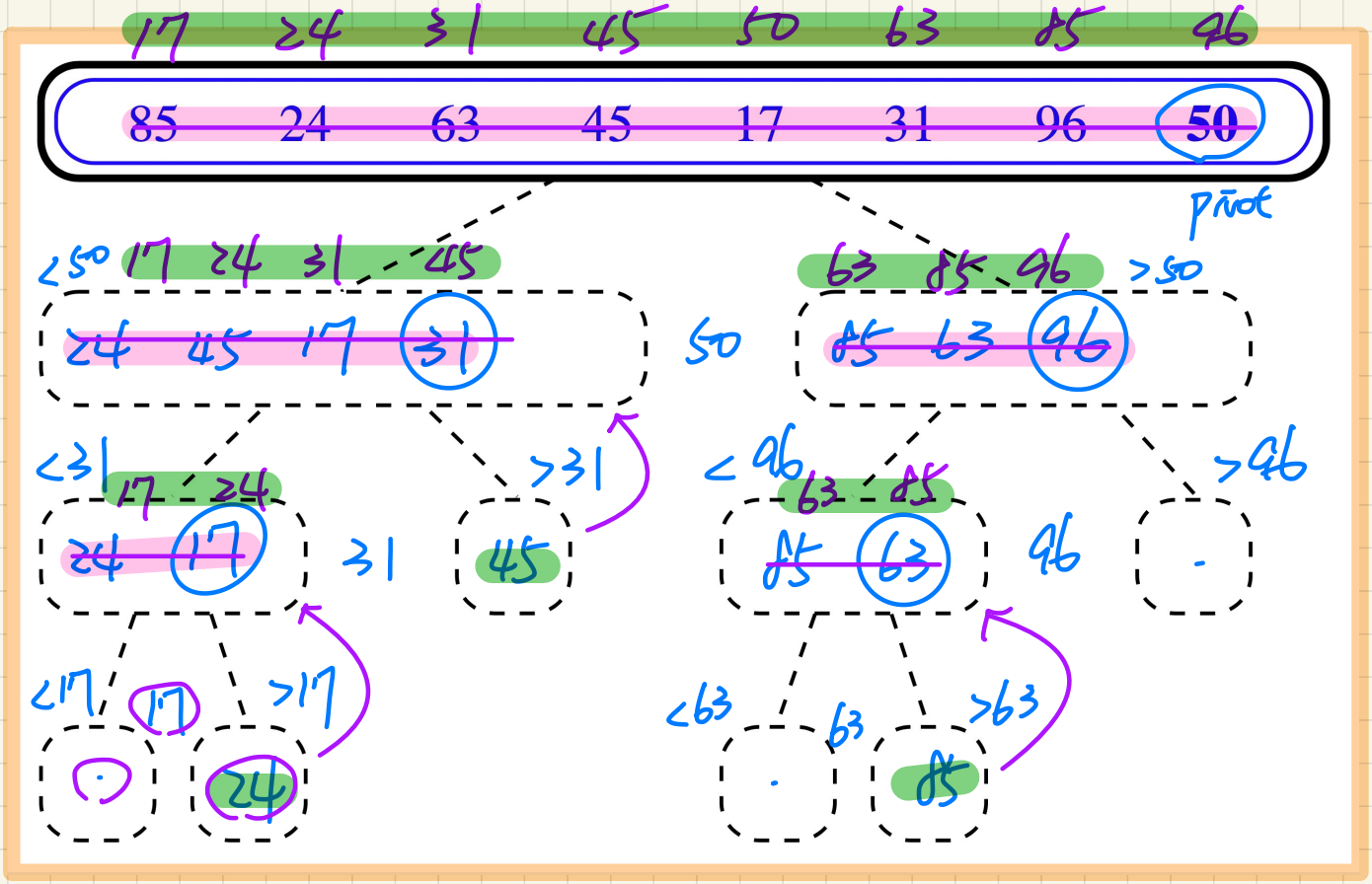
```
List<Integer> allLessThanOrEqualTo(int pivotIndex, List<Integer> list)
  List<Integer> sublist = new ArrayList<>();
  int pivotValue = list.get(pivotIndex);
  for(int i = 0; i < list.size(); i ++) {
    int v = list.get(i);
    if(i != pivotIndex && v <= pivotValue) { sublist.add(v); }
  }
  return sublist;
}
List<Integer> allLargerThan(int pivotIndex, List<Integer> list) {
  List<Integer> sublist = new ArrayList<>();
  int pivotValue = list.get(pivotIndex);
  for(int i = 0; i < list.size(); i ++) {
    int v = list.get(i);
    if(i != pivotIndex && v > pivotValue) { sublist.add(v); }
  }
  return sublist;
}
```



Quick Sort: Tracing

→ split partition

→ concatenate



Quick Sort: Worst-Case Running Time

```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue = list.get(pivotIndex);
    List<Integer> left = allLessThanOrEqualTo(pivotIndex, list);
    List<Integer> right = allLargerThan(pivotIndex, list);
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = new ArrayList<>();
    sortedList.addAll(sortedLeft);
    sortedList.add(pivotValue);
    sortedList.addAll(sortedRight);
  }
  return sortedList;
}
```

O(1)

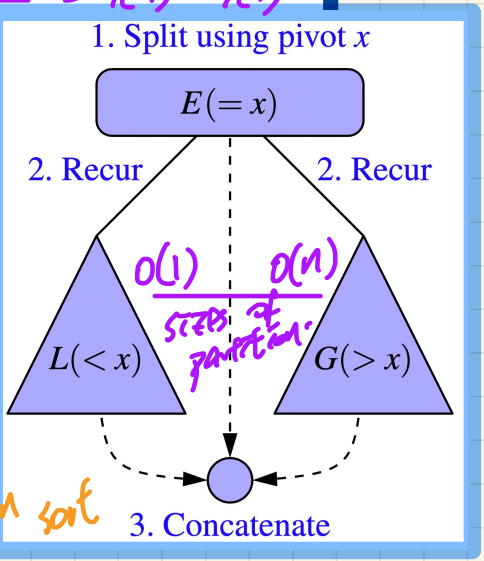
O(n)

O(1)

bad pivot >> median
<< median

|left| << |right|

T(n) T(1)



4 partitions

9 7 5 3 1

n-1

9 7 5 3

9 7 5

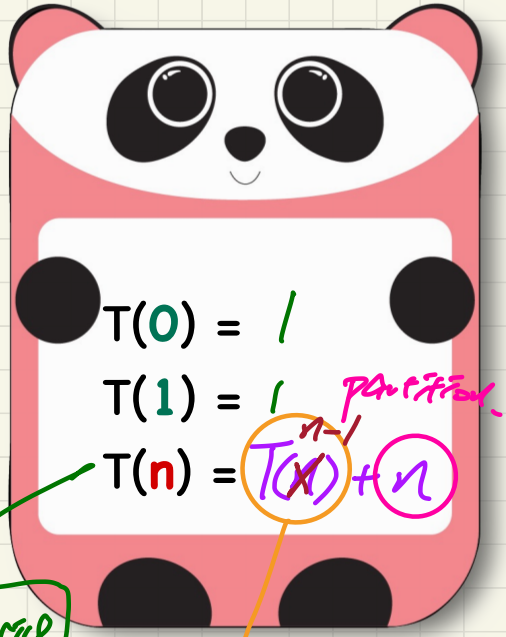
O(n)

9 7

9

overall RT is $O(n^2)$ ≈ selection sort, insertion sort

Running Time as a Recurrence Relation



Exercise

a half that's much larger than the other

Quick Sort: Best-Case Running Time

```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue = list.get(pivotIndex);
    List<Integer> left = allLessThanOrEqualTo(pivotIndex, list);
    List<Integer> right = allLargerThan(pivotIndex, list);
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = new ArrayList<>();
    sortedList.addAll(sortedLeft);
    sortedList.add(pivotValue);
    sortedList.addAll(sortedRight);
  }
  return sortedList;
}
```

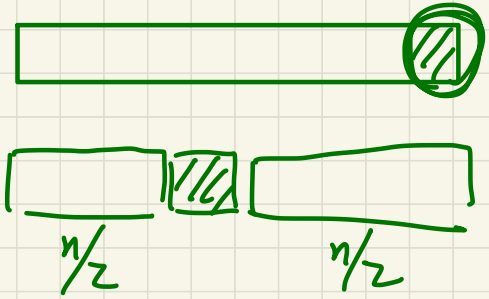
pivot ≈ median

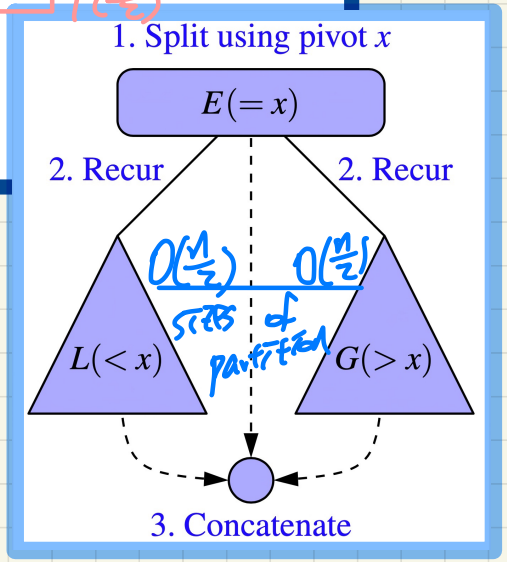
O(n)

O(1)

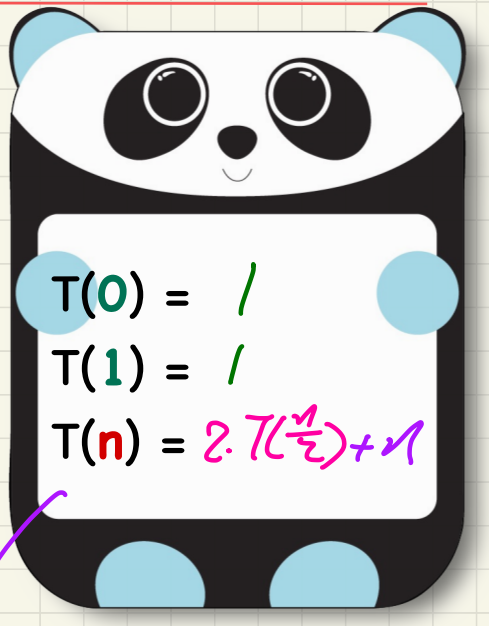
T(n/2)

T(n/2)





Running Time as a Recurrence Relation



Exercise